PJC Programming in C++

Contents

- 1. Introduction
  - 1.1 Compilers
    - 1.1.1 Linux
    - 1.1.2 Windows
  - 1.2 Literature

- 2. Let's start, then...
  - 2.1 Simplest C++ programs
  - 2.2 Simplest In/Out operations
    - 2.2.1 Outputting information
    - 2.2.2 Data input
  - 2.3 Functions
  - 2.4 Command-line arguments

- 3. Preprocesor
  - 3.1 What is the preprocessor?
  - 3.2 Preprocessor directives
  - 3.3 Predefined preprocessor macros

- 4. Basic data types
  - 4.1 Introductory remarks
  - 4.2 Integral types
    - 4.2.1 Useful aliases for integral types
  - 4.3 Floating-point types
  - 4.4 Logical type
  - 4.5 Enumeration types
  - 4.6 Pointers
    - 4.6.1 Pointers to variables
    - 4.6.2 Generic pointers
  - 4.7 References

- 5. Static arrays and pointers
  - 5.1 Defining arrays
  - 5.2 Array type
  - 5.3 Pointer arithmetic
  - 5.4 Character arrays (C-strings)
  - 5.5 Multidimensional arrays
    - 5.5.1 Matrices
    - 5.5.2 Arrays of C-strings
  - 5.6 Arrays of type *std::array*
  - 5.7 Vectors (*std::vector*)

- 6. “Compound” types

- 6.1 Defining “compound” types
- 6.2 *typedef* and *using* specifiers

- 7. Variables
  - 7.1 Keywords and names
  - 7.2 Scope and visibility
  - 7.3 Storage classes
    - 7.3.1 Static variables
    - 7.3.2 External variables
  - 7.4 Type modifiers
    - 7.4.1 Volatile variables
    - 7.4.2 Constants
  - 7.5 L-values

- 8. Statements
  - 8.1 Categories of statements
  - 8.2 Labels
  - 8.3 Declarations
  - 8.4 Null statement
  - 8.5 Compound statement
  - 8.6 Expression statement
  - 8.7 Conditional statement
  - 8.8 Selection (*switch*)statement (*switch*)
  - 8.9 Iteration statements (loops)
    - 8.9.1 *while* loop
    - 8.9.2 *do-while* loop
    - 8.9.3 *for* loop
    - 8.9.4 *Foreach* loop
  - 8.10 *continue* and *break* statements
  - 8.11 *goto* statement
  - 8.12 *return* statement
  - 8.13 Exception handling statements

- 9. Operators
  - 9.1 Precedence and associativity of operators
  - 9.2 Overview of operators
    - 9.2.1 Scope-resolution operators
    - 9.2.2 Operators of precedence 15
    - 9.2.3 Operators of precedence 14
    - 9.2.4 Operators of precedence 13
    - 9.2.5 Arithmetic operators
    - 9.2.6 Relational operators and comparisons
    - 9.2.7 Bitwise operators
    - 9.2.8 Logical operators
    - 9.2.9 Assignment operators
    - 9.2.10 Conditional operator
    - 9.2.11 Exception-throw operator
    - 9.2.12 Comma operator
    - 9.2.13 Alternative operator names

- 10. Standard conversions. Order of evaluation
  - 10.1 Conversions

- 10.2 Order of evaluation

- 11. Functions
  - 11.1 Introduction
  - 11.2 Declarations and definitions of functions
  - 11.3 Function call
  - 11.4 Default arguments
  - 11.5 Variable-length argument lists
  - 11.6 Reference arguments
  - 11.7 Functions returning references
  - 11.8 Recursive functions
  - 11.9 Static functions
  - 11.10 Inlined functions
  - 11.11 Function overloading
  - 11.12 Function pointers
  - 11.13 Lambda functions
  - 11.14 Function templates

- 12. Dynamic memory management
  - 12.1 Introduction
  - 12.2 Memory allocation — *new* operator
  - 12.3 Deallocation of memory — operator *delete*
  - 12.4 Dynamic multidimensional arrays
  - 12.5 Memory management in C
  - 12.6 Functions operating on memory
  - 12.7 Placement *new* operator

- 13. C-structures and unions
  - 13.1 C-structures
  - 13.2 Templates of structures
  - 13.3 Unions

- 14. Classes (I)
  - 14.1 Introduction
  - 14.2 Accessibility of class members
  - 14.3 Fields
  - 14.4 Methods
  - 14.5 Static member functions
  - 14.6 Constructors
  - 14.7 Destructors
  - 14.8 Creating objects
  - 14.9 Arrays of objects
  - 14.10 Bit fields

- 15. Classes (II)
  - 15.1 Constant methods
    - 15.1.1 *mutable* fields
  - 15.2 Volatile methods
  - 15.3 Constructors – further details
    - 15.3.1 Copy constructors
    - 15.3.2 Initialization lists
  - 15.4 Friend functions

- 15.5 Nested classes
- 15.6 Pointers to class members

- 16. Input/Output
  - 16.1 Streams
    - 16.1.1 Predefined streams
  - 16.2 Operators << and >>
  - 16.3 Formatting
    - 16.3.1 Format flags
    - 16.3.2 Manipulators
  - 16.4 Unformatted I/O operations
    - 16.4.1 Unformatted input
    - 16.4.2 Unformatted output
  - 16.5 Files
  - 16.6 Handling stream errors
  - 16.7 Internal files
    - 16.7.1 C-strings as internal files
    - 16.7.2 Internal files represented by C++ strings

- 17. Strings
  - 17.1 C-strings
    - 17.1.1 Functions on C-strings
    - 17.1.2 Functions operating on characters
    - 17.1.3 Conversion functions
  - 17.2 Strings in C++
    - 17.2.1 Constructors
    - 17.2.2 Methods and operators

- 18. Templates
  - 18.1 Templates of classes

- 19. Operator overloading. Move semantics and smart pointers
  - 19.1 Introduction
  - 19.2 Overloading with global functions
    - 19.2.1 Binary operators
    - 19.2.2 Unary operators
  - 19.3 Overloading with methods of classes
    - 19.3.1 Binary operators
    - 19.3.2 Unary operators
  - 19.4 “Special” operators
    - 19.4.1 Assignment operator
    - 19.4.2 Indexing operator
    - 19.4.3 Function call operator
    - 19.4.4 Overloading indirect member selection operator
  - 19.5 Move semantics
  - 19.6 Smart pointers
    - 19.6.1 Smart pointers of the *unique_ptr* type
    - 19.6.2 Smart pointers of the *shared_ptr* type

- 20. More on conversions
  - 20.1 Conversions to and from a user-defined type
    - 20.1.1 Conversions *to* a defined type

        - 20.1.2 Conversions *from* user-defined type
    - 20.2 Explicit conversions
        - 20.2.1 *const_cast* conversions
        - 20.2.2 Static casts
        - 20.2.3 Dynamic conversions
        - 20.2.4 Forced conversions

- 21. Inheritance and polymorphism
    - 21.1 Fundamentals of inheritance
    - 21.2 Constructors and destructors of derived classes
    - 21.3 Assignment operator in derived classes
    - 21.4 Virtual methods and polymorphism
    - 21.5 Abstract classes
    - 21.6 Virtual destructors
    - 21.7 Multiple inheritance

- 22. Exceptions
    - 22.1 Throwing exceptions
    - 22.2 Catching exceptions
    - 22.3 Hierarchies of exceptions
    - 22.4 Exceptions in constructors and destructors
    - 22.5 Exception specifications
    - 22.6 Rethrowing exceptions
    - 22.7 Standard exceptions

- 23. Modules and namespaces
    - 23.1 Modules of a program
        - 23.1.1 Headers and implementation files
    - 23.2 Namespaces

- 24. The Standard Library (STL)
    - 24.1 Collections and iterators
        - 24.1.1 Vectors
        - 24.1.2 Iterators
        - 24.1.3 Operations on collections
    - 24.2 Algorithms and function objects
        - 24.2.1 Algorithms
        - 24.2.2 Function objects
    - 24.3 Examples
    - 24.4 List of algorithm

- 25. Run-time type identification
    - 25.1 Operator *typeid*
    - 25.2 Operator *dynamic_cast*

- Index

---

1. Introduction

We will discuss the language as it is specified by the standard; that means, we will not be concerned with, undoubtedly very important, issues like graphics (building GUIs) or network programming. These aspects of C++ are beyond the standard, although the new standard (accepted in 2011) contains at least multi-threading and regular expressions. However, the material covered here should be sufficient to start studying textbooks, tutorials or manuals describing all these important issues without anyone's help.

In the course of our discussion, many elements of the C++ language will be introduced in fragments of the text where we analyse short examples of C++ code: these examples, therefore, should be considered part of the main text and should never be skipped!

C++ is a very rich programming language. Unfortunately, it is also rather complicated. This fact is, to some extend at least, a consequence of the assumption that it should be compatible with the (much older) C programming language. In principle, it should be possible to compile any correct C program using C++ compiler (but, of course, not *vice versa.*). I will sometimes make some comments on compatibility of these two related languages.

Another source of complication is the fact that C++ is a hybrid language: you can use its object oriented features but you are not obliged to do so — it is perfectly possible not to use any classes, objects, inheritance, hermetization or any other constructs specific to object oriented programming. It is also worth mentioning that the new standard introduced many constructs which up to now were characteristic rather to functional languages.

The new standard (accepted in 2011, extended in 2014; now most compilers support even the 2017 version) brings us many new elements in both the language itself and, to even more extend, in the the standard library. It is impossible to cover all these changes in this lecture – some of them, which seem the most important or useful, will be described, although sometimes only very briefly.

Subsections

- 1.1 Compilers
  - 1.1.1 Linux
  - 1.1.2 Windows
    - 1.1.2.1 Visual C++ and Studio .NET
- 1.2 Literature

Subsections

- 1.1.1 Linux
- 1.1.2 Windows
  - 1.1.2.1 Visual C++ and Studio .NET

1.1 Compilers

In order to run a C++ program, we will have to compile it to obtain its executable form (the same applies to many other languages, like Fortran, Ada, Pascal). Compilation is the process of transforming a text file with the program (the **source**) into **executable**: a file which contains the same program expressed in terms of instructions which are known to the processor of the computer. In contrast to Java, there is no intermediate form, the **byte code**, which would be platform independent (byte code, of course, also has to be eventually transformed into processor instructions: in Java this task is performed by Java Virtual Machine). The resulting program does not require any other external program to be executed (like the Java virtual machine or Python interpreter) — it is run directly under control of the operating system (the situation is a little different on the `.NET` platform). It can, however, require some library files to be installed.

As they do not need any additional translations during execution, C/C++ programs are usually fast (pure C programs are generally almost as fast as the corresponding Fortran programs; C++ ones are often, although not always, somewhat slower).

As a matter of fact,

> **compilation is only one of the steps leading from a source to the corresponding executable: for brevity we use the term *compilation* to denote the whole chain of actions which leads from a source (or sources) to an executable program.**

Other steps are **preprocessing** and **linking**. The latter is a process of combining several files resulting from compilation, perhaps together with some library files, into one executable file. We will not go into details of this process.

Of course, the executable program is platform dependent. However, if we observe the rules of the standard, the same source can be compiled and run on various platforms (as we have mentioned, graphics or networking are *not* covered by the standard of the C++ programming language: these elements usually *do* depend on platform). What we gain is better efficiency of the resulting code, as compared to, e.g., programs written in interpreted languages.

What is then important for us is to actually have a C++ compiler installed on our machine

(together with linker, libraries, etc.). We will therefore say a few words about available compilers. In order to test the one you will choose, the following program can be used:

P1: testInst.cpp Test of installation

```
/*
 * Test of installation. The program should print the names
 * of four programming languages (sorted alphabetically).
 */
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs{"Python", "Haskell",
    "C++",    "Java"};
    sort(vs.begin(),vs.end());
    for (const auto& e : vs) cout << e << " ";
    cout << endl;
}
```

It should print on your screen names of four programming languages sorted alphabetically. The program may be hard to understand now but its only purpose is to test if your installation of C++ (compiler, linker, libraries) works. You should put this program into separate directory and try to compile and run it; remember, that

for each project you should always create a separate directory

with thoroughly chosen name and localization; neglectig it is asking for trouble!

1.1.1 Linux

Normally, users of Linux do not have to install anything: they already have all what is needed, even if they are not aware of this fact. If, for some reason, appropriate packages have not been installed, you can install them with one command, the form of which is specific to your distribution (the same applies to Mac computers).

If you have a source file with a C++ program in the current directory, like `testInst.cpp` above, the command

```
g++ -o testInst testInst.cpp
```

should result in compilation (and linking) of the program. If there are no errors, the file `testInst` should appear in the same catalog. This is the executable that we need in order to run the program. Traditionally, in Linux world, names of executables do not have any extension, although this is not forbidden: you can name the ececutable as you wish (e.g., `testInst.exe`). The name is specified by the '`-o`' option, as above. Actually, you do not have to specify any name: if you don't, the default `a.out` will be used.

If our program uses new features, introduced in the 2011/2014 standards, it may be necessary to add an option '`-std=c++14`'; it is also highly recommended to add options

which enforce conformance with the standard, e.g.,

```
g++ -o testInst -std=c++14 -pedantic-errors -Wall testInst.cpp
```

Now we want to run the program. We do it by entering the name of the executable with '`./`' at the beginning, in our case '`./testInst`', and pressing `Enter`. Your session could look like this:

```
cpp> g++ -o testInst -pedantic-errors -Wall testInst.cpp
cpp> ./testInst
C++ Haskell Java Python
```

If the program consists of many files, you should put them all on the command line. Of course, it is ususally more convenient to employ *wild cards*; the command

```
cpp> g++ -o testInst *.cpp
```

will compile all source files from the current directory and produce one executable named **`testInst`**.

Additional information on compilation and linking are available on the *man* page ('`man g++`') or in **info** pages ('`info g++`').

In many editors it is possible to assign compilation to a key stroke; there are also graphical interfaces that one can use in tandem with a compiler (notably **Eclipse**, **Anjuta**, **Geany**, **KDevelop**, **CodeWarrior**, **Code::Blocks** — they can be easily located in the Internet). Most good text editors can help you write C++ programs by providing syntax highlighting, folding, automatic indentation etc.

1.1.2 Windows

1.1.2.1 Visual C++ and Studio .NET

A good C++ compiler is included in the Microsoft's Visual Studio (there is a free version called **Visual StudioExpress**). It comes with rich graphical interface, examples, help files, debugger, etc. Those using this tool should only remember to use proper options when creating the project: for our simple examples you should choose the option '`Emty project`'; otherwise you could get rather obscure error messages during compilation. The compiler can be invoked from the graphical interface or directly from the command line using the command **cl** (the abbreviation comes from *compile and link*). There are many compilation options that can be selected; as a minimum it would be worthwile to select '`-Wall -Za -GR -EHsc`' which will, among other things, switch on warnings and switch off language extensions.

Subsections

- 1.1.1 Linux
- 1.1.2 Windows
  - 1.1.2.1 Visual C++ and Studio .NET

1.1 Compilers

In order to run a C++ program, we will have to compile it to obtain its executable form (the same applies to many other languages, like Fortran, Ada, Pascal). Compilation is the process of transforming a text file with the program (the **source**) into **executable**: a file which contains the same program expressed in terms of instructions which are known to the processor of the computer. In contrast to Java, there is no intermediate form, the **byte code**, which would be platform independent (byte code, of course, also has to be eventually transformed into processor instructions: in Java this task is performed by Java Virtual Machine). The resulting program does not require any other external program to be executed (like the Java virtual machine or Python interpreter) — it is run directly under control of the operating system (the situation is a little different on the `.NET` platform). It can, however, require some library files to be installed.

As they do not need any additional translations during execution, C/C++ programs are usually fast (pure C programs are generally almost as fast as the corresponding Fortran programs; C++ ones are often, although not always, somewhat slower).

As a matter of fact,

> **compilation is only one of the steps leading from a source to the corresponding executable: for brevity we use the term *compilation* to denote the whole chain of actions which leads from a source (or sources) to an executable program.**

Other steps are **preprocessing** and **linking**. The latter is a process of combining several files resulting from compilation, perhaps together with some library files, into one executable file. We will not go into details of this process.

Of course, the executable program is platform dependent. However, if we observe the rules of the standard, the same source can be compiled and run on various platforms (as we have mentioned, graphics or networking are *not* covered by the standard of the C++ programming language: these elements usually *do* depend on platform). What we gain is better efficiency of the resulting code, as compared to, e.g., programs written in interpreted languages.

What is then important for us is to actually have a C++ compiler installed on our machine

(together with linker, libraries, etc.). We will therefore say a few words about available compilers. In order to test the one you will choose, the following program can be used:

P1: testInst.cpp Test of installation

```
/*
 * Test of installation. The program should print the names
 * of four programming languages (sorted alphabetically).
 */
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs{"Python", "Haskell",
    "C++",    "Java"};
    sort(vs.begin(),vs.end());
    for (const auto& e : vs) cout << e << " ";
    cout << endl;
}
```

It should print on your screen names of four programming languages sorted alphabetically. The program may be hard to understand now but its only purpose is to test if your installation of C++ (compiler, linker, libraries) works. You should put this program into separate directory and try to compile and run it; remember, that

for each project you should always create a separate directory

with thoroughly chosen name and localization; neglectig it is asking for trouble!

1.1.1 Linux

Normally, users of Linux do not have to install anything: they already have all what is needed, even if they are not aware of this fact. If, for some reason, appropriate packages have not been installed, you can install them with one command, the form of which is specific to your distribution (the same applies to Mac computers).

If you have a source file with a C++ program in the current directory, like `testInst.cpp` above, the command

```
g++ -o testInst testInst.cpp
```

should result in compilation (and linking) of the program. If there are no errors, the file `testInst` should appear in the same catalog. This is the executable that we need in order to run the program. Traditionally, in Linux world, names of executables do not have any extension, although this is not forbidden: you can name the ececutable as you wish (e.g., `testInst.exe`). The name is specified by the '`-o`' option, as above. Actually, you do not have to specify any name: if you don't, the default `a.out` will be used.

If our program uses new features, introduced in the 2011/2014 standards, it may be necessary to add an option '`-std=c++14`'; it is also highly recommended to add options

which enforce conformance with the standard, e.g.,

```
g++ -o testInst -std=c++14 -pedantic-errors -Wall testInst.cpp
```

Now we want to run the program. We do it by entering the name of the executable with '`./`' at the beginning, in our case '`./testInst`', and pressing `Enter`. Your session could look like this:

```
cpp> g++ -o testInst -pedantic-errors -Wall testInst.cpp
cpp> ./testInst
C++ Haskell Java Python
```

If the program consists of many files, you should put them all on the command line. Of course, it is ususally more convenient to employ *wild cards*; the command

```
cpp> g++ -o testInst *.cpp
```

will compile all source files from the current directory and produce one executable named **`testInst`**.

Additional information on compilation and linking are available on the *man* page ('`man g++`') or in **info** pages ('`info g++`').

In many editors it is possible to assign compilation to a key stroke; there are also graphical interfaces that one can use in tandem with a compiler (notably **Eclipse**, **Anjuta**, **Geany**, **KDevelop**, **CodeWarrior**, **Code::Blocks** — they can be easily located in the Internet). Most good text editors can help you write C++ programs by providing syntax highlighting, folding, automatic indentation etc.

1.1.2 Windows

1.1.2.1 Visual C++ and Studio .NET

A good C++ compiler is included in the Microsoft's Visual Studio (there is a free version called **Visual StudioExpress**). It comes with rich graphical interface, examples, help files, debugger, etc. Those using this tool should only remember to use proper options when creating the project: for our simple examples you should choose the option '`Emty project`'; otherwise you could get rather obscure error messages during compilation. The compiler can be invoked from the graphical interface or directly from the command line using the command **cl** (the abbreviation comes from *compile and link*). There are many compilation options that can be selected; as a minimum it would be worthwile to select '`-Wall -Za -GR -EHsc`' which will, among other things, switch on warnings and switch off language extensions.

Subsections

- 1.1.1 Linux
- 1.1.2 Windows
  - 1.1.2.1 Visual C++ and Studio .NET

1.1 Compilers

In order to run a C++ program, we will have to compile it to obtain its executable form (the same applies to many other languages, like Fortran, Ada, Pascal). Compilation is the process of transforming a text file with the program (the **source**) into **executable**: a file which contains the same program expressed in terms of instructions which are known to the processor of the computer. In contrast to Java, there is no intermediate form, the **byte code**, which would be platform independent (byte code, of course, also has to be eventually transformed into processor instructions: in Java this task is performed by Java Virtual Machine). The resulting program does not require any other external program to be executed (like the Java virtual machine or Python interpreter) — it is run directly under control of the operating system (the situation is a little different on the `.NET` platform). It can, however, require some library files to be installed.

As they do not need any additional translations during execution, C/C++ programs are usually fast (pure C programs are generally almost as fast as the corresponding Fortran programs; C++ ones are often, although not always, somewhat slower).

As a matter of fact,

> **compilation is only one of the steps leading from a source to the corresponding executable: for brevity we use the term *compilation* to denote the whole chain of actions which leads from a source (or sources) to an executable program.**

Other steps are **preprocessing** and **linking**. The latter is a process of combining several files resulting from compilation, perhaps together with some library files, into one executable file. We will not go into details of this process.

Of course, the executable program is platform dependent. However, if we observe the rules of the standard, the same source can be compiled and run on various platforms (as we have mentioned, graphics or networking are *not* covered by the standard of the C++ programming language: these elements usually *do* depend on platform). What we gain is better efficiency of the resulting code, as compared to, e.g., programs written in interpreted languages.

What is then important for us is to actually have a C++ compiler installed on our machine

(together with linker, libraries, etc.). We will therefore say a few words about available compilers. In order to test the one you will choose, the following program can be used:

P1: testInst.cpp Test of installation

```
/*
 * Test of installation. The program should print the names
 * of four programming languages (sorted alphabetically).
 */
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs{"Python", "Haskell",
    "C++",    "Java"};
    sort(vs.begin(),vs.end());
    for (const auto& e : vs) cout << e << " ";
    cout << endl;
}
```

It should print on your screen names of four programming languages sorted alphabetically. The program may be hard to understand now but its only purpose is to test if your installation of C++ (compiler, linker, libraries) works. You should put this program into separate directory and try to compile and run it; remember, that

for each project you should always create a separate directory

with thoroughly chosen name and localization; neglectig it is asking for trouble!

1.1.1 Linux

Normally, users of Linux do not have to install anything: they already have all what is needed, even if they are not aware of this fact. If, for some reason, appropriate packages have not been installed, you can install them with one command, the form of which is specific to your distribution (the same applies to Mac computers).

If you have a source file with a C++ program in the current directory, like `testInst.cpp` above, the command

```
g++ -o testInst testInst.cpp
```

should result in compilation (and linking) of the program. If there are no errors, the file `testInst` should appear in the same catalog. This is the executable that we need in order to run the program. Traditionally, in Linux world, names of executables do not have any extension, although this is not forbidden: you can name the ececutable as you wish (e.g., `testInst.exe`). The name is specified by the '`-o`' option, as above. Actually, you do not have to specify any name: if you don't, the default `a.out` will be used.

If our program uses new features, introduced in the 2011/2014 standards, it may be necessary to add an option '`-std=c++14`'; it is also highly recommended to add options

which enforce conformance with the standard, e.g.,

```
g++ -o testInst -std=c++14 -pedantic-errors -Wall testInst.cpp
```

Now we want to run the program. We do it by entering the name of the executable with '`./`' at the beginning, in our case '`./testInst`', and pressing `Enter`. Your session could look like this:

```
cpp> g++ -o testInst -pedantic-errors -Wall testInst.cpp
cpp> ./testInst
C++ Haskell Java Python
```

If the program consists of many files, you should put them all on the command line. Of course, it is ususally more convenient to employ *wild cards*; the command

```
cpp> g++ -o testInst *.cpp
```

will compile all source files from the current directory and produce one executable named **`testInst`**.

Additional information on compilation and linking are available on the *man* page ('`man g++`') or in **info** pages ('`info g++`').

In many editors it is possible to assign compilation to a key stroke; there are also graphical interfaces that one can use in tandem with a compiler (notably **Eclipse**, **Anjuta**, **Geany**, **KDevelop**, **CodeWarrior**, **Code::Blocks** — they can be easily located in the Internet). Most good text editors can help you write C++ programs by providing syntax highlighting, folding, automatic indentation etc.

1.1.2 Windows

1.1.2.1 Visual C++ and Studio .NET

A good C++ compiler is included in the Microsoft's Visual Studio (there is a free version called **Visual StudioExpress**). It comes with rich graphical interface, examples, help files, debugger, etc. Those using this tool should only remember to use proper options when creating the project: for our simple examples you should choose the option '`Emty project`'; otherwise you could get rather obscure error messages during compilation. The compiler can be invoked from the graphical interface or directly from the command line using the command **cl** (the abbreviation comes from *compile and link*). There are many compilation options that can be selected; as a minimum it would be worthwile to select '`-Wall -Za -GR -EHsc`' which will, among other things, switch on warnings and switch off language extensions.

1.2 Literature

There are many good books on C++. We will mention just a few:

C++ Primer
Stanley B. Lippman, Josée Lajoie, Barbara E. Moo
The C++ Programming Language
Bjarne Stroustrup
The C++ Standard Library, 2nd Edition
N.M. Josuttis
Algorithms in C++
R. Sedgewick

The Internet remains, of course, an inexhaustible source of information on C/C++; on hundreds of web pages you can find examples, tutorials, specifications etc.

2. Let's start, then...

In this chapter, we will analyze the structure of the simplest C++ programs. We will also learn how one can write out data from a program to the screen and how one can read data entered from the keyboard. Generally, such *input/output* operations are far from being trivial; nevertheless, we have to deal with them right from the beginning, to a limited extend at least, if we want to be able to write even a very simple examples.

Subsections

- 2.1 Simplest C++ programs
- 2.2 Simplest In/Out operations
  - 2.2.1 Outputting information
  - 2.2.2 Data input
- 2.3 Functions
- 2.4 Command-line arguments

2.1 Simplest C++ programs

As always in such cases, we will start from the simplest program, namely the **Hello, World** program. Looking at this and other C++ programs, those who know Java, (or, e.g., PHP) will easily notice the resemblance of its syntax to the syntax of C/C++. Therefore, they will not have to learn from scratch. Those who do not know other languages will be in a somewhat worse position. They can be comforted, however, by the fact that it will be much easier for them to learn those other languages in the future, when they have already learned C/C++.

The resemblance between C++ and other languages does not apply to input/output operations. These are very specific for C++. Unfortunately, they are also quite different in pure C and in C++. We will mainly use constructs from C++, but the reader is encouraged to find and learn C-specific mechanisms, which can be, and quite often are, used also in C++ (it is a quite common practice to mix C++ with traditional C-specific constructs, which sometimes are more effective).

Source files with C++ code have usually names with extensions '`.cpp`', '`.cxx`', or '`.C`', but other forms are used as well. We will use '`.cpp`', as it is the most popular convention. Pure C source files have extension '`.c`',. A special kind of source files which we will discuss later, the so called *header files*, have extension '`.h`' or, in C++, they have no extension at all.

Let us then consider a file with the respectable **Hello, World** program (B. Kernighan, 1973):

P2: helloWorld.cpp 'Hello, World' in C++

```
1.  #include <iostream>                    ❶
2.  using namespace std;                   ❷
3.
4.  /*
5.      We always start with
6.      program Hello, World!
7.   */
8.
9.  int main() { // Comments as in Java
10.     cout << "Hello, World" << endl;     ❸
11. }
```

As you certainly have guuessed, the program will write to the screen the phrase '`Hello, World!`'.

> Those who know Java will notice that the structure of the program is somewhat different from a corresponding Java program. There are no classes here: the function named **main** is *global* here.

Generally, a program in C++ consists of one or more *modules* written in one or several files. Each such module contains preprocessor directives (if they are needed), declarations

and definitions of classes, variables, functions, and, of course, comments. Preprocessor directives start with the character ” as the first non-blank character of a line. We can see an example in our program: `#include <iostream>` at the very beginning. These lines are processed by a program called **preprocessor** *before* compilation. The preprocessor modifies the *text* of the program: it does not analyze the program itself. After these stage a modified text of the program is sent to the compiler proper — the compiler will never even see the lines starting with ”!

Exactly one module of every program has to contain a function with the name **main**. Generally, execution of a program amounts to executing instructions in this function. Therefore,

```
int main() {
    return 0;
}
```

or even

```
int main(){}
```

Let's have a closer look at our **Hello, World** program from file **`helloWorld.cpp`**, as it demonstrates some basic elements of any C++ program:

`#include...` (❶)

This line instructs the preprocessor (*not* the compiler) to include the file **`iostream`** into the program. The text of this file replaces the line starting with `#include` and will be seen by the compiler exactly as if we had have typed this text into the program ourselves. In our case the file **`iostream`** will be included (a file included in this way is called **header file**). It defines (or at least declares — more about it later) tools needed for input/output operations that we will use in the program (in particular it defines tools which are necessary for writing data on screen and reading it from keyboard). Let us notice that the file is really included and then compiled: it is therefore very different from **import** in Java, which merely informs the compiler where to look for definitions of entities (classes) which are not defined within the current file.

The file **`iostream`** itself is located in a catalog known to the preprocessor (this is indicated by the use of angle braces (`<...>`). The catalog containing such special files is supplied by the compiler's vendor together with the compiler. We can also include our own files in this way: to do it we use double quotation marks instead of angle brackets. Thus, `#include <bib.h>` includes standard header file (provided by the compiler's vendor), while `#include "bib.h"` would look for a user's file **`bib.h`** in current directory, and only then, if not found, in the standard catalog, exactly as in the previous case. The directive `#include` is *not* intended for the compiler. It will be seen by the preprocessor only, whose task is to modify the *text* of the source program and send the result to the compiler. We will have more to say about the preprocessor in one of the next chapters.

`using namespace std;` (❷)

This line informs the compiler that names (of objects, classes, functions) from the namespace ***std*** should be “imported” (made visible) into the current (default) namespace. It is used here because names of various entities from the file **`iostream`** all belong to this namespace.

We could have omitted the **using** directive here, but then we would have to write, e.g., `std::cout`, `std::endl` instead of simply `cout` and `endl`.

Another solution, more recommended, would be to use the so called using-declarations and import not all names from the namespace ***std***, but selectively only the ones we

need; e.g., `using std::cout;`.
We wil cover namespaces in more detail in chapter on namespaces .

Comments (lines 4-7 and 9)

Multi-line comments are delimited by '`/*`' (beginning) and '`*/`' (end). Everything inside, together with the delimiting symbols themselves, is ignored: the compiler will see just one blank at this place (this means that you cannot insert comments “into” a keyword or identifier). Comments cannot be nested (some compilers support such nesting, though). In C++, as in Java, we can use another form of comments: everything from and including two consecutive slashes, '`//`' (without any intervening blanks), through the end of the current line is a comment (see line 9 for an example).

main function (lines 9-11)

The **main** has to return (“evaluate to”) a value of type **int**, i.e., an integer number. We denote it by specifying the return type **int** *before* the function's name (which is **main**). Let us notice that **main** is always *global*: it is not contained in any class (as it would be in Java). The returned value (of integer type) is passed, after the function's completion, to the “caller”: a program or a script that invoked the program. If the program finishes its task successfully, it should return zero (by convention; this is not obligatory). If something went wrong, a nonzero positive value not exceeding 255 should be returned. This value may convey some useful information about the nature of failure to the caller (e.g., a shell script).

Normally, when a function declares that it returns a value (**main** does), it must return it explicitly. In this respect, the **main** function is exceptional — if there is no **return** statement, one will be silently added by the compiler at the very end of the function.

Function **main** plays the rôle of the entry point to the program. It will be the first function called. When the control flow exits it — e.g., because the **return** statement or the closing brace of the function's body has been encountered — the program stops (we disregard here multi-threaded programs which behave differently and some other subtleties, not necessary for us here).

The **main** function may have parameters through which the system passes command-line arguments to the program. As the first argument, the name of the program itself is passed: this means, that there is always at least one command-line argument. We will say more about this mechanism shortly. If we do not need command-line arguments in our program, the list of parameters may be empty. The list is given in parentheses after the name of the function — even if it is empty, the parentheses cannot be omitted. To emphasis that it is empty, the keyword **void** may be used in place of a list: '`int main(void)...`'.

Blocking (lines 9-11)

Everything between braces ('`{`' and '`}`') on lines 9 and 11 is the definition of the **main** function. It comes after declaring its name, parameters and return type in its **header line**, i.e., line 9 of our program. Embracing a sequence of statements in a pair of braces creates the so called **compound statement**. This is necessary if the syntax permits a single statement, but we need more. In particular, defining a function (the same will apply to classes, unions,...) we must put its body between braces: in this case we have to do it even if the function's body contains only one statement (or even none at all, what is possible in some circumstances). Blocking sequences of statements by using braces will affect scoping and visibility — more about it later.

Statements (❸)

The definition of a function consists of a sequence of statements, each of which is terminated by a semicolon. White space (spaces, tabs, new-line characters) can be used almost everywhere to enhance legibility of the program (of course, keywords and identifiers — names of variables, functions etc. — cannot be broken by a white space). The compiler will know where a statement ends by the presence of a semicolon.

In our case there is only one statement. It tells the computer to print on the screen a string (sequence of characters) contained in the double quotation marks. To achieve this task, we used a mechanism specific to C++ (and absent in C). In order to use it, we had to include the header file `iostream` at the very beginning of the program. Input/output operation are generally difficult and complicated (not only in C/C++). We will study them in more details in a special chapter . Below, we will only provide some basic information that should make it possible to start writing simple programs.

Let us also note that in C++, which is basically a procedural language, instructions are executed one by one, as they appear in the text of a program (although some repetitions and jumps are possible, as we will see).

Before we delve into intricacies of the C++ language, let us devote a moment to a rather rudimentary introduction to input/output operations.

Prev: 2. Let's start, then... **Up:** 2. Let's start, then... **Next:** 2.2 Simplest In/Out operations

Subsections

- 2.2.1 Outputting information
- 2.2.2 Data input

2.2 Simplest In/Out operations

After we have included the header file `iostream`, we have at our disposal various objects defined there (in fact declared only, but that is not our concern now). Among them, there are two objects named ***cout*** and ***cin***. They represent what is called standard streams: **standard input stream**, a.k.a. *stdin* (***cin***) and **standard output stream**, a.k.a. *stdout* (***cout***).

2.2.1 Outputting information

The object ***cout*** represents standard output stream. It is a predefined object of the class **ostream**, whose definition is visible due to inclusion of `iostream`. It is responsible for outputting information from our program to the “outer world”. In this case, the rôle of the outer world is played by the screen of our computer — ***cout*** stands for *console output* (in Java a similar object is called ***System.out***).

As ***cout*** is an object, we can invoke various *methods* (functions) on it. We will learn more about them in chapter on input/output operations. For now it will suffice to know how we can use ***cout*** to be able to insert a piece of information — actually, a string of characters — into the stream of information flowing from the program to the computer's screen, or whatever the operating system considers to be the *sink* of the standard output stream at the moment. We can do it with the help of the **stream-insertion operator**, '<<' (two consecutive *less than* symbols). We can see an example in the line ❸ of the program `helloWorld.cpp`. We then write

```
cout << "Hello, World" << endl;
```

to insert a string of characters (given here literally by enclosing it in double quotation marks) to the output stream represented by ***cout***. After the next token '<<', we send to the stream something which is called ***endl***. This is not enclosed in quotation marks, so we know that it is not a string " ***endl***" that we want to appear on the screen. This is a so called *manipulator* which also became visible thank to inclusion of `iostream`. Inserting it into an output stream will add the end-of-line character (*line feed*), so the cursor on the screen will be set at the beginning of the next line. We did not have to add it but if we wouldn't have done it, the next output or input would start in the same line right after the displayed text `"Hello, World"`. We would get a similar effect by inserting the end-of-line character directly at the end of the string to be displayed: `"Hello, World\n"` The symbol '`\n`' (consisting of two characters: the backslash and the letter '`n`') within a literal string denotes the line feed character. Actually, the effect of ***endl*** and '`\n`' are similar but not identical: inserting ***endl*** to the output stream not only inserts line feed character but also *flushes* the

output buffer, so the result appears on the screen immediately (otherwise the system could wait for more data before displaying them).

In this simple example, we have put two objects into the stream: a string of characters and a manipulator. We could have inserted more. Note however that each piece of data must be inserted separately by using '<<' operator. It would be *illegal* to separate them by commas:

```
cout << "Hello, World" , endl; // NO!!!
```

The following three lines, however, would be legal

```
cout << "The first line,"        << endl
     << "the second,"            << endl
     << "and finally the third." << endl;
```

Let us notice that this is *one* statement, outputting three strings and three end-of-line characters.

Those who know Java will remember that in this language we can write

```
int k = 7;
double x = 8.6;
System.out.println(k);
System.out.println(x);
```

This snippet of code will write out numbers 7 and 8.6, although the type of argument is in both cases different and none of them is a reference to an object of class **String**. This is possible due to overloading of the **println** function. Similar mechanism works also in C++ for the '<<' operator. We can insert to the output stream not only strings of characters but also values of variables of other types for which the '<<' operator has been defined — these are in particular all built-in types, but also other types defined in standard C++ libraries. For example, we can do it with objects of class **string** from the standard library, as in the following program

P3: john.cpp '<<' operator

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. int main() {
6.     double weight = 76.5;
7.     int    height = 182;
8.     string name   = "John";
9.     cout << name   << " weighs "  << weight << " kg and is "
10.         << height << " cm tall." << endl;
11. }
```

which outputs

```
John weighs 76.5 kg and is 182 cm tall
```

The class **string** used in this example will be described in the chapter on strings in C++. We will come to other new elements appearing in this example very shortly.

2.2.2 Data input

To input data from the “outer world” to the program, we use a similar mechanism but now applied to the standard input stream represented by the object ***cin*** (of class **istream**). By default input data flows from the keyboard, which plays the rôle of the source of the stream (***cin*** stands for *console input*).

As with output streams, we will confine ourselves to simple **extraction** of data from the input stream. We can do it with the help of the **stream-extraction operator**, '>>' (two consecutive *greater than* symbols), as in

P4: read.cpp '>' operator

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  int main() {
6.      string name;
7.      int height;
8.      double weight;
9.      cout << "Enter your name, height and weight: ";
10.     cin  >> name >> height >> weight;
11.     cout << name << ", you are " << height  << " cm tall "
12.          << "and you weigh " << weight << " kg" << endl;
13. }
```

which gives, when run:

```
cpp> ./read
Enter your name, height and weight: Tom 182 76.5
Tom, you are 182 cm tall and you weigh 76.5 kg
cpp>
```

Let us note that:

- Direction of “arrows” shows the direction of information flow: for ***cout*** to the left, towards the ***cout***, i.e., the screen; for ***cin*** information flows “out of” ***cin***, i.e., from the keyboard.
- Each piece of information is extracted separately (you cannot use just commas to separate them).
- The type of data must be compatible with the declared type of variables in which the data is to be stored. In the example above, the program expects first a string (which is to be stored in a variable of the type very appropriately named **string**), then a whole number (this type is called **int**), and then a real number, possibly with non-vanishing fractional part (in C++ it is type **double**). Entering a real number — with a dot separating the integer and fractional parts — as the second data, when a whole number is expected, will cause an error which can escape unnoticed if we do not check it ourselves.
- Consecutive items are separated by an arbitrary long nonempty sequences of white spaces (blank – SP, tabulator – HT, new-line character – LF, carriage return – CR). White spaces before the first item are ignored; after that each such sequence is a separator. It follows that in this way it is not possible to enter strings with embedded spaces! Suppose we have a string variable ***nameSurname*** and we try to read in its value:
  `cin >> nameSurname;`
  Then, after entering '`John Doe`' on the keyboard, we would get '`John`' as the value of

nameSurname and '`Doe`' would remain in the keyboard buffer waiting for the next input operation (most probably provoking an error).

Input/output operation are quite complicated in every programming language. This is particularly true when reading data from the keyboard, when user's imagination in entering wrong and invalid data is virtually unlimited. We will postpone the details to the special chapter on input/output operations. In particular, in the section on input/output operations on files, we will show how to read from or write to text files.

Prev: 2.1 Simplest C++ programs **Up:** 2. Let's start, then... **Next:** 2.3 Functions

Subsections

- 2.2.1 Outputting information
- 2.2.2 Data input

2.2 Simplest In/Out operations

After we have included the header file `iostream`, we have at our disposal various objects defined there (in fact declared only, but that is not our concern now). Among them, there are two objects named ***cout*** and ***cin***. They represent what is called standard streams: **standard input stream**, a.k.a. *stdin* (***cin***) and **standard output stream**, a.k.a. *stdout* (***cout***).

2.2.1 Outputting information

The object ***cout*** represents standard output stream. It is a predefined object of the class **ostream**, whose definition is visible due to inclusion of `iostream`. It is responsible for outputting information from our program to the “outer world”. In this case, the rôle of the outer world is played by the screen of our computer — ***cout*** stands for *console output* (in Java a similar object is called ***System.out***).

As ***cout*** is an object, we can invoke various *methods* (functions) on it. We will learn more about them in chapter on input/output operations. For now it will suffice to know how we can use ***cout*** to be able to insert a piece of information — actually, a string of characters — into the stream of information flowing from the program to the computer's screen, or whatever the operating system considers to be the *sink* of the standard output stream at the moment. We can do it with the help of the **stream-insertion operator**, '<<' (two consecutive *less than* symbols). We can see an example in the line ❸ of the program `helloWorld.cpp`. We then write

```
cout << "Hello, World" << endl;
```

to insert a string of characters (given here literally by enclosing it in double quotation marks) to the output stream represented by ***cout***. After the next token '<<', we send to the stream something which is called ***endl***. This is not enclosed in quotation marks, so we know that it is not a string " ***endl***" that we want to appear on the screen. This is a so called *manipulator* which also became visible thank to inclusion of `iostream`. Inserting it into an output stream will add the end-of-line character (*line feed*), so the cursor on the screen will be set at the beginning of the next line. We did not have to add it but if we wouldn't have done it, the next output or input would start in the same line right after the displayed text `"Hello, World"`. We would get a similar effect by inserting the end-of-line character directly at the end of the string to be displayed: `"Hello, World\n"` The symbol '`\n`' (consisting of two characters: the backslash and the letter '`n`') within a literal string denotes the line feed character. Actually, the effect of ***endl*** and '`\n`' are similar but not identical: inserting ***endl*** to the output stream not only inserts line feed character but also *flushes* the

output buffer, so the result appears on the screen immediately (otherwise the system could wait for more data before displaying them).

In this simple example, we have put two objects into the stream: a string of characters and a manipulator. We could have inserted more. Note however that each piece of data must be inserted separately by using '<<' operator. It would be *illegal* to separate them by commas:

```
cout << "Hello, World" , endl; // NO!!!
```

The following three lines, however, would be legal

```
cout << "The first line,"        << endl
     << "the second,"            << endl
     << "and finally the third." << endl;
```

Let us notice that this is *one* statement, outputting three strings and three end-of-line characters.

Those who know Java will remember that in this language we can write

```
int k = 7;
double x = 8.6;
System.out.println(k);
System.out.println(x);
```

This snippet of code will write out numbers 7 and 8.6, although the type of argument is in both cases different and none of them is a reference to an object of class **String**. This is possible due to overloading of the **println** function. Similar mechanism works also in C++ for the '<<' operator. We can insert to the output stream not only strings of characters but also values of variables of other types for which the '<<' operator has been defined — these are in particular all built-in types, but also other types defined in standard C++ libraries. For example, we can do it with objects of class **string** from the standard library, as in the following program

P3: john.cpp '<<' operator

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. int main() {
6.     double weight = 76.5;
7.     int    height = 182;
8.     string name   = "John";
9.     cout << name   << " weighs "  << weight << " kg and is "
10.         << height << " cm tall." << endl;
11. }
```

which outputs

```
John weighs 76.5 kg and is 182 cm tall
```

The class **string** used in this example will be described in the chapter on strings in C++. We will come to other new elements appearing in this example very shortly.

2.2.2 Data input

To input data from the “outer world” to the program, we use a similar mechanism but now applied to the standard input stream represented by the object ***cin*** (of class **istream**). By default input data flows from the keyboard, which plays the rôle of the source of the stream (***cin*** stands for *console input*).

As with output streams, we will confine ourselves to simple **extraction** of data from the input stream. We can do it with the help of the **stream-extraction operator**, '>>' (two consecutive *greater than* symbols), as in

P4: read.cpp '>' operator

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  int main() {
6.      string name;
7.      int height;
8.      double weight;
9.      cout << "Enter your name, height and weight: ";
10.     cin  >> name >> height >> weight;
11.     cout << name << ", you are " << height  << " cm tall "
12.          << "and you weigh " << weight << " kg" << endl;
13. }
```

which gives, when run:

```
cpp> ./read
Enter your name, height and weight: Tom 182 76.5
Tom, you are 182 cm tall and you weigh 76.5 kg
cpp>
```

Let us note that:

- Direction of “arrows” shows the direction of information flow: for ***cout*** to the left, towards the ***cout***, i.e., the screen; for ***cin*** information flows “out of” ***cin***, i.e., from the keyboard.
- Each piece of information is extracted separately (you cannot use just commas to separate them).
- The type of data must be compatible with the declared type of variables in which the data is to be stored. In the example above, the program expects first a string (which is to be stored in a variable of the type very appropriately named **string**), then a whole number (this type is called **int**), and then a real number, possibly with non-vanishing fractional part (in C++ it is type **double**). Entering a real number — with a dot separating the integer and fractional parts — as the second data, when a whole number is expected, will cause an error which can escape unnoticed if we do not check it ourselves.
- Consecutive items are separated by an arbitrary long nonempty sequences of white spaces (blank – SP, tabulator – HT, new-line character – LF, carriage return – CR). White spaces before the first item are ignored; after that each such sequence is a separator. It follows that in this way it is not possible to enter strings with embedded spaces! Suppose we have a string variable ***nameSurname*** and we try to read in its value:
  `cin >> nameSurname;`
  Then, after entering '`John Doe`' on the keyboard, we would get '`John`' as the value of

nameSurname and '`Doe`' would remain in the keyboard buffer waiting for the next input operation (most probably provoking an error).

Input/output operation are quite complicated in every programming language. This is particularly true when reading data from the keyboard, when user's imagination in entering wrong and invalid data is virtually unlimited. We will postpone the details to the special chapter on input/output operations. In particular, in the section on input/output operations on files, we will show how to read from or write to text files.

Prev: 2.1 Simplest C++ programs **Up:** 2. Let's start, then... **Next:** 2.3 Functions

Subsections

- 2.2.1 Outputting information
- 2.2.2 Data input

2.2 Simplest In/Out operations

After we have included the header file `iostream`, we have at our disposal various objects defined there (in fact declared only, but that is not our concern now). Among them, there are two objects named ***cout*** and ***cin***. They represent what is called standard streams: **standard input stream**, a.k.a. *stdin* (***cin***) and **standard output stream**, a.k.a. *stdout* (***cout***).

2.2.1 Outputting information

The object ***cout*** represents standard output stream. It is a predefined object of the class **ostream**, whose definition is visible due to inclusion of `iostream`. It is responsible for outputting information from our program to the “outer world”. In this case, the rôle of the outer world is played by the screen of our computer — ***cout*** stands for *console output* (in Java a similar object is called ***System.out***).

As ***cout*** is an object, we can invoke various *methods* (functions) on it. We will learn more about them in chapter on input/output operations. For now it will suffice to know how we can use ***cout*** to be able to insert a piece of information — actually, a string of characters — into the stream of information flowing from the program to the computer's screen, or whatever the operating system considers to be the *sink* of the standard output stream at the moment. We can do it with the help of the **stream-insertion operator**, '<<' (two consecutive *less than* symbols). We can see an example in the line ❸ of the program `helloWorld.cpp`. We then write

```
cout << "Hello, World" << endl;
```

to insert a string of characters (given here literally by enclosing it in double quotation marks) to the output stream represented by ***cout***. After the next token '<<', we send to the stream something which is called ***endl***. This is not enclosed in quotation marks, so we know that it is not a string " ***endl***" that we want to appear on the screen. This is a so called *manipulator* which also became visible thank to inclusion of `iostream`. Inserting it into an output stream will add the end-of-line character (*line feed*), so the cursor on the screen will be set at the beginning of the next line. We did not have to add it but if we wouldn't have done it, the next output or input would start in the same line right after the displayed text `"Hello, World"`. We would get a similar effect by inserting the end-of-line character directly at the end of the string to be displayed: `"Hello, World\n"` The symbol '`\n`' (consisting of two characters: the backslash and the letter '`n`') within a literal string denotes the line feed character. Actually, the effect of ***endl*** and '`\n`' are similar but not identical: inserting ***endl*** to the output stream not only inserts line feed character but also *flushes* the

output buffer, so the result appears on the screen immediately (otherwise the system could wait for more data before displaying them).

In this simple example, we have put two objects into the stream: a string of characters and a manipulator. We could have inserted more. Note however that each piece of data must be inserted separately by using '<<' operator. It would be *illegal* to separate them by commas:

```
cout << "Hello, World" , endl; // NO!!!
```

The following three lines, however, would be legal

```
cout << "The first line,"        << endl
     << "the second,"            << endl
     << "and finally the third." << endl;
```

Let us notice that this is *one* statement, outputting three strings and three end-of-line characters.

Those who know Java will remember that in this language we can write

```
int k = 7;
double x = 8.6;
System.out.println(k);
System.out.println(x);
```

This snippet of code will write out numbers 7 and 8.6, although the type of argument is in both cases different and none of them is a reference to an object of class **String**. This is possible due to overloading of the **println** function. Similar mechanism works also in C++ for the '<<' operator. We can insert to the output stream not only strings of characters but also values of variables of other types for which the '<<' operator has been defined — these are in particular all built-in types, but also other types defined in standard C++ libraries. For example, we can do it with objects of class **string** from the standard library, as in the following program

P3: john.cpp '<<' operator

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. int main() {
6.     double weight = 76.5;
7.     int    height = 182;
8.     string name   = "John";
9.     cout << name   << " weighs "  << weight << " kg and is "
10.         << height << " cm tall." << endl;
11. }
```

which outputs

```
John weighs 76.5 kg and is 182 cm tall
```

The class **string** used in this example will be described in the chapter on strings in C++. We will come to other new elements appearing in this example very shortly.

2.2.2 Data input

To input data from the “outer world” to the program, we use a similar mechanism but now applied to the standard input stream represented by the object ***cin*** (of class **istream**). By default input data flows from the keyboard, which plays the rôle of the source of the stream (***cin*** stands for *console input*).

As with output streams, we will confine ourselves to simple **extraction** of data from the input stream. We can do it with the help of the **stream-extraction operator**, '>>' (two consecutive *greater than* symbols), as in

P4: read.cpp '>' operator

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  int main() {
6.      string name;
7.      int height;
8.      double weight;
9.      cout << "Enter your name, height and weight: ";
10.     cin  >> name >> height >> weight;
11.     cout << name << ", you are " << height  << " cm tall "
12.          << "and you weigh " << weight << " kg" << endl;
13. }
```

which gives, when run:

```
cpp> ./read
Enter your name, height and weight: Tom 182 76.5
Tom, you are 182 cm tall and you weigh 76.5 kg
cpp>
```

Let us note that:

- Direction of “arrows” shows the direction of information flow: for ***cout*** to the left, towards the ***cout***, i.e., the screen; for ***cin*** information flows “out of” ***cin***, i.e., from the keyboard.
- Each piece of information is extracted separately (you cannot use just commas to separate them).
- The type of data must be compatible with the declared type of variables in which the data is to be stored. In the example above, the program expects first a string (which is to be stored in a variable of the type very appropriately named **string**), then a whole number (this type is called **int**), and then a real number, possibly with non-vanishing fractional part (in C++ it is type **double**). Entering a real number — with a dot separating the integer and fractional parts — as the second data, when a whole number is expected, will cause an error which can escape unnoticed if we do not check it ourselves.
- Consecutive items are separated by an arbitrary long nonempty sequences of white spaces (blank – SP, tabulator – HT, new-line character – LF, carriage return – CR). White spaces before the first item are ignored; after that each such sequence is a separator. It follows that in this way it is not possible to enter strings with embedded spaces! Suppose we have a string variable ***nameSurname*** and we try to read in its value:
  `cin >> nameSurname;`
  Then, after entering '`John Doe`' on the keyboard, we would get '`John`' as the value of

nameSurname and '`Doe`' would remain in the keyboard buffer waiting for the next input operation (most probably provoking an error).

Input/output operation are quite complicated in every programming language. This is particularly true when reading data from the keyboard, when user's imagination in entering wrong and invalid data is virtually unlimited. We will postpone the details to the special chapter on input/output operations. In particular, in the section on input/output operations on files, we will show how to read from or write to text files.

Prev: 2.1 Simplest C++ programs **Up:** 2. Let's start, then... **Next:** 2.3 Functions

2.3 Functions

Functions are essentials constituents of all computer programs (not only those written in C/C++). We will consider them in detail in chapter on functions later on. Here, we will provide a short introduction just to be able to start using them in our examples.

The rôle of functions in programming is similar to their rôle in mathematics. A function's definition is a kind of a recipe which tells how to obtain a result given some input data. The definition itself does not cause any calculations to be performed. However, when a function is defined, we can use it by supplying input data — we then get a result obtained from this data according to the recipe which was coded in this function's definition. We can do it as many times as we wish getting different results from different data (the data passed to a function is called its argument). For example, the following definition: $f(a, b, x) = ax + b$ means that every time we use this function, we have to supply three numbers as arguments to get one number which is the value of the product of the first and third argument, incremented by the second. When we use the function (in programming we say that we *call*, or *invoke* it), we write, say, $f(3, 5, w)$. That means *apply the recipe substituting 3 for a, 5 for b, and the current value associated with the symbol w for x*. Note that names a, b and x, which were used in the definition, do not appear in the expression $f(3, 5, w)$. [This is a feature of C/C++; there are languages where there *is* a way to use formal names of function's parameters when calling it — this is very useful in such languages like Python, Ada or Fortran 90/95.]

How is this expressed in programming? Let us consider:

P5: func.cpp Function

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  double linear(double a, double b, double x) {   ❶
5.      return a*x + b;
6.  }
7.
8.  int main() {
9.      double c = 2, z = 3;
10.     double result = linear(c,5,z);              ❷
11.     cout << "The result = " << result << endl;
12. }
```

The definition of **linear** starting at line ❶ states that:

- we are defining a function named **linear**;
- this function will need three numbers of type **double** as input data (arguments). Type **double** is an approximation of real numbers known from mathematics; more about it in chapter on data types . In the definition, these arguments will be named ***a***, ***b*** and ***x***; as

in mathematics, the list of formal arguments of a function is written after its name and in parentheses;

- the result calculated by the function will be also of type **double** — this information is given *before* the function's name;
- the definition itself (the “recipe”) is enclosed in braces. In our case the definition consists of one line only, but of course it could have been more;
- the statement **return** means that the function completes its calculations here and its result is the value of an expression appearing after the keyword **return**.

On line ❷ of our program we *call* (invoke) the function **linear**. We pass (“send”), as arguments, the value of variable ***c*** (which is 2) as the first argument, value 5 as the second, and the value of variable ***z*** (which is 3) as the third. These three values will be substituted for ***a***, ***b*** and ***x*** during the execution of the function; the value returned (in our case 11) will be substituted for the the expression '`linear(c,5,z)`' and its value assigned to the variable ***result***, which is then printed.

Prev: 2.2 Simplest In/Out operations **Up:** 2. Let's start, then... **Next:** 2.4 Command-line arguments

2.4 Command-line arguments

Programs can receive input data in the form of command-line arguments. We will show here how it can be done postponing detailed explanation until we know more about arrays, pointers and C-strings.

If we want to use command-line arguments in a program, we have to change the header line of the **main** function. Now, it has to be defined with parameters:

```
int main(int argc, char **argv)
```

or, equivalently,

```
int main(int argc, char *argv[])
```

The first parameter, conventionally called ***argc***, will be assigned an integer value indicating the number of command-line arguments. This will always be at least one, as the name of the program itself (which must have been given to invoke it) is the first argument passed to the program. Variable ***argv*** looks strange: it is an array (table): a collection of strings containing command-line arguments. The numbering starts from 0 and is indicated by an integer in square brackets (*index*): ***argv[0]***, ***argv[1]***, ..., ***argv[argc-1]***. Note, that the last argument has index ***[argc-1]*** and not ***[argc]***, as the numbering starts from zero! [This is an equivalent of the array usually called ***args*** in Java.] Since in C++, as we will see, arrays “don't know their sizes”, we had to specify the size of the array separately, as value of ***argc*** (which stands for *argument count*). More details on arrays will be given in chapter on strings as arrays of characters.

A program outputting its command-line arguments could look like this:

P6: arguments.cpp Command-line arguments

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(int argc, char **argv) {
5.     cout << "Program name "         << argv[0]  << endl
6.          << "Number of arguments: " << argc     << endl;
7.     for (int i = 1; i < argc; i++)
8.         cout << "Argument nr " << i       << " is "
9.                               << argv[i] << endl;
10. }
```

and could be used as follows:

```
cpp> g++ -o arguments arguments.cpp
cpp> ./arguments 12 a "b c" 'd e'
Program name ./arguments
Number of arguments: 5
Argument nr 1 is 12
Argument nr 2 is a
```

```
Argument nr 3 is b c
Argument nr 4 is d e
```

As we see, it *is* possible to pass as arguments strings with embedded spaces, if only we enclose them in apostrophes or double quotation marks. In the example above we have 5 argument: the first one, with index 0, is the name of the program (**arguments**).

Prev: 2.3 Functions **Up:** 2. Let's start, then... **Next:** 3. Preprocesor

3. Preprocesor

When talking about the `#include` directive, we mentioned briefly the preprocessor. This is a special program which processes the *text* of C/C++ program before its compilation proper. We will describe its use in this chapter in more detail.

Generally, in C++ programs, only the most important directives should be used (it would be quite hard not to use `#include`...). Other directives, used very intensively in pure C code, can (and should) be replaced by appropriate constructs from the C++ language itself.

Subsections

- 3.1 What is the preprocessor?
- 3.2 Preprocessor directives
- 3.3 Predefined preprocessor macros

T.R. Werner, September 27, 2023; 22:32

3.1 What is the preprocessor?

Preprocessor is a program that reads our source file with text of a program looking for directives which are intended for it. These directives are then applied to the *text* being processed — the result is another text: a modified version of the original source file. This resulting text is then sent for further processing to the compiler. Preprocessor directives themselves are removed by the preprocessor and will not be seen by the compiler.

The preprocessor does not perform any check of syntactic correctness of the program; in fact, it knows nothing about C/C++ syntax! In particular

preprocessor directives are not C/C++ statements and therefore do *not* end with a semicolon.

One of the drawbacks of using preprocessor directives is the fact that the text seen by the compiler may be effectively different from the original source text of our program. This can lead to confusing and hard to understand errors or warning messages issued by the compiler, because it “sees” something different than we do in our source file. Therefore, we should exercise caution when using preprocessor directives — as we already stated, there are often better alternatives that can be expressed directly in the C++ language.

For this reason, we will describe only those directives which are absolutely necessary or most convenient: the rest can be found in literature.

All preprocessor directives start with character hash character, '#', as the first non-blank character of a line. If a directive does not fit into one line, we have to add at its end a special mark for the preprocessor to know that the next line is a continuation of this directive. The continuation mark is a backslash, '\', which must be the last character of the line. For example,

```
#define size 256
```

is equivalent to

```
#define size \
         256
```

Let us then look at the most important preprocessor directives.

3.2 Preprocessor directives

`#include <file>` `#include "file"`

includes the contents of a file **`file`** which replaces the line with the directive. White spaces between angle brackets (*less than* and *greater than* marks) are not allowed. White spaces between `#include` and opening angle bracket are optional.

In the first of these forms, with angle brackets, the file **`file`** will be searched for in a "well known" directory, specific for a given compiler. The name of such catalogs is very often **`include`**, as for example '**`/usr/include`**'.

In the second form, with quotation marks, the file is searched for in the current catalog and, if not found, in the "well known" directory as if angle brackets were used. Usually there are compilation options which allow the user to include other catalogs to the list of catalogs which are searched when trying to resolve '`#include <file>`' directives.

Files included with `#include` directive are usually normal C/C++ source files, which, in particular, can contain other `#include` directives (i.e., `#include`'s *can* be nested). Some compilers use some form of precompiled version of included files; if this is the case, it should be transparent to the user.

`#define name value` `#define name` `#undefine name`

The first of these form replaces every occurence of a lexem `name` with `value`. This means that the lexem `name` will not appear in the resulting text at all. There must be no space in `name`: anything after `name` (even spaces and quotes) to the end of the line is considered to be `value`. We say *lexem* to emphasize that the replacement will take place only if `name` constitutes a separate lexeme (identifier, keyword, literal value...), but not if it is just a substring of a longer lexeme. Thus, the following lines

```
#define dim 256

int k = dim;
int dimen = 2*dim;
```

are equivalent to

```
int k = 256;
int dimen = 2*256;
```

In the second line, fortunately, we will not get '`int 256en = 2*256`', as '`dim`' in '`dimen`' is not a separate lexem, but rather a substring of the lexem `dimen`.

If a `name` appears as the first argument of the `#define` directive, this `name` becomes defined even if no `value` has been specified. In this case `name` will not be replaced by anything, but nevertheless will be deemed to be defined (what can be then tested; we will see examples shortly).

A common programming error is to write something like '`#define dim=256`', instead of '`#define dim 256`'. After the first of this form, every occurence of the lexem `dim` would

be replaced not by '256', but rather by ' `=256`' (with the equal sign). Thus, the instruction ' `k=m=dim`' would become ' `k=m==256`' what formally may make sense, but means something completely different (errors of this type belong to the hardest to detect).

A name `name` which was defined by `#define` directive can be undefined (erased from the list of defined names) by the directive ' `#undefine name`'.

In the following snippet, function **function** will be compiled with all occurrences of **int** declarations replaced with **double**. After that, a preprocessor name ' `int`' will be undefined and normal meaning of **int** will be restored:

```
...
#define int double
int function(int k, int m) {
    int x,y,z;
    ...
    ...
}
#undefine int
...
```

The `#define` directive is often used to define a constant which then appears in declarations of arrays as their dimension (see the chapter on arrays). This practice is not recommended: it is much better to use constants defined directly in the code, as we will explain in the chapter on variables .

`defined` `!defined`

The preprocessor directive ' `defined name`' is basically a function which tests if `name` is, or is not currently defined (by previous ' `define name`' directive). If it is defined, the function returns 1 (interpreted as **true**); if not, 0 is returned (interpreted as **false**). The value returned can be then used in conditional directives (see below). With the exclamation mark, the meaning is reversed: 1 (**true**) will be returned if the name is not defined, and 0 (**false**) otherwise. The same directive can be written with parentheses, explicitly as a function: `defined(name)` or `!defined(name)`. This function can appear only after `#if`, `#elif` and as a subexpression in more complicated logical expressions (see below).

`#if` `#ifdef` `#ifndef` `#else` `#elif` `#endif`

Withe the help of these directives, we can manage conditional compilation, that is including or excluding (or modifying) certain portions of the program's text before sending it to compilation. The meaning of `#if`, `#else`, `#endif` is obvious and intuitive (`#elif` stands for **else if**). Typical usage could look like:

```
1.  #define dimen
2.  ...
3.  ...
4.  #if defined dimen
5.      // This will be compiled
6.      // if 'dimen'  i s  defined
7.  #else
8.      // This will be compiled if
9.      // 'dimen' is  n o t  defined
10. #endif
```

The expression in the fourth line could have been replaced with ' `#ifdef dimen`', i.e., `#ifdef` is an abbreviated form of ' `#if defined`'. Similarly, `#ifndef` (which stands for *if not defined*) is an abbreviated form of ' `#if !defined`'.

Let's consider an example. Suppose our program will be sometimes compiled with a C compiler and sometimes as a C++ program. If it is a C++ compiler, we would like to use '<<'

for outputting data. There is no such operator in C, so we will use standard C function **printf** (line ❶ of the program below). Any C++ compiler defines a name (macro) `__cplusplus` (two underscores at the beginning), while C compilers don't. We can solve our problem like this:

P7: cvscpp.c Conditional compilation

```
1.  #ifdef __cplusplus
2.      #include <iostream>
3.      using namespace std;
4.  #else
5.      #include <stdio.h>
6.  #endif
7.
8.  int main() {
9.  #ifdef __cplusplus
10.     cout << "Hello, C++" << endl;
11. #else
12.     printf("Hello, C\n");          ❶
13. #endif
14. }
```

To use C++, we invoke (under Linux) the **g++** compiler; to use C, we invoke **gcc** (the extension of the source file's name should be `.c`). We get:

```
cpp> g++ -o cvscpp cvscpp.c
cpp> ./cvscpp
Hello, C++
cpp> gcc -o cvscpp cvscpp.c
cpp> ./cvscpp
Hello, C
cpp>
```

In the same way we can use the fact that Visual Studio compiler always defines macro `_WIN32` (one underscore at the beginning), even on 64-bit machines, while `gcc` compilers define `__linux__` (double underscore at both sides). This allows us to compile the same source file on these two platforms.

We can use a similar technique to avoid including same file more than once. Such a possibility is quite real, as `#include` directive can be nested, and it is quite likely that one of the included header files will in turn try to include another, which has already been included. Suppose the compiler (preprocessor) tries to include several times a header file `head.h` (encountering '`#include "head.h"`'), but in this file we use the following construction (called **include guard**)

```
#ifndef HEAD_H
#define HEAD_H

   // code here

#endif
```

Notice that the file will actually be included once. At the beginning the name (macro) `HEAD_H` *is not* defined so '`#ifndef HEAD_H`' is true. Thus, everything between '`#ifndef HEAD_H`' and `#endif` will be passed for further processing. The first thing the preprocessor will do now is defining `HEAD_H`! Next time it tries to include the header file `head.h`, the name `HEAD_H` will already be defined, so '`#ifndef HEAD_H`' will be false and everything through `#endif` (i.e., to the end of the file `head.h`) will be skipped. Of

course, we have to be very careful not to use the same macro name more than once!

Most compilers, although it is not required by the standard, will take care about the problem if a file starts with the line `#pragma once`.

We can built more complicated logical expressions after `#if` or `#elif` using the operators of logical alternative (`||`), logical conjunction (`&&`) and logical negation (`!`), exactly as we do it in C/C++ (see chapter on operators).

`#error any message`

When preprocessor encounters this directive, it displays an error message containing the string given after the directive itself. Further processing is sometimes abandoned, although not all compiler behave that way.

Suppose, for example, we are trying to compile the following program:

P8: preplog.cpp #error directive

```
1. #if   defined(POL) && defined(FRA)
2.    #error Please define only one country
3. #elif !(defined(POL) || defined(FRA))
4.    #error Please define a country
5. #endif
6.
7. #ifdef POL
8.    #define country "Poland"
9.    #define capital "Warsaw"
10. #elif defined(FRA)
11.    #define country "France"
12.    #define capital "Paris"
13. #endif
14.
15. #include <iostream>
16. using namespace std;
17.
18. int main() {
19.     cout << capital << " is the capital of "
20.          << country << "." << endl;
21.     return 0;
22. }
```

Normally, we would get an an error on the third line, as neither the name ***POL*** nor ***FRA*** is defined. However, we can define one or both of them directly from the command line using the option `-Dname`, which defines `name` without assigning any value to it (we could also use `-Dname=somethig`, which would additionally assign the value `somethig` to the macro `name` — note the presence of equal sign here). The effect of this can be seen from the following session trace:

```
cpp> g++ -o preplog preplog.cpp
preplog.cpp:4:5: #error Please define a country
cpp> g++ -o preplog -DPOL -DFRA preplog.cpp
preplog.cpp:2:5: #error Please define only one country
cpp> g++ -o preplog -DPOL preplog.cpp
cpp> ./preplog
Warsaw is the capital of Poland.
cpp> g++ -o preplog -DFRA preplog.cpp
cpp> ./preplog
Paris is the capital of France.
cpp>
```

3.3 Predefined preprocessor macros

We mentioned the symbol `__cplusplus` that is always defined by C++ preprocessors (while standard C compiler defines `__STDC__`). However, there are other macros which should be defined by any C/C++ preprocessor. They usually begin and end with double underscore and all are assigned some values which can be used within the program:

`__LINE__`
: current line number as an integer value;
`__FILE__`
: name of the source file currently processed as a string;
`__DATE__`
: a string containing the current date;
`__TIME__`
: a string containing the current time;
`__FUNCTION__`
: name of function within which the macro was used.

The last macro (`__FUNCTION__`) is formally nonstandard, but usually implemented. We can see these macros in action in the following little program

P9: datetime.cpp Predefined preprocessor macros

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     cout << "File:     " << __FILE__     << endl
6.          << "Date:     " << __DATE__     << endl
7.          << "Line:     " << __LINE__     << endl
8.          << "Time:     " << __TIME__     << endl
9.          << "Function: " << __FUNCTION__ << endl;
10. }
```

which produces

```
cpp> gpp datetime.cpp
cpp> ./a.out
File:     datetime.cpp
Date:     Jul 10 2017
Line:     7
Time:     23:02:15
Function: main
```

4. Basic data types

In this chapter we will discuss types of data that can be manipulated by C/C++ programs. This is an essential issue in programming languages, as all data (numbers, character string, etc) must be somehow represented in the memory of the computer and it must be known what operation can be performed on this data and what will be the result of these operations. As we will learn later, we can define our own types based on those that are already defined. In this chapter we confine ourselves to basic (built-in) types which must be defined in all C/C++ implementations. For the novices, the most difficult to understand are pointer types and, in C++ only, reference type. Pointer types are intimately connected with arrays: this connection will be the subject of the next chapter.

Subsections

- 4.1 Introductory remarks
- 4.2 Integral types
  - 4.2.1 Useful aliases for integral types
- 4.3 Floating-point types
- 4.4 Logical type
- 4.5 Enumeration types
- 4.6 Pointers
  - 4.6.1 Pointers to variables
  - 4.6.2 Generic pointers
- 4.7 References

Prev: 3.3 Predefined preprocessor macros **Up:** Programming in C++ **Next:** 4.1 Introductory remarks

4.1 Introductory remarks

Both C and (even more so) C++ are examples of programming languages with **strong typing** (as many other languages, e.g., Java or Pascal). That means, loosely speaking, that all “pieces of data” (variables) have to have their type established before they can be used. To be able to refer to these “pieces of data”, we have to give them names. Such names are also called **identifiers**.

There are some rules specifying legal names (identifiers) of variables (and other objects which we refer to in our program, like functions or classes). Names can contain only letters, digits and underscore (no currency symbols, as in some other languages). The first character must not be a digit. A name must not have the same spelling as a reserved word (a keyword).

Upper- and lower-case letters are considered *different*.

Therefore, two names (identifiers) ***A_book*** and ***a_book*** will be treated as names of two completely different entities in the program.

The “pieces of data” which are stored in the computer memory and to which we can refer (usually by their names) are **variables**. Every variable must have a type. a **type** is defined by a set of values and a set of operations on these values. For example, a variable of type **unsigned char** can assume exactly 256 different values (these are integer numbers from 0 to 255). Compiler uses the information about the type of a variable to allocate an appropriate amount of memory for storing its value and to know how to interpret various operations on this variable. Thus, before we can use a variable in our program, we have to inform the compiler about its type. We do it by *declaring* the variable as being of a specified type. Declaration of a variable is usually connected with its **definition**, i.e., reserving memory in which it will be stored.

Built-in types are similar to those known perhaps to the reader from other languages. Unlike it is Java, however, sizes of variables (amount of memory needed to store them) are not guaranteed by the standard. For example, any variable of type **int** must occupy exactly 4 bytes in Java; in C/C++ it can be 2, 4 or 8, depending on implementation (usually it is 4). Therefore, it would be very useful to have some means allowing us to determine the size of a variable from within our program. Such functionality is provided by the built-in operator **sizeof**, which returns the length (in bytes) of the binary representation of a single variable or of variables of a given type. We can use this operator as function, specifying, as the argument, a name of a variable (in this case parentheses are optional) or the name of a type (in parentheses). The name of a type is enough, as

all variables of a given type have the same size.

As we have already said, all variables — named pieces of information, with a well defined address in memory and type — must be declared and defined before they can be used. Let

us see how we can to declare, as an example, variables of type **int**:

P10: vardecl.cpp Defining variables

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int k1;
6.     int k2(1);
7.     int k3{};
8.     int k4{1};
9.     int n=1, m = n, i{1}, j{i};
10. }
```

As we can see, we specify the type of the variable first, and then the name of this variable. We do not have, but rather we should, set sensible values of the newly introduced variables. We do it by **initializing** them right when they are defined. For example, in the program above:

- ***k1*** is defined, but not initialized; its value is undefined and can be anything;
- ***k2*** is defined and initialized with 1;
- ***k3*** and ***k4*** are defined and initialized, but using new syntax, indroduced in the C++11 standard. This is an example of the called **uniform initializer**: with curly braces (called also **brace-init**).

The last line shows that many declarations/definitions of variables of the same type can be combined into one statement; this notation is equivalent to the series of definitions:

```
int n = 1;
int m = n;
int i{1};
int j{i};
```

which shows, in particular, that defining, e.g., ***m*** we can treat ***n*** as already existing. The first two lines above show also the “classic” form of initialization with equal sign (normally, the equal sign denotes assignment, but if the object on the left hand side does not exist and is just being created, it is *not* an assignment, but initialization).

Let us see an example showing definitions of variables and the **sizeof** operator in action:

P11: lengths.cpp Sizes of data of various types

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. int main() {
6.     long double ld = 0;
7.     string      st = "Hermenegilda";
8.     short       sh = 0;
9.     long       *lo = nullptr;
10.    cout << "long double: " << sizeof ld          << endl
11.         << "double     : " << sizeof(double)     << endl
12.         << "float      : " << sizeof(float)      << endl
13.         << "long long  : " << sizeof(long long)  << endl
14.         << "long       : " << sizeof(long)       << endl
15.         << "int        : " << sizeof(int)        << endl
```

```
        << "short      : " << sizeof sh         << endl
        << "char       : " << sizeof(char)      << endl
        << "bool       : " << sizeof(bool)      << endl
        << "string     : " << sizeof st         << endl
        << "long*      : " << sizeof lo         << endl;
}
```

produced this output on a linux 64-bit machine

```
long double: 16
double     : 8
float      : 4
long long  : 8
long       : 8
int        : 4
short      : 2
char       : 1
wchar_t    : 4
char16_t   : 2
char32_t   : 4
bool       : 1
string     : 32
long*      : 8
```

As one can see, in this system **long** has the same representation as **long long**; on 32-bit machines **long** has usually the same size as **int** (but still is considered to be a distinct type). Note also that the size of objects of type **std::string** depends on its implementation and varies between compilers.

As we can see, declaration has the following form:

```
Type variable;
```

but can be also expressed as

```
auto variable = value;
```

or

```
decltype(expression) variable;
```

The keyword **auto** means here, that the compiler is supposed to guess the type of the declared variable based on the value used to initialize it (of course this type will be strictly defined and cannot be changed). The keyword **decltype** means, that variable ***variable*** is to have the same type as is the type of the expression inside parentheses. The expression itself will *not* be evaluated, compiler will only check of what type the result would be. An example should clarify this issue:

P12: autodecl.cpp Declarations with **auto** and **decltype**.

```
#include <iostream>
using namespace std;

int main() {
    auto k = 7;             // k has type int
    auto x = 1.;            // x has type double
    decltype(x) y = 7;      // y has type double, although
                            // '7' is a literal of type int
    decltype(k*x) z = 7;    // product k*x has type double
    cout << "k/2="   << k/2 << ", y/2=" << y/2
         << ", z/2=" << z/2 << endl;
}
```

This little program prints

```
k/2=3, y/2=3.5, z/2=3.5
```

which shows that indeed ***y*** and ***z*** have type **double** — if they were **int** s, the division by 2 would have given exactly 3 (as is the case for ***k***). There are some subtleties associated with **auto** and **decltype** that we will cover later. At the moment the usefulness of these constructs may seem doubtful; as we will see, however, they are in fact *extremely* useful!

Prev: 4. Basic data types **Up:** 4. Basic data types **Next:** 4.2 Integral types

Subsections

- 4.2.1 Useful aliases for integral types

4.2 Integral types

Below we list standard C/C++ integral types; in parentheses, typical sizes of variables of a given type is indicated. Typical, because the C++ standard (unlike Java standard) establishes only minimum sizes and states that sizes of variables of types mentioned must form a nondecreasing sequence. So integer types are: **char** (1), **short** (2), **int** (4) **long** (4 lub 8), i **long long** (8). Type **short** can also be spelled as **short int**, **long** as **long int**, and **long long** as **long long int**. The new standard introduced also three other *character* types: **wchar_t** (4), **char16_t** (2), and **char32_t** (4); the are considered to be different (unsigned) types which are to be used to represent Unicode characters.

All these integer types (except the last three character types) come in two flavors: without sign (**unsigned**), or with sign (**signed**). The name of an unsigned type is the same as the name of the corresponding signed type, but preceded by the keyword **unsigned** (e.g., **unsigned int**) — as a matter of fact, names of signed types can also be explicitly preceded by **signed**, although it not necessary. The keyword **unsigned** alone can be used instead of more verbose **unsigned int**.

With the type **char** the situation is more complicated. Type **char** is in fact physically equivalent to either **signed char** or **unsigned char** — this depends on implementation. However, all three types, **signed char**, **unsigned char** and **char** are considered different. If we want to be sure what the signedness of a character variable is, we should explicitly use **unsigned char** or **signed char**.

In order to make internationalization of applications possible, there are also three additional character types. They are by default *unsigned* — their **signed** versions do not exist. These are:
wchar_t (*wide character*) — large enough to hold any character in the machine's largest character set;
char16_t — two bytes interpreted as Unicode 16 code;
char32_t — four bytes interpreted as Unicode 32 code.

Let us say a few words about the difference between signed and unsigned types.

In an *unsigned* variable, all bits of its binary representation are interpreted as zero's and one's of the representation of this number in the standard base 2 notation.

For example, let us consider an **unsigned char** variable (one byte long by definition in C/C++). There are 8 bits in a byte and they are interpreted as coefficients (0 or 1) at consecutive powers of 2: from the 0-th power (conventionally this bit is depicted as the

rightmost one) to the 7-th power (leftmost bit). The bit pattern in the figure, interpreted as **unsigned char**, yields the value W_{uns}:

$$W_{\text{uns}} = w_0 \cdot 2^0 + w_1 \cdot 2^1 + w_2 \cdot 2^2 + w_3 \cdot 2^3 + w_4 \cdot 2^4 + w_5 \cdot 2^5 + w_6 \cdot 2^6 + w_7 \cdot 2^7$$

where, from right to left, $w_{\text{i}} = (1, 1, 0, 0, 1, 0, 0, 1)$. Therefore,

$$W_{\text{uns}} = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 = 147$$

There is no way to represent a negative value in a variable of an **unsigned** type.

Now, let us interpret the same bit pattern as a **signed char**. In the so called *2's complement* arithmetic, used on most platforms, the term with the highest power of 2 (corresponding to the leftmost bit in the conventional notation) is taken with minus sign. In our case it is the 8-th bit — the leftmost one. Therefore, we have now

$$W_{\text{sgn}} = \sum_{i=0}^{6} w_i \cdot 2^i - w_7 \cdot 2^7$$

and for our particular example

$$W_{\text{sgn}} = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 - 1 \cdot 2^7 = -109$$

so the same bit pattern is now interpreted as a negative number.

From the above considerations it should be clear that an **unsigned char** variable will assume the largest possible value if $w_{\text{i}} = 1$ for $i = 0,\ldots, 7$, which gives $1 + 2 + \cdots + 64 + 128 = 2^8 - 1 = 255$. The smallest possible value will be, of course, zero.

We can make a **signed char** variable as large as possible by setting $w_{\text{i}} = 1$ for $i = 0,\ldots, 6$ and $w_7 = 0$, since in this way we include all positive terms but exclude the negative one. This leaves us with the number represented by 01111111 which amounts to $1 + 2 + \cdots + 32 + 64 = 2^7 - 1 = 127$ We will get the smallest possible value if we include the negative term ($w_7 = 1$), and exclude all positive terms ($w_{\text{i}} = 0$ for $i = 0,\ldots, 6$) getting 10000000 which is -128. It is easy to see that -1 is represented by 11111111.

Generally, for numbers represented on n bits, we will get for **unsigned** types values in the range $[0, 2^{\text{n}} - 1]$, while for **signed** types this range will be $[-2^{\text{n-1}}, 2^{\text{n-1}} - 1]$. Specifically, for typical sizes of integer variables that are used in computer programs we get:

Bytes	Sign	Min	Max
1	signed	-128	127
1	unsigned	0	255
2	signed	-32 768	32 767
2	unsigned	0	65 535
4	signed	-2 147 483 648	2 147 483 647
4	unsigned	0	4 294 967 295
8	signed	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
8	unsigned	0	18 446 744 073 709 551 615

Table 4.1: integer types in C/C++

The type **char** (as the other character types) is an integer type, but is treated in many

contexts in a special way, as it is intended to represent characters. Therefore, we should never use variables of character types in arithmetic calculations, although formally it is perfectly possible. Values of character variables usually (but not necessarily) correspond to ASCII codes of characters. Only the first 128 values (from 0 to 127) are assigned a standard meaning in ASCII — others can be interpreted differently on different platforms in a locale dependent way. As a matter of fact, C/C++ standard does not even guarantee conformance with ASCII codes; in most implementations, however, numerical values corresponding to basic characters *do* conform to ASCII standard.

Let us consider the following example which illustrates declarations of variables of different types:

P13: decl1.cpp integer types

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      unsigned long int ul1 = 13UL;
6.      unsigned long     ul2 = 0xD;  // 13 in hex
7.      signed   short    ss1 = 015;  // 13 in oct
8.      short             ss2 = 13;   // 13 in dec
9.      unsigned char     aa1 = 65;
10.     signed   char     aa2 = 'A';  // ASCII('A') = 65
11.     int               aa3 = 65;
12.     int               aa4 = 'A';
13.     char              aa5 = '\101';   // 65 in oct
14.     char              aa6 = '\x41';   // 65 in hex
15.     cout << aa1 << " " << aa2 << endl
16.          << aa3 << " " << aa4 << endl
17.          << aa5 << " " << aa6 << endl;
18. }
```

The program prints

```
a A
65 65
a A
```

All variables ***aa1***... ***aa4*** have the same numerical value: 65 (which is the ASCII code of the letter '`A`'). If we print this values as the value of a variable of type **char**, we will see '`A`', as in this case the compiler knows that the value should be treated as the ASCII code of a letter. The same value but in a variable of other integer type is treated as itself, namely as the number 65. This is so even if a character literal was used to initialize the variable: what counts is the declared type of this variable and not the way it has been initialized (assigned a value) — see the declaration of ***aa4***.

Sometimes we need to specify the type of a literal constant. If a literal can be interpreted as being of integer type, it will be treated as **int** (equivalent to **signed int**). To indicate that this should be considered to be a **long**, we add a letter '`L`' (lower or upper case) or '`LL`' for **long long**; similarly, if we want to enforce unsigned type, we add the letter '`U`' (lower or upper case). These modifiers can be applied simultaneously in any order. Therefore, '`13UL`' is the literal of 13 as **unsigned long** (see variable ***ul1***), and '`1LL`' is 1 of type **long long**.

Numerical integer literals can be expressed in decimal notation, but also in octal and hexadecimal form. Integer value with leading 0 (zero) are treated as octal values; e.g., 037 will be interpreted as decimal $3 \cdot 8 + 7 = 31$ and 015 as $8 + 5 = 13$ (see ***ss1***). If we use

octal notation, we have to remember that only digits from 0 to 7 can be used. In particular, '\0' is used to express the character with ASCII code zero (this is called ***NUL*** character and plays an important rôle in the so called C-strings).

We use '0x' (that is zero-ex; 'x' can be in lower or upper case) for numbers in hexadecimal notation. Therefore, 0x2D amounts to decimal value $2 \cdot 16 + 13 = 45$, while 0xD denotes decimal 13 (see ***ul2***). In hexadecimal notation, one uses the letters A to F (lower or upper case) as digits from 10 to 15.

Character literals are denoted by a character enclosed in apostrophes (*not* double quotes!) — as on line defining ***aa2***. This can be problematic for characters which cannot be entered from the keyboard. In such situations, one can use the form '\ooo' (including apostrophes), where ooo denotes at most three-digit number in octal notation (leading zero can be omitted in this case). For an example, see definition of ***aa5***. In the next line, we used another notation: '\Xhh'. After the letter 'X' (or 'x') one can put at most two-digit number in hexadecimal notation denoted here as 'hh'. There is no way to use this form in decimal notation: only octal or hexadecimal are permitted.

Some characters have special meaning and can be expressed in yet another notation: a backslash followed by one letter:

\n	newline (LF)	\t	horizontal tab (HT)
\v	vertical tab (VT)	\b	backspace (BS)
\r	carriage return (CR)	\f	form feed (FF)
\a	bell (alert) (BEL)	\\	backslash
\?	question mark	\'	single quote
\"	double quote		

Table 4.2: Escaped characters

For example

```
cout << "\101nn\x61 Anna\n\x4Ao\145 Joe" << endl;
```

will print

```
Anna Anna
Joe Joe
```

as, e.g., ASCII code of upper-case 'A' is 65_{10}=101_8=41_{16}. Note that '\n' within the text enforces new line.

4.2.1 Useful aliases for integral types

As we have said, sizes of data of various integral types are not fixed by the standard (as they are in Java). Sometimes we want to be sure, that a declared variable has a given size. For example, if our variable should be signed and capable of holding values larger than 33 thousand, it should be at least 4-bytes long. On a platform where **int** is 2-byte long (what is rare but possible), we cannot use **int**, but **long** (which must be at least 4-byte long). In such situation, it is possible to avoid specifying a type explicitly: we can use aliases (typedefs), defined in header **cstdint**, of existing types which have the required size (**typedef** will be covered in section about typedefs later). For example, **int32_t** is an alias for an integral signed type which, on a given platform, is exactly 32-bit (4-byte) long — on most platform, it is just an alias of **int**. The number in the name of such aliases indicates the number of bits

(not bytes). Therefore, we can use **int8_t**, **int16_t**, **int32_t**, **int64_t** for *signed* integral types of the specified sizes. In a similar way, by prepending 'u' to the name of an alias, we get aliases of *unsigned* integral types: **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t**. All these names are really typedefs defined in namespace `std`, so their full names are **std::int32_t** etc.

The header `cstdint` defines also *preprocessor macros* which expand to minimum and maximum *values* of these types: **INT32_MIN**, **INT32_MAX**, **UINT32_MAX**, ect. Macros corresponding to minimum values are not defined for unsigned types, as for them the minimum is always 0. Since these are macros, their names cannot be prepended by `std::`.

Prev: 4.1 Introductory remarks **Up:** 4. Basic data types **Next:** 4.3 Floating-point types

Subsections

- 4.2.1 Useful aliases for integral types

4.2 Integral types

Below we list standard C/C++ integral types; in parentheses, typical sizes of variables of a given type is indicated. Typical, because the C++ standard (unlike Java standard) establishes only minimum sizes and states that sizes of variables of types mentioned must form a nondecreasing sequence. So integer types are: **char** (1), **short** (2), **int** (4) **long** (4 lub 8), i **long long** (8). Type **short** can also be spelled as **short int**, **long** as **long int**, and **long long** as **long long int**. The new standard introduced also three other *character* types: **wchar_t** (4), **char16_t** (2), and **char32_t** (4); the are considered to be different (unsigned) types which are to be used to represent Unicode characters.

All these integer types (except the last three character types) come in two flavors: without sign (**unsigned**), or with sign (**signed**). The name of an unsigned type is the same as the name of the corresponding signed type, but preceded by the keyword **unsigned** (e.g., **unsigned int**) — as a matter of fact, names of signed types can also be explicitly preceded by **signed**, although it not necessary. The keyword **unsigned** alone can be used instead of more verbose **unsigned int**.

With the type **char** the situation is more complicated. Type **char** is in fact physically equivalent to either **signed char** or **unsigned char** — this depends on implementation. However, all three types, **signed char**, **unsigned char** and **char** are considered different. If we want to be sure what the signedness of a character variable is, we should explicitly use **unsigned char** or **signed char**.

In order to make internationalization of applications possible, there are also three additional character types. They are by default *unsigned* — their **signed** versions do not exist. These are:
wchar_t (*wide character*) — large enough to hold any character in the machine's largest character set;
char16_t — two bytes interpreted as Unicode 16 code;
char32_t — four bytes interpreted as Unicode 32 code.

Let us say a few words about the difference between signed and unsigned types.

In an *unsigned* variable, all bits of its binary representation are interpreted as zero's and one's of the representation of this number in the standard base 2 notation.

For example, let us consider an **unsigned char** variable (one byte long by definition in C/C++). There are 8 bits in a byte and they are interpreted as coefficients (0 or 1) at consecutive powers of 2: from the 0-th power (conventionally this bit is depicted as the

rightmost one) to the 7-th power (leftmost bit). The bit pattern in the figure, interpreted as **unsigned char**, yields the value W_{uns}:

$$W_{\text{uns}} = w_0 \cdot 2^0 + w_1 \cdot 2^1 + w_2 \cdot 2^2 + w_3 \cdot 2^3 + w_4 \cdot 2^4 + w_5 \cdot 2^5 + w_6 \cdot 2^6 + w_7 \cdot 2^7$$

where, from right to left, $w_{\text{i}} = (1, 1, 0, 0, 1, 0, 0, 1)$. Therefore,

$$W_{\text{uns}} = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 = 147$$

There is no way to represent a negative value in a variable of an **unsigned** type.

Now, let us interpret the same bit pattern as a **signed char**. In the so called *2's complement* arithmetic, used on most platforms, the term with the highest power of 2 (corresponding to the leftmost bit in the conventional notation) is taken with minus sign. In our case it is the 8-th bit — the leftmost one. Therefore, we have now

$$W_{\text{sgn}} = \sum_{i=0}^{6} w_i \cdot 2^i - w_7 \cdot 2^7$$

and for our particular example

$$W_{\text{sgn}} = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 - 1 \cdot 2^7 = -109$$

so the same bit pattern is now interpreted as a negative number.

From the above considerations it should be clear that an **unsigned char** variable will assume the largest possible value if $w_{\text{i}} = 1$ for $i = 0,\ldots, 7$, which gives $1 + 2 + \cdots + 64 + 128 = 2^8 - 1 = 255$. The smallest possible value will be, of course, zero.

We can make a **signed char** variable as large as possible by setting $w_{\text{i}} = 1$ for $i = 0,\ldots, 6$ and $w_7 = 0$, since in this way we include all positive terms but exclude the negative one. This leaves us with the number represented by 01111111 which amounts to $1 + 2 + \cdots + 32 + 64 = 2^7 - 1 = 127$ We will get the smallest possible value if we include the negative term ($w_7 = 1$), and exclude all positive terms ($w_{\text{i}} = 0$ for $i = 0,\ldots, 6$) getting 10000000 which is -128. It is easy to see that -1 is represented by 11111111.

Generally, for numbers represented on n bits, we will get for **unsigned** types values in the range $[0, 2^{\text{n}} - 1]$, while for **signed** types this range will be $[-2^{\text{n-1}}, 2^{\text{n-1}} - 1]$. Specifically, for typical sizes of integer variables that are used in computer programs we get:

Bytes	Sign	Min	Max
1	signed	-128	127
1	unsigned	0	255
2	signed	-32 768	32 767
2	unsigned	0	65 535
4	signed	-2 147 483 648	2 147 483 647
4	unsigned	0	4 294 967 295
8	signed	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
8	unsigned	0	18 446 744 073 709 551 615

Table 4.1: integer types in C/C++

The type **char** (as the other character types) is an integer type, but is treated in many

contexts in a special way, as it is intended to represent characters. Therefore, we should never use variables of character types in arithmetic calculations, although formally it is perfectly possible. Values of character variables usually (but not necessarily) correspond to ASCII codes of characters. Only the first 128 values (from 0 to 127) are assigned a standard meaning in ASCII — others can be interpreted differently on different platforms in a locale dependent way. As a matter of fact, C/C++ standard does not even guarantee conformance with ASCII codes; in most implementations, however, numerical values corresponding to basic characters *do* conform to ASCII standard.

Let us consider the following example which illustrates declarations of variables of different types:

P13: decl1.cpp integer types

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      unsigned long int ul1 = 13UL;
6.      unsigned long     ul2 = 0xD;  // 13 in hex
7.      signed   short    ss1 = 015;  // 13 in oct
8.      short             ss2 = 13;   // 13 in dec
9.      unsigned char     aa1 = 65;
10.     signed   char     aa2 = 'A';  // ASCII('A') = 65
11.     int               aa3 = 65;
12.     int               aa4 = 'A';
13.     char              aa5 = '\101';   // 65 in oct
14.     char              aa6 = '\x41';   // 65 in hex
15.     cout << aa1 << " " << aa2 << endl
16.          << aa3 << " " << aa4 << endl
17.          << aa5 << " " << aa6 << endl;
18. }
```

The program prints

```
a A
65 65
a A
```

All variables ***aa1***... ***aa4*** have the same numerical value: 65 (which is the ASCII code of the letter '`A`'). If we print this values as the value of a variable of type **char**, we will see '`A`', as in this case the compiler knows that the value should be treated as the ASCII code of a letter. The same value but in a variable of other integer type is treated as itself, namely as the number 65. This is so even if a character literal was used to initialize the variable: what counts is the declared type of this variable and not the way it has been initialized (assigned a value) — see the declaration of ***aa4***.

Sometimes we need to specify the type of a literal constant. If a literal can be interpreted as being of integer type, it will be treated as **int** (equivalent to **signed int**). To indicate that this should be considered to be a **long**, we add a letter '`L`' (lower or upper case) or '`LL`' for **long long**; similarly, if we want to enforce unsigned type, we add the letter '`U`' (lower or upper case). These modifiers can be applied simultaneously in any order. Therefore, '`13UL`' is the literal of 13 as **unsigned long** (see variable ***ul1***), and '`1LL`' is 1 of type **long long**.

Numerical integer literals can be expressed in decimal notation, but also in octal and hexadecimal form. Integer value with leading 0 (zero) are treated as octal values; e.g., 037 will be interpreted as decimal $3 \cdot 8 + 7 = 31$ and 015 as $8 + 5 = 13$ (see ***ss1***). If we use

octal notation, we have to remember that only digits from 0 to 7 can be used. In particular, '\0' is used to express the character with ASCII code zero (this is called ***NUL*** character and plays an important rôle in the so called C-strings).

We use '0x' (that is zero-ex; 'x' can be in lower or upper case) for numbers in hexadecimal notation. Therefore, 0x2D amounts to decimal value $2 \cdot 16 + 13 = 45$, while 0xD denotes decimal 13 (see ***ul2***). In hexadecimal notation, one uses the letters A to F (lower or upper case) as digits from 10 to 15.

Character literals are denoted by a character enclosed in apostrophes (*not* double quotes!) — as on line defining ***aa2***. This can be problematic for characters which cannot be entered from the keyboard. In such situations, one can use the form '\ooo' (including apostrophes), where ooo denotes at most three-digit number in octal notation (leading zero can be omitted in this case). For an example, see definition of ***aa5***. In the next line, we used another notation: '\Xhh'. After the letter 'X' (or 'x') one can put at most two-digit number in hexadecimal notation denoted here as 'hh'. There is no way to use this form in decimal notation: only octal or hexadecimal are permitted.

Some characters have special meaning and can be expressed in yet another notation: a backslash followed by one letter:

\n	newline (LF)	\t	horizontal tab (HT)
\v	vertical tab (VT)	\b	backspace (BS)
\r	carriage return (CR)	\f	form feed (FF)
\a	bell (alert) (BEL)	\\	backslash
\?	question mark	\'	single quote
\"	double quote		

Table 4.2: Escaped characters

For example

```
cout << "\101nn\x61 Anna\n\x4Ao\145 Joe" << endl;
```

will print

```
Anna Anna
Joe Joe
```

as, e.g., ASCII code of upper-case 'A' is 65_{10}=101_8=41_{16}. Note that '\n' within the text enforces new line.

4.2.1 Useful aliases for integral types

As we have said, sizes of data of various integral types are not fixed by the standard (as they are in Java). Sometimes we want to be sure, that a declared variable has a given size. For example, if our variable should be signed and capable of holding values larger than 33 thousand, it should be at least 4-bytes long. On a platform where **int** is 2-byte long (what is rare but possible), we cannot use **int**, but **long** (which must be at least 4-byte long). In such situation, it is possible to avoid specifying a type explicitly: we can use aliases (typedefs), defined in header **cstdint**, of existing types which have the required size (**typedef** will be covered in section about typedefs later). For example, **int32_t** is an alias for an integral signed type which, on a given platform, is exactly 32-bit (4-byte) long — on most platform, it is just an alias of **int**. The number in the name of such aliases indicates the number of bits

(not bytes). Therefore, we can use **int8_t**, **int16_t**, **int32_t**, **int64_t** for *signed* integral types of the specified sizes. In a similar way, by prepending 'u' to the name of an alias, we get aliases of *unsigned* integral types: **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t**. All these names are really typedefs defined in namespace `std`, so their full names are **std::int32_t** etc.

The header `cstdint` defines also *preprocessor macros* which expand to minimum and maximum *values* of these types: **INT32_MIN**, **INT32_MAX**, **UINT32_MAX**, ect. Macros corresponding to minimum values are not defined for unsigned types, as for them the minimum is always 0. Since these are macros, their names cannot be prepended by `std::`.

Prev: 4.1 Introductory remarks **Up:** 4. Basic data types **Next:** 4.3 Floating-point types

4.3 Floating-point types

The standard floating-point types are **float**, **double** and **long double**. They approximately correspond to mathematical notion of real numbers (so they can contain both whole and fractional parts).

One **float**-type data usually occupies 4 bytes, and one **double** — 8 bytes, as we saw in the program `lengths.cpp`. The standard does not specify how the floating-point numbers are represented in the computer memory (as it *does* in Java). However, in almost all modern implementation, the IEEE 754 standard is adopted (IEEE 854 for **long double**). The type **long double** is rarely used (variables of this type usually occupy 12 or 16 bytes). As a matter of fact, using **float** s is not recommended either. During calculations they are silently converted to **double** s anyway, so their use leads to longer execution time and deteriorated accuracy of results.

Literals (constants) of floating-point numbers have usual form of numbers with the decimal point separating the whole and fractional parts (either part can be empty). The so called “scientific notation” is also understood. In this notation the letter 'e' or, equivalently, 'E' separates the mantissa and the exponent: the value is then equal to the product of mantissa and 10 raised to the power indicated by the exponent part (the decimal point in mantissa is not necessary). The exponent part must be a whole number (possibly with plus or minus sign). Literals in such form represent values of type **double**. If a **float** is needed, we append the letter 'f' (or 'F') immediately after the last digit (without any intervening spaces). Similarly, the letter 'L' appended to a literal indicates the type **long double** (lower case 'l' will also work but is not recommended, as it is hard to distinguish letter 'l' from digit '1').

```
float k = 1.23F, m = .1F, n = 3.F;
double x = 1.2, y = 50., z = 5e-3, v = 0.1e2;
long double u = 1.23L, v = 30.4e-20L;
```

In the above example, ***z*** is value $5 \cdot 10^{-3} = 0.005$ and ***v*** is $0.1 \cdot 10^{2} = 10$

4.4 Logical type

Logical type is named **bool** (not **boolean**, as in Java). As a matter of fact, any integer or pointer value can play the rôle of a logical value: value 0 (or **nullptr** for pointers) corresponds to **false** and *any* nonzero value (not necessarily 1) stands for **true**. This feature is a legacy from pure C where there is no separate logical type. It is however recommended to explicitly declare variables intended to represent logical values as **bool** s.

Let us consider an example. In the following program, the variable ***k*** will assume the value equal to the number of nonzero elements in the array ***tab*** before the first zero element (assuming that there *is* a zero element in this array!):

```
int tab[] = { 2, -4, -3, 0, 3 }, k = -1;
while ( tab[++k] );     // now k = 3
```

When ***k*** becomes 3, the element ***tab[3]*** is 0, what is interpreted as **false** and the loop stops. Before that, the values of ***tab[k]*** are all nonzero, so the expression in parentheses is interpreted as **true** and the loop continues.

Pointers (see one of the next sections in this chapter) can also be interpreted as boolean (logical) values: the empty pointer (**nullptr**) is, in logical context, interpreted as **false**; any nonempty pointer corresponds to **true**.

There are two literals with logical values: **true** and **false** (with obvious meaning). These two tokens are reserved words (keywords) and cannot be assigned any other meaning.

4.5 Enumeration types

The concept of **enumerated types** is very useful, although not indispensable.

Formally,

> **an enumerated type is a (usually small) set of named integer constants.**

For example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

defines an enumerated type **days**. Variables of this type can assume exactly seven values, which are listed ("enumerated") in braces following the name. All possible values of an enumeration of this type have their names (much like all possible values of the type **bool** are named — there are only two of them: **true** and **false**).

As we see, the definition of an enumerated type consists of the keyword **enum**, optionally a name of the type being defined and a list of named constants enclosed in braces. Type name can be omitted, but then, for the statement to make any sense, we usually define at least one variable of a newly created type:

```
enum {spades, hearts, diamonds, clubs} card1, card2;
...
card1 = spades;
card2 = card1;
...
if (card2 == clubs) { ... }
```

In the above example, we have created two variables of an anonymous enumerated type describing card suits. It will be impossible to declare any other variable of this type, as it hasn't any name. Anonymous enumerations are sometimes used when we want to introduce a named constant, but using normal **const** s would be inconvenient (as, e.g., in the definition of a class; see section on initialization lists).

Internally, the constants constituting an enumerated type, say **days**, are associated with consequtive integers starting from zero; in our case ***mon*** will be represented by 0, ***tue*** by 1,..., ***sun*** by 6. We can change this correspondence by assigning integer values to enumeration constants explicitly:

```
enum days {mon, tue=0, wed=0, thu=0, fri=0, sat, sun};
```

The rules are as follows:

- the first element will be represented by zero if not assigned another value explicitly. In our example above, ***mon*** will be associated with 0;
- any other element will be associated with the value by 1 larger than the value of the preceding constant, unless it is assigned a different value explicitly. In our example, all

constants from ***tue*** through ***fri*** correspond to 0. However, ***sat*** has not been assigned any value explicitly so it will be associated with 1 (as the preceding ***fri*** is 0). Similarly, ***sun*** will correspond to 2.

Values of enumeration constants do not have to be all different nor they have to have consecutive values in ascending order (they can also be negative). For example, our definition of **days** could look like this:

```
enum days {mon, tue=0, wed=0, thu=0, fri=0, sat, sat=3};
```

With this definition ***sat*** is still 1, but ***sun*** is now 3. There is no element corresponding to 2.

Under new standard, C++11, we can explicitly determine an integer type to be used for enumeration constants: we specify it after the enum's name and the colon:

```
enum suit : unsiged {spades, hearts, diamonds, clubs};
```

According to the new standard, the type behind enumaration constants, if not explicitly specified, must be **int** (it was not specified in older versions of the standard).

Have a look at the following program:

P14: enums.cpp Enumerated types

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  enum days {mon, tue=0, wed=0, thu=0, fri=0, sat, sun};  ❶
6.
7.  void info(days day) {
8.      static string  dayType[] = {"  weekday",            ❷
9.                                  " saturday", "  holiday"};
10.     int rate = 100*(1 + day);                            ❸
11.     cout << "Type of day:"  << dayType[day]   << ". "    ❹
12.          << "The rate is: " << rate << " USD" << endl;
13. }
14.
15. int main() {
16.     info(mon);            ❺
17.
18.     days day = sat;       ❻
19.     info(day);            ❼
20.
21.     day = sun;            ❽
22.     info(day);            ❾
23. }
```

We define an enumerated type **days** (❶). This definition is global (not within a function), so it will be visible everywhere, in particular in both **main** and **info** functions. Next, we define function **info**. Its parameter is of type **days**. This function is then called in **main**. We pass a variable of type **days** as the argument. In line ❺ it is the constant ***mon*** — one of the literal constants defined by **days**. In lines ❼ and ❾ it is the current value of the variable ***day*** declared as being of type **days**. As can be seen on line ❻, declaration of a variable of our type **days** does not differ from declarations of other variables: first we specify a type and then the name of a variable being declared/defined. Optionally, we can initialize newly created variable with a legal value; in our case legal values are only those defined by the **days** type. What is important here is the fact that we *cannot* assign an integer value to ***day***;

```
days day = 1; // WRONG
```

would be illegal. Although values of type **days** *are* represented by integers, such a conversion in not permitted in C++ (it is permitted, but not recommended, in C).

We modify the value of ***day*** on line ❽. Again, the assignment needs a value of type **days** (in this case it is ***sun***).

Inside the **inf** function, we define a three-element array of **string** s (ignore the keyword **static** — it will be explained later and actually is not necessary here). Elements of the array are referred to on line ❹. Notice that we use variable of type **days** as index which should normally be an integer. This is legal: in this case automatic conversion will be performed and the integer value corresponding to ***day*** will be used as an index. In our case these values are 0, 1 and 2 — just what is appropriate as an index of a three-element array. You can also notice a strange addition on line ❸: '`1 + day`'. The first argument is an **int**, the second is a **days** value; the latter will then be converted to the corresponding integer value and the result will be an **int** value. Again it should be emphasized, that the automatic conversion the other way around — from **int** to **days** — will not be performed. For example, it would be impossible to call the function **info** with '`info(k)`', ***k*** being an integer (even if its value would be 0, 1 or 2); any argument passed to **info** must have type **days**. This feature ensures that **info** will always be invoked with a legal value of the argument: it must be a **days** value and hence correspond to 0, 1 or 2. We then do not have to check whether the index is legal inside the function **info**: it will always be. Normally compilers are able to check if the type of an argument passed to a function is valid: enumerated types allow us to check also the validity of *values* of arguments.

The reader can check, that the program compiles and prints

```
Type of day:  weekday. The rate is: 100 USD
Type of day: saturday. The rate is: 200 USD
Type of day:  holiday. The rate is: 300 USD
```

Enumerations are often used as in the program above: to have control over validity of arguments passed to functions.

The new C++11 standard introduces in addition another way to define enumeration; we will cover this subject after introducing classes.

Subsections

- 4.6.1 Pointers to variables
- 4.6.2 Generic pointers

4.6 Pointers

Pointers play a fundamental rôle in both C and C++ programming languages. If you look into the code of virtually any program, you will easily notice that pointers are everywhere. Without understanding pointers it is impossible to understand C/C++.

Pointers allow us, among other things,

- to process dynamic data structures;
- to deal with memory, strings, arrays;
- to efficiently pass arguments to functions, including parameters that are themselves functions

Pointers to functions are a special type of pointers, that we will discuss in section on functions .

4.6.1 Pointers to variables

As we know, the type of a variable specifies the kind of information that can be stored in this variable. For pointers, it is the *address* of another variable in computer memory. This another variable can be of any type; in particular, it can be a pointer variable itself.

Usually, although not always, we have to specify the type of variables whose address can be assigned to a newly created pointer variable. In other words, pointers must “know” the type of variables they can point to. This makes it possible for compilers to perform type checking; it also allows for some operations on pointers, as for their completion information about the size of variables pointed to by pointers is required.

Values stored in pointer variables are addresses of other variables.

Suppose we want to create a pointer suited to store addresses of other variables which are of type **double**. Then we want to assign values to these pointers, i.e., we want them to store addresses of other *existing* variables of type **double**. Let's see how to do it considering the following example:

P15: pointers.cpp Pointers

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      double x, y = 1.5, u;                ❶
6.
7.      double *px;                          ❷
8.      px = &x;
9.
10.     double* py = &y;                     ❸
11.
12.     double *pz, *pu = &u, v;             ❹
13.
14.     cout << "1. *py = "  << *py
15.          << " y = " << y << endl;        ❺
16.
17.     x = 0.5;                             ❻
18.     cout << "2. *px = "  << *px
19.          << " x = " << x << endl;
20.
21.     *px = 5*x;                           ❼
22.     cout << "3. *px = "  << *px
23.          << " x = " << x << endl;
24.
25.     pz = px;                             ❽
26.     cout << "4. *pz = "  << *pz << endl;
27.
28.     *pu = v = *pz = 10;                  ❾
29.     cout << "5. *pu = "  << *pu
30.          << " u = " << u << " v = "
31.          << v << " x = " << x << endl;
32.
33.     cout << "6.  py = " <<  py << endl; ❿
34.     cout << "7. &py = " << &py << endl;
35. }
```

which outputs:

```
1. *py = 1.5 y = 1.5
2. *px = 0.5 x = 0.5
3. *px = 2.5 x = 2.5
4. *pz = 2.5
5. *pu = 10 u = 10 v = 10 x = 10
6.  py = 0xbffffa30
7. &py = 0xbffffa18
```

We declare and define (allocate memory for) three **double** variables on line ❶. One of them, ***y***, is initialized with the value 1.5, the others remain uninitialized, but, what is important here, they exist, i.e., they do have a well defined address in the computer memory.

In line ❷, we declare/define a **pointer** variable (or pointer, for short) which is suitable to point to (store addresses of) **double** s. After this line has been executed, the pointer ***px*** exists but has no particular value. It does not **point to** any variable.

The declaration of a pointer looks like this

```
Type *name;
```

or

```
Type* name;
```

or

```
Type * name;
```

i.e., it does not matter whether the asterisk is attached to the name of a type, to the name od a declared variable, or surrounded by blanks.

Pointer *px* declared as '`Type* px;`' is of type Type*.

This notation means that a variable named ***px*** will be a pointer which can store as its value addresses of variables of a type **Type**. How can we assign a value to ***px***? We see an example in the next line. This line assigns the value of the expression on the right hand side to the pointer on the left hand side. The value of the expression on the right hand side is just what we need: the address of an existing **double** variable, in this case of the variable ***x***. The variable ***x*** exists because it was created in line ❶; it has no well defined value, but it does have a well defined address. As we see, the symbol '`&`' stands for the **address-of operator** which returns the address of its operand (an expression it precedes). For this to be possible, the expression on the right hand side of the address operator must be an l-value (see the section on l-values).

If *var* is the identifier of a variable, then *&var* is the address of this variable.

Instead of first defining a pointer and then assigning it a value, we can do both in one statement, as it is shown on line ❸: we declare/define a pointer ***py*** and initialize its value with the address of the variable ***y***. Therefore, the value of ***py*** is the address of ***y***, while the value of ***y*** is 1.5 assigned on line ❶.

a variable whose address is the value of a pointer is said to be *pointed to* by this pointer.

This situation is illustrated in the figure. The variable ***py*** exists in the memory under some address — in the example it is `0xbffffa18` (addresses are usually written in hexadecimal notation). This variable occupies 8 bytes, although this can depend on platform.

The value stored under this address, i.e., the value of ***py*** is the address (in our example it is `0xbffffa30`) of the variable ***y*** of type **double**. This variable occupies 8 bytes and stores the value of ***y***: a number.

In line ❹ of the program `pointers.cpp`,

```
double *pz, *pu = &u, v;
```

we declare another three variables: ***pz***, ***pu*** and ***v***. This statement illustrates an important fact:

Declaring in one statement two or more pointers, one has to precede the identifier of each of them with an asterisk.

For example, the variable ***v*** has type **double**, and *not* a pointer (which would be of type **double***).

How can we reference a variable pointed to by a pointer? We do it by using the **indirection** (dereference) operator which is denoted by an asterisk (*).

If *pvar* is the identifier of a pointer, **pvar* is the name (an alias of the name) of a variable pointed to by *pvar*.

In our example ***py*** points to ***y*** (see line ❸), and the value of ***y*** is 1.5 (line ❶). Therefore, as ***py*** is an alias of the name of the variable pointed to by ***py***, it is currently just another name of ***y***. Printing on line ❺ the value of ***py***, we print 1.5 — the current value of ***y*** (see the first line of the program's output).

Similarly, in line ❻ of the program `pointers.cpp`, we assign the value 0.5 to ***x***. Printing now the value of ***px***, we print the value of the variable pointed to by ***px***, that is of ***x***, which is 0.5 (the second line of the output).

The expression ***px***, being currently a synonym of ***x***, can also be used on the left hand side of an assignment. An example can be seen on line ❼ of the program. The assignment is equivalent to '`x = 5*x`', which multiplies ***x*** by 5. Printing the values of both ***x*** and ***px***, we get the same result, namely 2.5, as expected.

We created a pointer ***pz*** on line ❹. In line ❽ we assign it the value of ***px***; this is the *address* of ***x***. Now ***x***, ***px*** and ***pz*** all mean the same thing: they are different names of ***x***. That is why printing the value of ***pz***, we get again 2.5.

In line ❾, we assign the value 10 to ***pz***. By doing so we assign this value to ***x***, as this is currently the variable pointed to by ***pz***. Printing ***x***, we thus get 10, as is seen in the fifth line of the output.

What will happen if we print the value of ***py***? This is a pointer, so its value is an address, not a number of type **double**. Therefore, printing from line ❿ will give us an address — in our case it will be the address of ***y*** which is `0xbffffa30`.

And what is the address if ***py*** itself: not the value stored in this pointer variable, but the address of this variable? To check it, we print the value of ***&py*** in the last line. We get `0xbffffa18`, what explains data illustrated in the figure.

Let us notice that the '`*`' operator plays a double rôle (neglecting the third: as binary operator it denotes multiplication). How can we differentiate between these two meanings? This should never be a problem: if an asterisk appears in a declaration, so to the left of it there is the name of a type, then it is the declaration of a pointer. In other cases, if the asterisk appears as a one-argument (unary) operator, it is to be interpreted as dereferencing operator. For example, in

```
int k = 7, *pk = &k, m = *pk;
```

the first asterisk means that ***pk*** is declared to be a pointer. To the left of the asterisk we have a comma, not the name of a type, but we know that this line is just an abbreviated

form of the sequence

```
int k   = 7;
int *pk = &k;
int m   = *pk;
```

so the rule applies. To the left of the second asterisk, we see the equal sign ' =''. It is therefore to be interpreted as the dereferencing operator: we initialize ***m*** with the value of the variable pointed to by ***pk*** — but ***pk*** was itself initialized with the address of ***k***, so the value assigned to ***m*** will be 7.

In Java, there is a special reference, called **null**, which “points to nothing”. In C/C++ we have similar notion: the empty pointer. Its value was accessible through the name (macro) ***NULL***, but one could also use literal value of 0 (zero). In the new standard there is a special keyword **nullptr** which denotes the empty pointer (and ***NULL*** should not be used anymore).

The pointer with value 0 is the “empty” pointer in C/C++. No existing object can have 0 as its address.

The value 0 can be assigned to any pointer of any type (it is better to use **nullptr** instead). This does *not* mean that there is a conversion from **int** (which is the type of literal 0) to pointer values. This particular value, 0, is an exception and it is the only integer value which can be converted to pointer type. Generally, pointers are *not* an integer type. For eample, the assignment '`pk = 7;`' is illegal and would cause a compilation error.

Before using a pointer, we have to assign it a value of the address of an existing variable. Neglecting to do so is a very common error:

```
int *q;
...
...
*q = 7;
```

After the first line, a pointer ***q*** exists but does not contain any address (its contents is purely accidental). In the last line we try to assign the value 7 to the integer variable pointed to by ***q***. But ***q*** does not point to any integer! We will therefore try to change the contents of an accidental region in memory. If we are lucky, this will be a region outside our address space and the system will refuse to do it: the program will crash with an error message like '`segmentation fault, core dumped`'. If not, the program will happily continue producing wrong results.

Consider the following program:

P16: pminmax.cpp Pointers

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     double x, y, *pmin, *pmax;      ❶
6.     cout << "Enter two numbers: ";
7.     cin  >> x >> y;
8.     if (x < y) {
```

```
9.          pmin = &x; pmax = &y;
10.     } else {
11.         pmin = &y; pmax = &x;
12.     }
13.     cout << "Min = " << *pmin << endl;
14.     cout << "Max = " << *pmax << endl;
15. }
```

which prints

```
Enter two numbers: 9.76 7.34
Min = 7.34
Max = 9.76
```

We define (in ❶) two **double** s and two pointers of type **double***. Then we read in two numbers to ***x*** and ***y***, and set the values of the pointers in such a way that ***pmin*** points to the smaller of the two numbers ***x*** and ***y***, while ***pmax*** points to the larger of them. Using the dereference operator we then print first the smaller and then the larger (last two lines).

4.6.2 Generic pointers

Sometimes we need a pointer which point to certain location in the computer's memory but without specifying the type of data stored at this address. Such pointers exist; they are called **generic pointers**. We declare them as being of type **void***.

In the following snippet, the variables ***pk***, ***p*** and ***q*** are pointers to **int** (i.e., their type is **int***), while ***pv*** is a generic pointer (of type **void***).

```
1.      int k = 8, *pk = &k, *p, *q;
2.      void *pv = pk;
3.      p = static_cast<int*>(pv);
4.      q = (int*)pv;
```

We see (in the second line) that it is possible to assign a value of type **int*** to a variable of type **void***. After this, ***pv*** contains as its value the same address as ***pk***: the address of ***k***. However, there is an important difference. The variable ***pk*** is of type **int***, so it “knows” that what it points to is an integer. Therefore, the expression ***pk*** is well defined: it is another name of the integer pointed to by ***pk***, i.e., of ***k***. The variable ***pv*** has type **void***: it holds an address, but it does not know of what. The expression `*pv` doesn't make sense: `pv` cannot be dereferenced because there is no information about the type of the object pointed to by this variable; in particular it is not known how many bytes this object occupies and how to interpret them.

Generic pointers cannot be dereferenced.

Suppose we have an address in a generic pointer and we know the type of the variable pointed to by this pointer, although the compiler might not be able to infer it. If we know what we are doing, we can enforce reinterpretation of a “raw” address in a generic pointer as an address of an object of a specific type. We do it by **casting** the value of our pointer onto another type: examples can be seen in the lines 3-4 of the snippet. We enforce explicit conversion of types from **void*** to **int***. This is denoted (as in Java) by specifying (in parentheses) the type onto which we want to cast (see the fourth line). This is the only way to do it in pure C. In C++, however, there is another way, illustrated in the third line. In fact, this method of casting is recommended in C++ — we postpone the details to section on

conversions . In our example, we copy the casted value of the generic pointer ***pv*** to both ***p*** and ***q***, so they contain the same address as ***pv***, but as they are of type **int*** they can be dereferenced.

What we have said so far about pointers does not explain their special rôle in C/C++. However, in the next chapters we will learn about arrays, functions, memory allocation, polymorphism etc., and everywhere we will encounter pointers.

Prev: 4.5 Enumeration types **Up:** 4. Basic data types **Next:** 4.7 References

Subsections

- 4.6.1 Pointers to variables
- 4.6.2 Generic pointers

4.6 Pointers

Pointers play a fundamental rôle in both C and C++ programming languages. If you look into the code of virtually any program, you will easily notice that pointers are everywhere. Without understanding pointers it is impossible to understand C/C++.

Pointers allow us, among other things,

- to process dynamic data structures;
- to deal with memory, strings, arrays;
- to efficiently pass arguments to functions, including parameters that are themselves functions

Pointers to functions are a special type of pointers, that we will discuss in section on functions .

4.6.1 Pointers to variables

As we know, the type of a variable specifies the kind of information that can be stored in this variable. For pointers, it is the *address* of another variable in computer memory. This another variable can be of any type; in particular, it can be a pointer variable itself.

Usually, although not always, we have to specify the type of variables whose address can be assigned to a newly created pointer variable. In other words, pointers must “know” the type of variables they can point to. This makes it possible for compilers to perform type checking; it also allows for some operations on pointers, as for their completion information about the size of variables pointed to by pointers is required.

Values stored in pointer variables are addresses of other variables.

Suppose we want to create a pointer suited to store addresses of other variables which are of type **double**. Then we want to assign values to these pointers, i.e., we want them to store addresses of other *existing* variables of type **double**. Let's see how to do it considering the following example:

P15: pointers.cpp Pointers

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      double x, y = 1.5, u;                ❶
6.
7.      double *px;                          ❷
8.      px = &x;
9.
10.     double* py = &y;                     ❸
11.
12.     double *pz, *pu = &u, v;             ❹
13.
14.     cout << "1. *py = "  << *py
15.          << " y = " << y << endl;        ❺
16.
17.     x = 0.5;                             ❻
18.     cout << "2. *px = "  << *px
19.          << " x = " << x << endl;
20.
21.     *px = 5*x;                           ❼
22.     cout << "3. *px = "  << *px
23.          << " x = " << x << endl;
24.
25.     pz = px;                             ❽
26.     cout << "4. *pz = "  << *pz << endl;
27.
28.     *pu = v = *pz = 10;                  ❾
29.     cout << "5. *pu = "  << *pu
30.          << " u = " << u << " v = "
31.          << v << " x = " << x << endl;
32.
33.     cout << "6.  py = " <<  py << endl; ❿
34.     cout << "7. &py = " << &py << endl;
35. }
```

which outputs:

```
1. *py = 1.5 y = 1.5
2. *px = 0.5 x = 0.5
3. *px = 2.5 x = 2.5
4. *pz = 2.5
5. *pu = 10 u = 10 v = 10 x = 10
6.  py = 0xbffffa30
7. &py = 0xbffffa18
```

We declare and define (allocate memory for) three **double** variables on line ❶. One of them, ***y***, is initialized with the value 1.5, the others remain uninitialized, but, what is important here, they exist, i.e., they do have a well defined address in the computer memory.

In line ❷, we declare/define a **pointer** variable (or pointer, for short) which is suitable to point to (store addresses of) **double** s. After this line has been executed, the pointer ***px*** exists but has no particular value. It does not **point to** any variable.

The declaration of a pointer looks like this

```
Type *name;
```

or

```
Type* name;
```

or

```
Type * name;
```

i.e., it does not matter whether the asterisk is attached to the name of a type, to the name od a declared variable, or surrounded by blanks.

Pointer *px* declared as '`Type* px;`' is of type Type*.

This notation means that a variable named ***px*** will be a pointer which can store as its value addresses of variables of a type **Type**. How can we assign a value to ***px***? We see an example in the next line. This line assigns the value of the expression on the right hand side to the pointer on the left hand side. The value of the expression on the right hand side is just what we need: the address of an existing **double** variable, in this case of the variable ***x***. The variable ***x*** exists because it was created in line ❶; it has no well defined value, but it does have a well defined address. As we see, the symbol '`&`' stands for the **address-of operator** which returns the address of its operand (an expression it precedes). For this to be possible, the expression on the right hand side of the address operator must be an l-value (see the section on l-values).

If *var* is the identifier of a variable, then *&var* is the address of this variable.

Instead of first defining a pointer and then assigning it a value, we can do both in one statement, as it is shown on line ❸: we declare/define a pointer ***py*** and initialize its value with the address of the variable ***y***. Therefore, the value of ***py*** is the address of ***y***, while the value of ***y*** is 1.5 assigned on line ❶.

a variable whose address is the value of a pointer is said to be *pointed to* by this pointer.

This situation is illustrated in the figure. The variable ***py*** exists in the memory under some address — in the example it is `0xbffffa18` (addresses are usually written in hexadecimal notation). This variable occupies 8 bytes, although this can depend on platform.

The value stored under this address, i.e., the value of ***py*** is the address (in our example it is `0xbffffa30`) of the variable ***y*** of type **double**. This variable occupies 8 bytes and stores the value of ***y***: a number.

In line ❹ of the program `pointers.cpp`,

```
double *pz, *pu = &u, v;
```

we declare another three variables: ***pz***, ***pu*** and ***v***. This statement illustrates an important fact:

Declaring in one statement two or more pointers, one has to precede the identifier of each of them with an asterisk.

For example, the variable ***v*** has type **double**, and *not* a pointer (which would be of type **double***).

How can we reference a variable pointed to by a pointer? We do it by using the **indirection** (dereference) operator which is denoted by an asterisk (*).

If *pvar* is the identifier of a pointer, **pvar* is the name (an alias of the name) of a variable pointed to by *pvar*.

In our example ***py*** points to ***y*** (see line ❸), and the value of ***y*** is 1.5 (line ❶). Therefore, as ***py*** is an alias of the name of the variable pointed to by ***py***, it is currently just another name of ***y***. Printing on line ❺ the value of ***py***, we print 1.5 — the current value of ***y*** (see the first line of the program's output).

Similarly, in line ❻ of the program `pointers.cpp`, we assign the value 0.5 to ***x***. Printing now the value of ***px***, we print the value of the variable pointed to by ***px***, that is of ***x***, which is 0.5 (the second line of the output).

The expression ***px***, being currently a synonym of ***x***, can also be used on the left hand side of an assignment. An example can be seen on line ❼ of the program. The assignment is equivalent to '`x = 5*x`', which multiplies ***x*** by 5. Printing the values of both ***x*** and ***px***, we get the same result, namely 2.5, as expected.

We created a pointer ***pz*** on line ❹. In line ❽ we assign it the value of ***px***; this is the *address* of ***x***. Now ***x***, ***px*** and ***pz*** all mean the same thing: they are different names of ***x***. That is why printing the value of ***pz***, we get again 2.5.

In line ❾, we assign the value 10 to ***pz***. By doing so we assign this value to ***x***, as this is currently the variable pointed to by ***pz***. Printing ***x***, we thus get 10, as is seen in the fifth line of the output.

What will happen if we print the value of ***py***? This is a pointer, so its value is an address, not a number of type **double**. Therefore, printing from line ❿ will give us an address — in our case it will be the address of ***y*** which is `0xbffffa30`.

And what is the address if ***py*** itself: not the value stored in this pointer variable, but the address of this variable? To check it, we print the value of ***&py*** in the last line. We get `0xbffffa18`, what explains data illustrated in the figure.

Let us notice that the '`*`' operator plays a double rôle (neglecting the third: as binary operator it denotes multiplication). How can we differentiate between these two meanings? This should never be a problem: if an asterisk appears in a declaration, so to the left of it there is the name of a type, then it is the declaration of a pointer. In other cases, if the asterisk appears as a one-argument (unary) operator, it is to be interpreted as dereferencing operator. For example, in

```
int k = 7, *pk = &k, m = *pk;
```

the first asterisk means that ***pk*** is declared to be a pointer. To the left of the asterisk we have a comma, not the name of a type, but we know that this line is just an abbreviated

form of the sequence

```
int k   = 7;
int *pk = &k;
int m   = *pk;
```

so the rule applies. To the left of the second asterisk, we see the equal sign ' =''. It is therefore to be interpreted as the dereferencing operator: we initialize ***m*** with the value of the variable pointed to by ***pk*** — but ***pk*** was itself initialized with the address of ***k***, so the value assigned to ***m*** will be 7.

In Java, there is a special reference, called **null**, which “points to nothing”. In C/C++ we have similar notion: the empty pointer. Its value was accessible through the name (macro) ***NULL***, but one could also use literal value of 0 (zero). In the new standard there is a special keyword **nullptr** which denotes the empty pointer (and ***NULL*** should not be used anymore).

The pointer with value 0 is the “empty” pointer in C/C++. No existing object can have 0 as its address.

The value 0 can be assigned to any pointer of any type (it is better to use **nullptr** instead). This does *not* mean that there is a conversion from **int** (which is the type of literal 0) to pointer values. This particular value, 0, is an exception and it is the only integer value which can be converted to pointer type. Generally, pointers are *not* an integer type. For eample, the assignment '`pk = 7;`' is illegal and would cause a compilation error.

Before using a pointer, we have to assign it a value of the address of an existing variable. Neglecting to do so is a very common error:

```
int *q;
...
...
*q = 7;
```

After the first line, a pointer ***q*** exists but does not contain any address (its contents is purely accidental). In the last line we try to assign the value 7 to the integer variable pointed to by ***q***. But ***q*** does not point to any integer! We will therefore try to change the contents of an accidental region in memory. If we are lucky, this will be a region outside our address space and the system will refuse to do it: the program will crash with an error message like '`segmentation fault, core dumped`'. If not, the program will happily continue producing wrong results.

Consider the following program:

P16: pminmax.cpp Pointers

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     double x, y, *pmin, *pmax;      ❶
6.     cout << "Enter two numbers: ";
7.     cin  >> x >> y;
8.     if (x < y) {
```

```
9.          pmin = &x; pmax = &y;
10.     } else {
11.         pmin = &y; pmax = &x;
12.     }
13.     cout << "Min = " << *pmin << endl;
14.     cout << "Max = " << *pmax << endl;
15. }
```

which prints

```
Enter two numbers: 9.76 7.34
Min = 7.34
Max = 9.76
```

We define (in ❶) two **double** s and two pointers of type **double***. Then we read in two numbers to ***x*** and ***y***, and set the values of the pointers in such a way that ***pmin*** points to the smaller of the two numbers ***x*** and ***y***, while ***pmax*** points to the larger of them. Using the dereference operator we then print first the smaller and then the larger (last two lines).

4.6.2 Generic pointers

Sometimes we need a pointer which point to certain location in the computer's memory but without specifying the type of data stored at this address. Such pointers exist; they are called **generic pointers**. We declare them as being of type **void***.

In the following snippet, the variables ***pk***, ***p*** and ***q*** are pointers to **int** (i.e., their type is **int***), while ***pv*** is a generic pointer (of type **void***).

```
1.      int k = 8, *pk = &k, *p, *q;
2.      void *pv = pk;
3.      p = static_cast<int*>(pv);
4.      q = (int*)pv;
```

We see (in the second line) that it is possible to assign a value of type **int*** to a variable of type **void***. After this, ***pv*** contains as its value the same address as ***pk***: the address of ***k***. However, there is an important difference. The variable ***pk*** is of type **int***, so it “knows” that what it points to is an integer. Therefore, the expression ***pk*** is well defined: it is another name of the integer pointed to by ***pk***, i.e., of ***k***. The variable ***pv*** has type **void***: it holds an address, but it does not know of what. The expression `*pv` doesn't make sense: `pv` cannot be dereferenced because there is no information about the type of the object pointed to by this variable; in particular it is not known how many bytes this object occupies and how to interpret them.

Generic pointers cannot be dereferenced.

Suppose we have an address in a generic pointer and we know the type of the variable pointed to by this pointer, although the compiler might not be able to infer it. If we know what we are doing, we can enforce reinterpretation of a “raw” address in a generic pointer as an address of an object of a specific type. We do it by **casting** the value of our pointer onto another type: examples can be seen in the lines 3-4 of the snippet. We enforce explicit conversion of types from **void*** to **int***. This is denoted (as in Java) by specifying (in parentheses) the type onto which we want to cast (see the fourth line). This is the only way to do it in pure C. In C++, however, there is another way, illustrated in the third line. In fact, this method of casting is recommended in C++ — we postpone the details to section on

conversions . In our example, we copy the casted value of the generic pointer ***pv*** to both ***p*** and ***q***, so they contain the same address as ***pv***, but as they are of type **int*** they can be dereferenced.

What we have said so far about pointers does not explain their special rôle in C/C++. However, in the next chapters we will learn about arrays, functions, memory allocation, polymorphism etc., and everywhere we will encounter pointers.

Prev: 4.5 Enumeration types **Up:** 4. Basic data types **Next:** 4.7 References

Subsections

- 4.6.1 Pointers to variables
- 4.6.2 Generic pointers

4.6 Pointers

Pointers play a fundamental rôle in both C and C++ programming languages. If you look into the code of virtually any program, you will easily notice that pointers are everywhere. Without understanding pointers it is impossible to understand C/C++.

Pointers allow us, among other things,

- to process dynamic data structures;
- to deal with memory, strings, arrays;
- to efficiently pass arguments to functions, including parameters that are themselves functions

Pointers to functions are a special type of pointers, that we will discuss in section on functions .

4.6.1 Pointers to variables

As we know, the type of a variable specifies the kind of information that can be stored in this variable. For pointers, it is the *address* of another variable in computer memory. This another variable can be of any type; in particular, it can be a pointer variable itself.

Usually, although not always, we have to specify the type of variables whose address can be assigned to a newly created pointer variable. In other words, pointers must “know” the type of variables they can point to. This makes it possible for compilers to perform type checking; it also allows for some operations on pointers, as for their completion information about the size of variables pointed to by pointers is required.

Values stored in pointer variables are addresses of other variables.

Suppose we want to create a pointer suited to store addresses of other variables which are of type **double**. Then we want to assign values to these pointers, i.e., we want them to store addresses of other *existing* variables of type **double**. Let's see how to do it considering the following example:

P15: pointers.cpp Pointers

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      double x, y = 1.5, u;                ❶
6.
7.      double *px;                          ❷
8.      px = &x;
9.
10.     double* py = &y;                     ❸
11.
12.     double *pz, *pu = &u, v;             ❹
13.
14.     cout << "1. *py = "  << *py
15.          << " y = " << y << endl;        ❺
16.
17.     x = 0.5;                             ❻
18.     cout << "2. *px = "  << *px
19.          << " x = " << x << endl;
20.
21.     *px = 5*x;                           ❼
22.     cout << "3. *px = "  << *px
23.          << " x = " << x << endl;
24.
25.     pz = px;                             ❽
26.     cout << "4. *pz = "  << *pz << endl;
27.
28.     *pu = v = *pz = 10;                  ❾
29.     cout << "5. *pu = "  << *pu
30.          << " u = " << u << " v = "
31.          << v << " x = " << x << endl;
32.
33.     cout << "6.  py = " <<  py << endl; ❿
34.     cout << "7. &py = " << &py << endl;
35. }
```

which outputs:

```
1. *py = 1.5 y = 1.5
2. *px = 0.5 x = 0.5
3. *px = 2.5 x = 2.5
4. *pz = 2.5
5. *pu = 10 u = 10 v = 10 x = 10
6.  py = 0xbffffa30
7. &py = 0xbffffa18
```

We declare and define (allocate memory for) three **double** variables on line ❶. One of them, ***y***, is initialized with the value 1.5, the others remain uninitialized, but, what is important here, they exist, i.e., they do have a well defined address in the computer memory.

In line ❷, we declare/define a **pointer** variable (or pointer, for short) which is suitable to point to (store addresses of) **double** s. After this line has been executed, the pointer ***px*** exists but has no particular value. It does not **point to** any variable.

The declaration of a pointer looks like this

```
Type *name;
```

or

```
Type* name;
```

or

```
Type * name;
```

i.e., it does not matter whether the asterisk is attached to the name of a type, to the name od a declared variable, or surrounded by blanks.

Pointer *px* declared as '`Type* px;`' is of type Type*.

This notation means that a variable named ***px*** will be a pointer which can store as its value addresses of variables of a type **Type**. How can we assign a value to ***px***? We see an example in the next line. This line assigns the value of the expression on the right hand side to the pointer on the left hand side. The value of the expression on the right hand side is just what we need: the address of an existing **double** variable, in this case of the variable ***x***. The variable ***x*** exists because it was created in line ❶; it has no well defined value, but it does have a well defined address. As we see, the symbol '`&`' stands for the **address-of operator** which returns the address of its operand (an expression it precedes). For this to be possible, the expression on the right hand side of the address operator must be an l-value (see the section on l-values).

If *var* is the identifier of a variable, then *&var* is the address of this variable.

Instead of first defining a pointer and then assigning it a value, we can do both in one statement, as it is shown on line ❸: we declare/define a pointer ***py*** and initialize its value with the address of the variable ***y***. Therefore, the value of ***py*** is the address of ***y***, while the value of ***y*** is 1.5 assigned on line ❶.

a variable whose address is the value of a pointer is said to be *pointed to* by this pointer.

This situation is illustrated in the figure. The variable ***py*** exists in the memory under some address — in the example it is `0xbffffa18` (addresses are usually written in hexadecimal notation). This variable occupies 8 bytes, although this can depend on platform.

The value stored under this address, i.e., the value of ***py*** is the address (in our example it is `0xbffffa30`) of the variable ***y*** of type **double**. This variable occupies 8 bytes and stores the value of ***y***: a number.

In line ❹ of the program `pointers.cpp`,

```
double *pz, *pu = &u, v;
```

we declare another three variables: ***pz***, ***pu*** and ***v***. This statement illustrates an important fact:

Declaring in one statement two or more pointers, one has to precede the identifier of each of them with an asterisk.

For example, the variable ***v*** has type **double**, and *not* a pointer (which would be of type **double***).

How can we reference a variable pointed to by a pointer? We do it by using the **indirection** (dereference) operator which is denoted by an asterisk (*).

If *pvar* is the identifier of a pointer, **pvar* is the name (an alias of the name) of a variable pointed to by *pvar*.

In our example ***py*** points to ***y*** (see line ❸), and the value of ***y*** is 1.5 (line ❶). Therefore, as ***py*** is an alias of the name of the variable pointed to by ***py***, it is currently just another name of ***y***. Printing on line ❺ the value of ***py***, we print 1.5 — the current value of ***y*** (see the first line of the program's output).

Similarly, in line ❻ of the program `pointers.cpp`, we assign the value 0.5 to ***x***. Printing now the value of ***px***, we print the value of the variable pointed to by ***px***, that is of ***x***, which is 0.5 (the second line of the output).

The expression ***px***, being currently a synonym of ***x***, can also be used on the left hand side of an assignment. An example can be seen on line ❼ of the program. The assignment is equivalent to '`x = 5*x`', which multiplies ***x*** by 5. Printing the values of both ***x*** and ***px***, we get the same result, namely 2.5, as expected.

We created a pointer ***pz*** on line ❹. In line ❽ we assign it the value of ***px***; this is the *address* of ***x***. Now ***x***, ***px*** and ***pz*** all mean the same thing: they are different names of ***x***. That is why printing the value of ***pz***, we get again 2.5.

In line ❾, we assign the value 10 to ***pz***. By doing so we assign this value to ***x***, as this is currently the variable pointed to by ***pz***. Printing ***x***, we thus get 10, as is seen in the fifth line of the output.

What will happen if we print the value of ***py***? This is a pointer, so its value is an address, not a number of type **double**. Therefore, printing from line ❿ will give us an address — in our case it will be the address of ***y*** which is `0xbffffa30`.

And what is the address if ***py*** itself: not the value stored in this pointer variable, but the address of this variable? To check it, we print the value of ***&py*** in the last line. We get `0xbffffa18`, what explains data illustrated in the figure.

Let us notice that the '`*`' operator plays a double rôle (neglecting the third: as binary operator it denotes multiplication). How can we differentiate between these two meanings? This should never be a problem: if an asterisk appears in a declaration, so to the left of it there is the name of a type, then it is the declaration of a pointer. In other cases, if the asterisk appears as a one-argument (unary) operator, it is to be interpreted as dereferencing operator. For example, in

```
int k = 7, *pk = &k, m = *pk;
```

the first asterisk means that ***pk*** is declared to be a pointer. To the left of the asterisk we have a comma, not the name of a type, but we know that this line is just an abbreviated

form of the sequence

```
int k   = 7;
int *pk = &k;
int m   = *pk;
```

so the rule applies. To the left of the second asterisk, we see the equal sign ' =''. It is therefore to be interpreted as the dereferencing operator: we initialize ***m*** with the value of the variable pointed to by ***pk*** — but ***pk*** was itself initialized with the address of ***k***, so the value assigned to ***m*** will be 7.

In Java, there is a special reference, called **null**, which “points to nothing”. In C/C++ we have similar notion: the empty pointer. Its value was accessible through the name (macro) ***NULL***, but one could also use literal value of 0 (zero). In the new standard there is a special keyword **nullptr** which denotes the empty pointer (and ***NULL*** should not be used anymore).

The pointer with value 0 is the “empty” pointer in C/C++. No existing object can have 0 as its address.

The value 0 can be assigned to any pointer of any type (it is better to use **nullptr** instead). This does *not* mean that there is a conversion from **int** (which is the type of literal 0) to pointer values. This particular value, 0, is an exception and it is the only integer value which can be converted to pointer type. Generally, pointers are *not* an integer type. For eample, the assignment '`pk = 7;`' is illegal and would cause a compilation error.

Before using a pointer, we have to assign it a value of the address of an existing variable. Neglecting to do so is a very common error:

```
int *q;
...
...
*q = 7;
```

After the first line, a pointer ***q*** exists but does not contain any address (its contents is purely accidental). In the last line we try to assign the value 7 to the integer variable pointed to by ***q***. But ***q*** does not point to any integer! We will therefore try to change the contents of an accidental region in memory. If we are lucky, this will be a region outside our address space and the system will refuse to do it: the program will crash with an error message like '`segmentation fault, core dumped`'. If not, the program will happily continue producing wrong results.

Consider the following program:

P16: pminmax.cpp Pointers

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     double x, y, *pmin, *pmax;      ❶
6.     cout << "Enter two numbers: ";
7.     cin  >> x >> y;
8.     if (x < y) {
```

```
9.          pmin = &x; pmax = &y;
10.     } else {
11.         pmin = &y; pmax = &x;
12.     }
13.     cout << "Min = " << *pmin << endl;
14.     cout << "Max = " << *pmax << endl;
15. }
```

which prints

```
Enter two numbers: 9.76 7.34
Min = 7.34
Max = 9.76
```

We define (in ❶) two **double** s and two pointers of type **double***. Then we read in two numbers to ***x*** and ***y***, and set the values of the pointers in such a way that ***pmin*** points to the smaller of the two numbers ***x*** and ***y***, while ***pmax*** points to the larger of them. Using the dereference operator we then print first the smaller and then the larger (last two lines).

4.6.2 Generic pointers

Sometimes we need a pointer which point to certain location in the computer's memory but without specifying the type of data stored at this address. Such pointers exist; they are called **generic pointers**. We declare them as being of type **void***.

In the following snippet, the variables ***pk***, ***p*** and ***q*** are pointers to **int** (i.e., their type is **int***), while ***pv*** is a generic pointer (of type **void***).

```
1.      int k = 8, *pk = &k, *p, *q;
2.      void *pv = pk;
3.      p = static_cast<int*>(pv);
4.      q = (int*)pv;
```

We see (in the second line) that it is possible to assign a value of type **int*** to a variable of type **void***. After this, ***pv*** contains as its value the same address as ***pk***: the address of ***k***. However, there is an important difference. The variable ***pk*** is of type **int***, so it “knows” that what it points to is an integer. Therefore, the expression ***pk*** is well defined: it is another name of the integer pointed to by ***pk***, i.e., of ***k***. The variable ***pv*** has type **void***: it holds an address, but it does not know of what. The expression `*pv` doesn't make sense: `pv` cannot be dereferenced because there is no information about the type of the object pointed to by this variable; in particular it is not known how many bytes this object occupies and how to interpret them.

Generic pointers cannot be dereferenced.

Suppose we have an address in a generic pointer and we know the type of the variable pointed to by this pointer, although the compiler might not be able to infer it. If we know what we are doing, we can enforce reinterpretation of a “raw” address in a generic pointer as an address of an object of a specific type. We do it by **casting** the value of our pointer onto another type: examples can be seen in the lines 3-4 of the snippet. We enforce explicit conversion of types from **void*** to **int***. This is denoted (as in Java) by specifying (in parentheses) the type onto which we want to cast (see the fourth line). This is the only way to do it in pure C. In C++, however, there is another way, illustrated in the third line. In fact, this method of casting is recommended in C++ — we postpone the details to section on

conversions . In our example, we copy the casted value of the generic pointer ***pv*** to both ***p*** and ***q***, so they contain the same address as ***pv***, but as they are of type **int*** they can be dereferenced.

What we have said so far about pointers does not explain their special rôle in C/C++. However, in the next chapters we will learn about arrays, functions, memory allocation, polymorphism etc., and everywhere we will encounter pointers.

Prev: 4.5 Enumeration types **Up:** 4. Basic data types **Next:** 4.7 References

4.7 References

References are often called *aliases* of variables. They were introduced in C++; in the C language they do not appear. As a matter of fact, there are no 'reference variables' — references, as their name suggests, always *refer to* existing, “normal” variables of a well defined type (**int**, **double**, **string**,...). Generally

references are just other names of existing variables.

One can ask if assigning two or more names to a variable makes any sense and can be of any use. As will become clear later, they *are* very useful and often indispensable.

First of all, if references are other names of existing variables, it is impossible to create a reference which does not refer to a variable (as it *is* possible to create a pointer which does not point to anything). Attaching a reference to a variable is permanent: when a reference is created, it cannot be attached (bound) to another variable. The target variable, of which a reference is another name, must be specified when this reference is created. From now on, the name of the reference and the name of the variable it is bounded to are synonyms. When we operate on a variable (for example, change its value), we can refer to this variable by its name or by the name of the associated reference.

A reference, say ***refk***, can be created and bound to an existing variable like this:

P17: refer.cpp References

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int k = 5;
6.      int &refk = k;
7.
8.      cout << "refk = " << refk << endl;
9.      cout << "   k = " << k    << endl;
10.
11.     k = 7;
12.
13.     cout << "refk = " << refk << endl;
14.     cout << "   k = " << k    << endl;
15.
16.     refk = 9;
17.
18.     cout << "refk = " << refk << endl;
19.     cout << "   k = " << k    << endl;
20. }
```

First, we declare/define a variable ***k*** of type **int**. Then, in the next line, we declare a reference ***refk*** to this variable.

Declaration of a reference has the form '`Type &ref = variable`'

where **Type** is the name of a type, ***ref*** is any name we want to assign to the reference, and ***variable*** is the name of an *existing* variable of type **Type** to which the reference is to refer. After such declaration

the name of the type of *ref* is Type&.

In our example ***refk*** refers to ***k***, and, as seen from the output

```
refk = 5
   k = 5
refk = 7
   k = 7
refk = 9
   k = 9
```

the names ***k*** and ***refk*** are equivalent (they refer to the same variable, i.e., to the same region in memory). We can modify or print this variable using the name ***k*** as well as ***refk***.

We remember (section on pointers), that the symbol '`&`' denotes the address operator. Here it indicates a reference. The rule is similar to the one applying to pointers: if the symbol '`&`' in a declaration follows a name of a type, it indicates a reference, in other contexts, when used as a unary operator, it plays the rôle of the address-of operator. No confusion is syntactically possible.

Consider another example:

P18: pointref.cpp Pointers and references

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main () {
5.      int k = 7, *p = &k, &refk = *p, m = 9;  ❶
6.
7.      p = &m;                                 ❷
8.      k = 11;
9.
10.     cout << "  *p = " <<   *p << endl;      ❸
11.     cout << "refk = " << refk << endl;      ❹
12. }
```

which gives the output

```
  *p = 9
refk = 11
```

Let us analyze declarations/definitions in ❶:

- '`k = 7`' is the definition of an **int** variable with initialization;
- '`*p = &k`' is the definition of a pointer ***p*** to **int** (see section on pointers) whose value is initialized with the address of ***k*** ('`&`' follows an equal sign, i.e., it is the address-of operator);
- '`&refk = *p`' is the definition of a reference ***refk*** to the variable of type **int** which is

specified on the right hand side of the equal sign. However, on the right hand side there is ' `*p`', which is ' ***p*** dereferenced', i.e., the variable pointed to by ***p***, which is ***k***. Consequently, ***refk*** will be permanently bound to (will be another name of) ***k***;
- '`m = 9`' is the definition of a “normal” **int** variable ***m*** with initialization.

We change the value of ***p*** on line ❷ — now it points to ***m***, which has value 9. Printing `*p` in line ❸gives us 9, as expected. The value of ***k*** is changed to 11. Printing ***refk*** on line ❹ we get 11. This means that, although ***refk*** was initially associated with `*p`, it remains bound to the variable originally pointed to by `*p`: changes of `p` do not affect this association.

Defining more than one reference in one statement, the symbol ' `&`' must precede all names of references. The declaration

```
double x = 3, &y = x, *z = &x, &u = *z;
```

declares two variables: ***x*** of type **double** initialized with 3 and a pointer ***z*** of type **double*** initialized with the address of ***x***. Notice that ***y*** is not a new variable: it is a reference to (another name of) ***x***. Similarly, ***u*** is a reference to the variable currently pointed to by ***z***, i.e., it is yet another name of ***x***. Therefore, the variable ***x*** acquires three names and this cannot be changed. On the other hand, the pointer ***z*** currently points to ***x***, but this can be changed later in the program.

That is all fine, but who needs variables with three different names? True, the sense of it remains unclear. However, the power of references will become evident when we discuss passing arguments to functions and returning values from functions. We will postpone this discussion to section on reference arguments and functions returning references .

References can refer to arrays, but there is no such thing like an array of references.

---

5. Static arrays and pointers

In the previous chapter, we started describing pointer types. Now, we will extend our discussion to encompass the issue of the so called **pointer arithmetic**. We will also show the intimate relation between pointers and arrays in C/C++. Talking about arrays, we will narrow our attention to static arays; the so called dynamic arrays will be the subject of the section on memory allocation .

Subsections

- 5.1 Defining arrays
- 5.2 Array type
- 5.3 Pointer arithmetic
- 5.4 Character arrays (C-strings)
- 5.5 Multidimensional arrays
  - 5.5.1 Matrices
  - 5.5.2 Arrays of C-strings
- 5.6 Arrays of type *std::array*
- 5.7 Vectors (*std::vector*)

5.1 Defining arrays

The array is a fundamental data structure in C/C++ and many other languages (although there exist languages where the rôle of lists or maps is even more emphasized). Generally,

> **an array is an ordered aggregate of data items of the same type with common identifier. Elements of this aggregate can be accessed by their index, which is a whole number indicating their position in the aggregate.**

Elements of the array are numbered with consecutive integers *starting from 0 (zero)*. It means that the first element is assigned the index 0, the second — index 1, etc. It follows that the last index has the value *one less* than the number of elements. Number of elements of an array is called its **size** or **dimension**. If ***arr*** is the identifier of an array, then its element with the index ***k*** (i.e., its $k+1$-th element) is denoted by `arr[k]`; we use *square* brackets to indicate an index.

How can one declare/define an array? Any static array must have its dimension known at compilation time. Therefore, declaring an array we have to inform the compiler what its size is. If an array is not a member of an object, it is allocated on the stack in exactly the same way as other local variables (and disappears when the flow of control leaves the block it was defined in).

Let us consider a few ways in which arrays can be defined:

```
1.      const int N = 20; // or constexpr int N = 20;
2.      int tab1[100],
3.          tab2[N],
4.          tab3[] = {1,2,3,4,5},
5.          tab4[5] = {1,2};
6.
7.      int tab5[]{1,2,3,4,5}, // C++11
8.          tab6[5]{1,2},      // C++11
9.          tab7[5]{};         // C++11
```

The array ***tab1*** defined in the second line is an array with 100 elements, each of them of the type **int**. They will be indexed with consecutive whole numbers from 0 through 99.

In the third line, a similar array ***tab2*** of **int** s is defined: this time the size of the array is 20. To indicate the size, we used the value of a variable ***N*** — this is acceptable if this variable has been defined as **const** or, better yet, as **constexpr**. This is actually the case: as we see in the first line, ***N*** has the type **const int** — an integer whose value is fixed at the moment of creation and cannot be modified. Actually, most compilers would accept a non-**const** variable: this, however wouldn't conform to the standard and could lead to nonportability of the code. The fact that dimensions of arrays have to be known at compile time is the reason why we call them static — their size cannot be changed at run time (e.g., their dimension cannot depend on any input data).

Arrays created on the stack as local variables must be declared with a size which is known at compile time; the size must be given as numeric literal, the name of a constant (integer variable declared as const or constexpr), or as an expression involving literals and constants.

Let us emphasize that ***tab1*** and ***tab2*** exist, but no particular values are assigned to their elements. The contents of the region of memory they occupy is more or less accidental.

The situation will be different with arrays ***tab3*** and ***tab4***. Consecutive elements of array ***tab3*** will be initialized with the values specified in braces. The size will be fixed (it is still a static array) but we do not have to specify it: the compiler will count how many initializers have been used and will allocate an array with that many elements.

And what about ***tab4***? We specified both the size (5) and a list of initializers, but only two of them. In this case the size will be 5, the first two elements will be initialized with the values provided, and the rest of them *will also* be initialized with value 0 (zero) of the appropriate type. Thus, if we want to define a big array and initialize all its elements with zero, we can write

```
int tab[100000] = {};
```

because providing brace-initializer, even empty, we force initialization of all elements of the array.

Note also that under the new standard (C++11), equal sign before the brace-initializer may be left out, as demonstrated on the last three lines of the example above.

Prev: 5. Static arrays and pointers **Up:** 5. Static arrays and pointers **Next:** 5.2 Array type

5.2 Array type

Suppose we define an array ***tab*** in a block (e.g., the body of a function)

```
int tab[100];
```

From now on, to the end of the current block where the declaration/definition is visible, the name ***tab*** refers to a variable of the type *100-element array of elements of type* ***int***; the name of this type is **int[100]**. Note that the size is an element of type specification: types **int[6]** and **int[5]** are different.

Therefore, the value of `sizeof(tab)` will be the size, in bytes, of the whole array (400, if `sizeof(int)` is 4). Knowing this number, one can easily recalculate the size of an array: in our case it will be

```
sizeof(tab) / sizeof(int)
```

or, in a form independent of the type of array

```
sizeof(tab) / sizeof(tab[0])
```

Unfortunately, this does not mean that in C/C++ arrays “know their size”, as they do in Java. The size is known and can be calculated as shown above *only* in the block where this array was defined. This is not very useful, as to define it we had to know its size anyway. When an array is used in almost any expression, in particular passed to a function, the information about its size is lost: it is converted to pointer which points to the first element of the array, so the only information which is passed is the address where the region in memory occupied by the array begins. After the definition

```
int tab[20];
```

the variable ***tab*** can be treated (almost) as a pointer of type **int* const** pointing to ***tab[0]***. The modifier **const** here means that the contents of the array pointed to by ***tab*** can be modified, but we cannot make ***tab*** to point to another region of memory (this will be explained later). Therefore, the expression ***tab*** is simply another name of the first element of the array ***tab***.

This conversion from array to pointer is particularly important when passing arrays to functions. When we pass an array, we pass (by value) the address of its first element and *nothing more*. In particular, we do not pass any information on the array's size. Actually, we do not even pass the information that this address represents the address of an array (and not the address of a “normal” single variable of the corresponding type). Therefore, if an array is an argument of a function, the corresponding parameter in the declaration/definition of this function is in fact of pointer type, and *not* of array type! Look at an example:

P19: arrays.cpp Arrays as arguments

```
1. #include <iostream>
```

```
using namespace std;

void fun1(double t[]) {
   cout << "Size of \'t\' in fun1: " << sizeof(t) << endl;
   cout << "Value of *t in fun1: "   << t[0]      << endl;
}

void fun2(double* t) {
   cout << "Size of \'t\' in fun2: " << sizeof(t) << endl;
   cout << "Value of *t in fun2: "   << t[0]      << endl;
}

int main() {
   double t[] = {6,2,3,2,1};
   cout << "Size of \'t\' in main: " << sizeof(t) << endl;
   cout << "Value of *t in main: "   << *t        << endl;
   fun1(t);
   fun2(t);
}
```

Running this program gives (on a platform with 8-byte pointers)

```
Size of 't' in main: 40
Value of *t in main: 6
Size of 't' in fun1: 8
Value of *t in fun1: 6
Size of 't' in fun2: 8
Value of *t in fun2: 6
```

In the **main** function, we define an array ***t***. Printing the value of `sizeof(t)` we get 40, which is not a surprise: 5 elements 8 bytes each make indeed 40 bytes. The value of ***t*** is 6 because this is the value of the first element of the array.

We then pass the array ***t*** to two functions. They are basically identical, the only difference being the declared type of the first parameter: in **fun1** we have ' **double t[]**', while in **fun2** it is ' **double* t**'. As we can see from the output, both these forms are equivalent. Although the form ' **double t[]**' may suggest array type, it is not so: used as a specification of the parameter type, it is exactly equivalent to **double***. In both cases it is just an address that is received by the function. That is why in both cases `sizeof(t)` yields 8 (4 on a 32-bit machine) when used inside either of these functions: it is just the size of a pointer, *not* the size of the array it is pointing to.

In the **fun2** function, the parameter is of type **double*** and no arrays at all are mentioned here. Nevertheless, we used an expression with an index, `t[0]`, as if ***t*** were an array. This is an example of “pointer arithmetic”, which we will discuss in the next section. For now, we have to remember that

passing an array to a function, we pass *only* the address where it starts in memory; information on its size is *not* passed.

It follows that almost always when we pass an array to a function, we have to pass its size separately. Passing arrays to functions is performed, as usually, by value, i.e., a *copy* of argument's value is pushed on the stack and becomes “visible” by the function called. However, this value is the value of a *pointer* (the address of the original array). Knowing this address, function has access to the *original* array, not to its copy. If the function modifies elements of the array that was passed to it, these modification will be in fact performed on the *original array*, as can be seen from the example below:

P20: arrfunc.cpp Passing arrays to functions

```
1. #include <iostream>
2. using namespace std;
3.
4. int* fun(int *arr1, int *arr2, int size) {
5.     int i, x, y, s1{}, s2{};
6.     for (i = 0; i < size; ++i) {
7.         x = arr1[i];
8.         y = arr2[i];
9.         arr1[i] = y;
10.        arr2[i] = x;
11.        s1 += y;
12.        s2 += x;
13.    }
14.    return s1 > s2 ? arr1 : arr2;
15. }
16.
17. void printArr(int *arr, int size) {
18.    for (int i = 0; i < size; ++i)
19.        cout << arr[i] << " ";
20.    cout << endl;
21. }
22.
23. int main() {
24.    int arr1[]{1,2,3}, arr2[]{4,5,6}, *arr3;
25.
26.    cout << "arr1 before: "; printArr(arr1,3);
27.    cout << "arr2 before: "; printArr(arr2,3);
28.    arr3 = fun(arr1,arr2,3);
29.    cout << "arr1  after: "; printArr(arr1,3);
30.    cout << "arr2  after: "; printArr(arr2,3);
31.    cout << "arr3       : "; printArr(arr3,3);
32. }
```

Note that in definitions of functions **fun** and **printArr** the parameters corresponding to arrays (***arr1***, ***arr2***, ***arr***) are of type **int*** and sizes are passed separately through parameters of type **int**.

Calling these functions, we pass, by value, the addresses of the first elements of the arrays. The function **fun** modifies elements of two arrays: elements from ***arr1*** are copied to the corresponding elements of ***arr2*** and *vice versa.* The function calculates also the sum of all elements of the two arrays and returns either ***arr1*** or ***arr2*** depending on which sum turned out to be larger (the construct used here means "if `s1>s2` then return ***arr1***, if not, return ***arr2***" — more about it in chapter on operators). Consequently, the return type of the function was declared as **int*** — the value returned will be the address of the "larger" array.

The program prints:

```
arr1 before: 1 2 3
arr2 before: 4 5 6
arr1  after: 4 5 6
arr2  after: 1 2 3
arr3       : 4 5 6
```

In the **main** function we print, using the function **printArr**, all three arrays. We can see that indeed the contents of ***arr1*** and ***arr2*** has been swapped. The returned value is assigned to ***arr3*** which is of type **int***, not an array type. This will be the address of the "larger" array, that is the address of ***arr1*** or ***arr2*** — printing ***arr3*** we can see that it must have been the address of ***arr1***. Note, that we pass ***arr3*** to function **printArr** although it is not declared as

an array: this is perfectly legal, because the function expects just an address of an integer, which is the case for the value of ***arr3***.

Prev: 5.1 Defining arrays **Up:** 5. Static arrays and pointers **Next:** 5.3 Pointer arithmetic

5.3 Pointer arithmetic

One can add (and subtract) whole numbers to pointers. This does not mean that pointers are of an integer type: such operations are defined in a special way.

Suppose ***p*** is a pointer of type **Type*** pointing to an element of an array, and ***shift*** is of an integer type. Then,

> **the value of the expression '`p+shift`' is the address contained in *p* increased by *shift* times the size of one variable of type Typ (i.e., `sizeof(Type)`).**

Therefore, if ***shift*** is 2, and `sizeof(Type)` is 4, as for **int**, then the value of '`p+shift`' is the address contained in ***p*** increased by 8 (= $2 \cdot 4$). Similarly, for the type **double** this address would be increased by 16, as normally `sizeof(double)` is 8. It is obvious now, that the pointer arithmetic cannot apply to generic pointers (of type **void***): without information on type, it would not be known by how many bytes an address should be increased.

Let us have a look at the following program

P21: arythmpoi.cpp Pointer arithmetic

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int tab[] = {11,22,33,44,55}, i = 3, *p, *q;
6.
7.      p = &tab[0] + 3;                          ❶
8.      cout << "*p     = " << *p << endl;
9.
10.     p = p - 2;                                ❷
11.     cout << "*p     = " << *p << endl;
12.
13.     q = tab;                                  ❸
14.     cout << "*(q+2) = " << *(q+2) << endl;
15.     cout << "q[2]   = " << q[2]   << endl;
16.
17.     cout << "q[i]   = " << q[i] << endl;      ❹
18.     cout << "i[q]   = " << i[q] << endl;      ❺
19. }
```

which prints

```
*p     = 44
*p     = 22
*(q+2) = 33
q[2]   = 33
```

```
q[i]    = 44
i[q]    = 44
```

The value of the expression `&tab[0]` on line ❶ is equal to the *address* of the first element of the array, i.e., the address of ***tab[0]*** which itself has the value 11. Note that

the value of `&tab[0]` is the same as the value of `tab`.

If we add 3 to the value of `&tab[0]`, we will get the same address but increased by three multiples of `sizeof(int)`, that is by $3\times4 = 12$. This will be exactly the address of the fourth element of the array (with index 3); its value is 44, what is what we get when printing the value of ***p***.

We then decrease ***p*** by 2 (❷). The address contained in ***p*** will be decreased by two multiples of 4, that is by 8. This will be the address of the second element of the array (with index 1 and value 22).

On line ❸ we assign the value of ***tab*** (which is the address of `tab[0]`) to the variable ***q*** of type **int***. What is now `*(q+2)` from the next line? As ***q*** points to ***tab[0]***, adding 2 to ***q*** produces the address of ***tab[2]***. Dereferencing it, we get the value of ***tab[2]*** which is 33. Therefore, `*(q+2)` is just ***tab[2]***.

Generally, if ***p*** is a pointer and ***i*** is of integer type, then

`p[i]` is *exactly* equivalent to `*(p+i)`.

Actually, the form `*(p+i)` is more fundamental; the other one, `p[i]`, is a syntactic sugar which can be treated as just a convenience notation. The compiler will change `p[i]` to `*(p+i)` anyway. Let us compare, for example, the last two lines of the program. The expression `q[i]` on line ❹ means `*(q+i)`. But on line ❺ we have `i[q]`. The variable ***i*** is neither a pointer nor an array. Moreover, a pointer has been used in the rôle of an index! Bizarre as it is, this is perfectly legal: the compiler will transform it to `*(i+q)`, and this is equivalent, from the point of view of pointer arithmetic, to `*(q+i)`, and hence to `q[i]`.

Let us now analyse the following example:

P22: littlebig.cpp Conversions of pointers

```cpp
#include <iostream>
using namespace std;

int main() {
    // higher byte: 'a'; lower byte: 'b'
    short sh = 'b'+256*'a';

    void *v = static_cast<void*>(&sh);
    char *c = static_cast<char*>(v);
    cout << "Order in memory: first "
         << c[0] << " then " << c[1] << endl;
}
```

We create a two-byte variable of type **short** and assign it the value `'b'+256*'a'`, which is the number whose higher byte is the ASCII code of 'a' (97) and the lower is equal to

ASCII code of 'b' (98). The address of this number is then converted first to generic type **void*** and then to the type **char***. We use the converting operator **static_cast** which will be fully explained in section on conversions . After these operations ***c*** contains the address of ***sh***, but is of type **char*** (recall that **char** is a one-byte variable). We can now look “into” the variable ***sh*** treating it as the two-element array of characters. The question is: does ***c*** points to lower or higher byte of ***sh***? In other words are bytes of **short** s written from the lowest to the highest or the other way around? The program prints

```
Order in memory: first b then a
```

which means that the lower byte ('b') was written first and has lower address. This corresponds to the so called **little endian** architecture. We would have got the result

```
Order in memory: first a then b
```

if our architecture were **big endian** (both these terms come from *Gulliver's Travels* by Jonathan Swift). It is worthwhile to remember that big endian order of bytes is traditionally used when transferring data through the network.

As we have already noted, it is illegal to add (subtract) integers to generic pointers (of type **void***); they do not carry any information on the type of variables pointed to and hence on their size. Similarly, it would not make any sense to use this construct to pointers pointing to functions (such pointers will be discussed in section on function pointers).

However, subtracting two pointers of the same type does make sense if they point to elements of the same array. For two such pointers, say ***p1*** and ***p2***, the result of subtraction `p2-p1` is the number of elements of this array which are located in memory between the addresses contained in ***p2*** and ***p1***. If the elements pointed to by the two pointers have indices ***i1*** and ***i2***, the same result can be calculated as `i2-i1`.

Adding pointers to pointers does not have any sense and is illegal.

Prev: 5.2 Array type **Up:** 5. Static arrays and pointers **Next:** 5.4 Character arrays (C-strings)

5.4 Character arrays (C-strings)

Character arrays and pointers are somewhat special in C/C++. It stems from the fact that in traditional C there is no special string type — strings are implemented just as arrays of characters in which a special character indicates where the string ends. This special character has the ASCII code equal to 0 (zero) and does not have any graphical representation. It can be entered into a program literally as '\0' (*including* apostrophes). It is called ***NUL*** character and should be not mixed with the null *pointer*, which was denoted as ***NULL*** (note double L), and is not recommended any more (use just literal 0, or, even better, **nullptr**).

Let us see how to define a C-string in a program. Various possibilities are illustrated in the following program. The line '`char tab1[] = "Betty";`', creates an array of six (*sic!*) characters from the literally given string: five letters of the name and the ***NUL*** character, '\0', as the last one. It has to be there to indicate the end of the string, so the compiler will add it automatically.

P23: chararr.cpp C-strings: character arrays

```
1. #include <iostream>
2. using namespace std;
3.
4. void print (const char* t) {
5.     cout << "String: " << t << endl;
6. }
7.
8. int main() {
9.     char  t1[] = "Betty";
10.    char  t2[] = {'K', 'i', 't', 't', 'y', '\0'};
11.    const char *t3 = "Alice";
12.    cout << "sizeof t1   : "   << sizeof(t1)    << endl;
13.    cout << "sizeof t2   : "   << sizeof(t2)    << endl;
14.    cout << "sizeof t3   : "   << sizeof(t3)    << endl;
15.    cout << "sizeof \'Eve\': " << sizeof("Eve") << endl;
16.    t1[0] = 'X';
17.    t2[0] = 'X';
18.    //tab3[0] = 'X'; // WRONG
19.
20.    print(t1);
21.    print(t2);
22.    print(t3);
23. }
```

Therefore, as can be seen from the output

```
sizeof t1   : 6
sizeof t2   : 6
sizeof t3   : 8
sizeof 'Eve': 4
```

```
String: Xetty
String: Xitty
String: Alice
```

the size of the array ***t1*** is 6. The array ***t2*** has been initialized explicitly with an array initializer in the form already known to us from the previous section. The size of this array is also 6, but note that now we had to add the terminating '\0' character by hand — in this case it would not have been added automatically. In both cases we end up with normal static arrays with elements of type **char**. In particular, we can modify elements of these arrays as we did later in the program.

Definition of ***t3*** is more interesting. We define ***t3*** as being of type **const char*** and initialize it with the address of the literal string. What is important is that ***t3*** is *not* an array, just a "plain" pointer and therefore its size is 8 (or 4 on a 32-bit machine). The similarity with ***t1*** is deceiving: ***t1*** denotes a true array (which can be converted to a pointer, but is not a pointer). It was allocated locally on the stack and characters from the literal string that we used to initialize it were copied to this array (including the '\0' character at the end). The definition of ***t3*** is different: the array itself was created *somewhere* and pointer ***t3*** was initialized with its address. The array will be stored in a read-only region of memory and generally is unmodifiable! That is why we had to declare the type of the pointer as **const char*** instead of **char*** (we will say more about **const** in one of the next chapters). Note, that the compiler would allow us to define ***t3*** as having the type **char*** (without **const**), but trying to modify the array pointed to by ***t3*** would lead to a crash anyway; this inconsequence is of historical origin...

Let us notice that when passing a C-string to a function, we do not have to pass its size: the presence of '\0' at the end of any legal C-string allows the function to find out what their size is.

Finally, note that the stream insertion operator '<<' treats values of type **char*** (or **const char***) in a special way. Normally, '`cout « str`' where ***str*** is of type **char*** should print the value of ***str*** which is an address (this would be the case if ***str*** were of type, e.g., **int***). However, for values of type **char*** the compiler assumes that what we want to have printed is the C-string pointed to by this pointer and not the address of this string. Therefore, all bytes starting from the location in memory pointed to by ***str*** will be treated as containing codes of consecutive characters to be printed, until '\0' (i.e., 0) is seen. It is our responsibility to ensure that '\0' is indeed there!

We will continue the subject of C-strings in a special chapter on strings .

Subsections

- 5.5.1 Matrices
- 5.5.2 Arrays of C-strings

5.5 Multidimensional arrays

C/C++ supports multidimensional arrays, although their implementation is not as efficient as in other languages (notably Fortran). Actually, any n-dimensional array is in fact a one-dimensional array of pointers which point to $(n - 1)$-dimensional arrays,... and so on recursively. Similarly to “normal” arrays, multidimensional *character* arrays are somewhat special, so we will discuss them separately. Of course, in this chapter we only talk about static arrays, i.e., allocated locally (on the stack) and with dimensions known at compile time.

5.5.1 Matrices

We will focus our attention on two-dimensional arrays (matrices). Let us consider the following definition:

```
int tab[2][4] = { {1,2,3}, {5,6,7,8} };
```

We declared a two-dimensional array (matrix) of integers with 2 rows and 4 columns (i.e., 8 elements). Conventionally, the first index corresponds to rows and the second to columns (what we call a row and what a column is a matter of convention, elements are located in memory linearly anyway).

The matrix ***tab*** is initialized with some initial values. The initializer on the right hand side of the assignment has a form of two arrays corresponding to rows of the matrix. It is important to remember the order in which elements will be stored: in C/C++ the order is row by row (*not* column by column, as in Fortran): the first row (with index 0) goes first

```
tab[0][0]  tab[0][1]  tab[0][2]  tab[0][3]
```

followed by the second row (with index 1)

```
tab[1][0]  tab[1][1]  tab[1][2]  tab[1][3]
```

In the initializer we had to enclose the elements of each row in braces: it was necessary here, because not all elements of the matrix were set. In the first row only three values have been set: the fourth will be assigned the value 0. We could have initialized all elements

```
int tab[2][4] = { 1,2,3,4,5,6,7,8 };
```

without braces indicating rows. This is, however, not very readable and should be avoided.

Additionally, clearly indicating consecutive rows clearly shows the true nature of a matrix as an array of arrays.

The elements of an array can be referenced by indicating row and column indices, each in separate pair of brackets — ***tab[1][2]*** has value 7 in our example; numbering starts from 0.

Suppose we defined a two-dimensional array with ***dim1*** rows and ***dim2*** columns.

```
constexpr int dim1 = ...;
constexpr int dim2 = ...;

int tab[dim1][dim2];
```

We know that elements are stored row by row. Given the address of the first element of the first row and indices ***m*** and ***n***, how to calculate the address of the element ***tab[m][n]***? This element belongs to $(m + 1)$-th row (with index ***m***), so to get to its location we have to jump over the first m rows, each with ***dim2*** elements; therefore we have to "skip" $m \cdot \mathrm{dim2}$ elements. Then we have to jump over the first ***n*** elements of the row that ***tab[m][n]*** belongs to, as it is $(n + 1)$-st element of this row. It follows that in total we have to skip

$$\mathrm{shift} = n + m \cdot \mathrm{dim2}$$

elements. What is important here is the fact that to calculate this offset, we do not need to know what the first dimension of our matrix is (i.e., we do not have to know ***dim1***). Generally, in order to be able to calculate the offset of an element with given indices (relative to the beginning of the array), we need all dimensions except the first. Let us look at an example:

P24: arr2dim.cpp Multidimensional arrays

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void exchange(int tab[][4], int w1, int w2) {
5.      int t,k;
6.      for (k = 0; k < 4; k++) {
7.          t           = tab[w1][k];
8.          tab[w1][k] = tab[w2][k];
9.          tab[w2][k] = t;
10.     }
11. }
12.
13. void printArray(int tab[][4], int dim1) {    ❶
14.     int w,k;
15.     for (w = 0; w < dim1; w++) {
16.         for (k = 0; k < 4; k++)
17.             cout << tab[w][k] << " ";
18.         cout << endl;
19.     }
20. }
21.
22. int main() {
23.     int tab[3][4] = { {1,2,3,4}, {5,6}, {1} };
24.
25.     cout << "Array before:\n"; printArray(tab,3);
26.     exchange(tab,0,1);
27.     cout << "Array  after:\n"; printArray(tab,3);
28. }
```

The program prints:

```
Array before:
```

```
1 2 3 4
5 6 0 0
1 0 0 0
Array  after:
5 6 0 0
1 2 3 4
1 0 0 0
```

In the main function we define a two-dimensional array with sizes 3×4. We initialize some of its elements; the rest will be initialized with zeros.

The function **exchange** exchanges two rows with indices passed as the last two arguments of the function. We do not need to understand all the details of the function; what is important is the form in which its first parameter was declared: `int[][4]`. No information on the first dimension of the function is passed. We could have specified it by declaring `int[3][4]`, but this would be just a documenting information for the reader of the code: the compiler would completely ignore it anyway.

However, information on the second dimension is necessary. Otherwise, the offset of elements relative to the beginning of the array could not be calculated inside the function.

The function **printArray** prints the elements of the array passed to it. This function needs to know how many rows to print, so the first dimension of the array must be passed (line ❶). Note, however, that it is passed separately through an additional argument, not as an element of the array's type.

Let us analyze the nature of the variable ***tab*** from the point of view of arrays/pointers correspondence. Suppose we define following definition:

```
int tab[dim1][dim2];
```

We remember from the section on pointer arithmetic that ***tab[i]*** is equivalent to `*(tab+i)`. Therefore, ***tab[i][j]*** will correspond to `*(tab[i]+j)` and should have an integer value, so `tab[i]+j` should be a pointer to an integer. As ***j*** is just an **int**, `tab[i]` should be a pointer to an integer (pointing to the beginning of the row with index ***i***). But this is just `*(tab+i)` and for this to be a pointer, `tab` must be a pointer to pointer to integer, that is something of the type **int****.

There is another problem with our arrays: the function **exchange** in the program **arr2dim.cpp** exchanges two rows with indices passed as arguments. However, it will only work for arrays with 4 columns, because of type declaration of its first parameter. If we had another matrix, with different number of columns, we would have to write another function, although its task would be identical. How could we write a more general function exchanging two given rows of a matrix?

One possibility is to define a function that accepts not a matrix but explicitly a one-dimensional array of pointers to rows, which are themselves one-dimensional arrays of **int** s. The advantage of this approach is the fact that passing an array we do not have to specify the size in its type: the type will be just a pointer. Therefore, we get:

P25: arr2dim2.cpp Matrix as an array of pointers

```
1. #include <iostream>
```

```
2.  using namespace std;
3.
4.  void exchange(int* arr[], int w1, int w2,
5.                            int dim1, int dim2) {
6.      for (int k = 0; k < dim2; k++) {
7.          int t      = arr[w1][k];
8.          arr[w1][k] = arr[w2][k];
9.          arr[w2][k] = t;
10.     }
11. }
12.
13. void printArray(int* arr[], int dim1, int dim2) {
14.     for (int w = 0; w < dim1; w++) {
15.         for (int k = 0; k < dim2; k++)
16.             cout << arr[w][k] << " ";
17.         cout << endl;
18.     }
19. }
20.
21. int main() {
22.     int tt[3][4] = { {1,2,3,4}, {7,6}, {1} };        ❶
23.
24.     const int dim1 = 3;
25.     const int dim2 = 4;
26.
27.     int* arr[dim1];                                  ❷
28.     for (int i = 0; i < dim1; i++) arr[i] = tt[i];
29.
30.     cout << "Before:\n"; printArray(arr,dim1,dim2);
31.     exchange(arr,0,1,dim1,dim2);
32.     cout << "After:\n";  printArray(arr,dim1,dim2);
33. }
```

The first parameter of the function **exchange** has the type *one-dimensional array of pointers to **int** s*. There is no information on any sizes here; sizes are sent to the function separately and can be different for different calls corresponding to matrices of different shapes.

There is, however a problem: given a matrix ***tt*** (line ❶), we have to construct an array of pointers pointing to rows — this is required by the function. Therefore, we have to define additional, auxiliary array ***arr*** of pointers (❷). Its size is equal to number of rows and its elements are assigned the values of addresses of rows of ***tt***, i.e., ***tt[0]***, ***tt[1]*** and ***tt[2]***. Thanks to pointer arithmetic, inside the function we can use ***arr*** as a matrix with two indices (because elements of ***tab*** are not of type **int** but of type **int***). The result

```
Before:
1 2 3 4
7 6 0 0
1 0 0 0
After:
7 6 0 0
1 2 3 4
1 0 0 0
```

shows that both functions, **exchange** and **printArray**, work properly although no information about sizes of arrays have been hard-coded in their definition.

5.5.2 Arrays of C-strings

In order to get used to all these complications, we will consider a special but important example of (one-dimensional) array of C-strings which can be treated as two-dimensional array of characters.

P26: arrstr.cpp Arrays of C-strings

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const char **v;
6.      const char *t[] = {"abcd", "efghi", "jklmno" };  ❶
7.      v = t;                                            ❷
8.      cout << "v+2          = " << v+2          << endl;
9.      cout << "v[2]         = " << v[2]         << endl;
10.     cout << "*(v+2)       = " << *(v+2)       << endl;
11.
12.     cout << "*(*(t+1)+2) = " << *(*(t+1)+2) << endl; ❸
13.     cout << "t[1][2]      = " << t[1][2]      << endl;
14.
15.     cout << "*(*(v+1)+2) = " << *(*(v+1)+2) << endl;
16.     cout << "v[1][2]      = " << v[1][2]      << endl;
17. }
```

What is ***t*** on line ❶? This in an array (because its name is followed by an opening square bracket) of pointers to **const char** s (because it follows type declaration **const char***). The pointers from the table point to arrays of characters (C-strings) initialized with strings given in literal form on the right hand side of the assignment (that is why we used **const** rather than "normal" **char** s). As *'array of elements of type **Type**'* corresponds to type **Type***, our ***t***, being an array of elements of type **const char***, corresponds to type **const char****. This makes the assignment from line ❷ legal.

Next we print the value of `v+2`. This is the address from ***v*** shifted by two lengths of elements of ***v***. Its type is **const char****, not **const char*** — therefore when printing it we get an address as such, not a string of characters.

In the next two lines we print the value of the variable *pointed to* by `v+2`: the type of this variable is **const char***, so a string is printed (all characters starting from the address `*(v+2)`, which is equivalent to ***v[2]***, to the first ***NUL*** character encountered). As ***v[2]*** is the address of the third string, we will see `jklmno` printed.

On lines starting from ❸, we do the same thing on the level of individual characters. The expression `*(t+1)` is equivalent to ***t[1]***, which is the pointer to the first character of the second string. Shifting it by two lengths of one **char**, we get `*(t+1)+2` which will be the address of the third character of the string pointed to by `*(t+1)`; dereferencing it, we get `*(*(t+1)+2)` which is equivalent to ***t[1][2]*** and this is the third character of the second string (the letter 'g'). We can check it looking at the output:

```
v+2          = 0x7fff364d7080
v[2]         = jklmno
*(v+2)       = jklmno
*(*(t+1)+2)  = g
t[1][2]      = g
*(*(v+1)+2)  = g
v[1][2]      = g
```

Last two lines show that all these operations can be performed on ***v*** (which was assigned the value of ***t***), although it has not been explicitly declared to be an array: it is just a pointer of the appropriate type.

Arrays of strings are used extensively in many C/C++ programs and it is very important to

understand what is going on “under the hood”. For example, command-line arguments are passed to the program as such an array — see the introductory section and the program arguments.cpp there.

Although the array of command-line arguments is declared as ' ***char* argv[]*** ' and not ' ***const char* argv[]*** ', we should never try to modify the C-strings pointed to by the elements of ***argv***!

To conclude this section, let us see how to read a C-string from standard input stream (usually connected to the keyboard). We have to remember that all white spaces are treated as separators and not parts of strings. Let us look at an example:

P27: readarr.cpp Reading C-strings

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      char nap1[100], nap2[100];
6.
7.      cout << "\nEnter two strings separated by spaces: ";
8.      cin  >>  nap1 >> nap2;
9.      cout << "\nString 1: " << nap1 << endl;
10.     cout <<   "String 2: " << nap2 << endl << endl;
11. }
```

We create character arrays ***nap1*** and ***nap2*** with sizes sufficient to store expected strings (by the way, it is a simple but generally *not* recommended way of reading strings). Then we read data to this arrays using normal mechanism provided by ***cin***. As we already noticed in section on reading input data , reading one piece of data terminates when a white space is encountered, so we can read two strings as below:

```
Enter two strings separated by a space: 'Linux Windows'

String 1: 'Linux
String 2: Windows'
```

Let us notice that enclosing a string in apostrophes does not help: a space is treated as a separator anyway. We can see also that the mechanism provided by using the object ***cin*** ensures that the terminating character ' \0' will be added automatically. Better ways of reading data from input streams will be covered in chapter on I/O operations .

Subsections

- 5.5.1 Matrices
- 5.5.2 Arrays of C-strings

5.5 Multidimensional arrays

C/C++ supports multidimensional arrays, although their implementation is not as efficient as in other languages (notably Fortran). Actually, any n-dimensional array is in fact a one-dimensional array of pointers which point to $(n - 1)$-dimensional arrays,... and so on recursively. Similarly to “normal” arrays, multidimensional *character* arrays are somewhat special, so we will discuss them separately. Of course, in this chapter we only talk about static arrays, i.e., allocated locally (on the stack) and with dimensions known at compile time.

5.5.1 Matrices

We will focus our attention on two-dimensional arrays (matrices). Let us consider the following definition:

```
int tab[2][4] = { {1,2,3}, {5,6,7,8} };
```

We declared a two-dimensional array (matrix) of integers with 2 rows and 4 columns (i.e., 8 elements). Conventionally, the first index corresponds to rows and the second to columns (what we call a row and what a column is a matter of convention, elements are located in memory linearly anyway).

The matrix ***tab*** is initialized with some initial values. The initializer on the right hand side of the assignment has a form of two arrays corresponding to rows of the matrix. It is important to remember the order in which elements will be stored: in C/C++ the order is row by row (*not* column by column, as in Fortran): the first row (with index 0) goes first

```
tab[0][0]  tab[0][1]  tab[0][2]  tab[0][3]
```

followed by the second row (with index 1)

```
tab[1][0]  tab[1][1]  tab[1][2]  tab[1][3]
```

In the initializer we had to enclose the elements of each row in braces: it was necessary here, because not all elements of the matrix were set. In the first row only three values have been set: the fourth will be assigned the value 0. We could have initialized all elements

```
int tab[2][4] = { 1,2,3,4,5,6,7,8 };
```

without braces indicating rows. This is, however, not very readable and should be avoided.

Additionally, clearly indicating consecutive rows clearly shows the true nature of a matrix as an array of arrays.

The elements of an array can be referenced by indicating row and column indices, each in separate pair of brackets — ***tab[1][2]*** has value 7 in our example; numbering starts from 0.

Suppose we defined a two-dimensional array with ***dim1*** rows and ***dim2*** columns.

```
constexpr int dim1 = ...;
constexpr int dim2 = ...;

int tab[dim1][dim2];
```

We know that elements are stored row by row. Given the address of the first element of the first row and indices ***m*** and ***n***, how to calculate the address of the element ***tab[m][n]***? This element belongs to $(m + 1)$-th row (with index ***m***), so to get to its location we have to jump over the first m rows, each with ***dim2*** elements; therefore we have to "skip" $m \cdot \mathrm{dim2}$ elements. Then we have to jump over the first ***n*** elements of the row that ***tab[m][n]*** belongs to, as it is $(n + 1)$-st element of this row. It follows that in total we have to skip

$$\mathrm{shift} = n + m \cdot \mathrm{dim2}$$

elements. What is important here is the fact that to calculate this offset, we do not need to know what the first dimension of our matrix is (i.e., we do not have to know ***dim1***). Generally, in order to be able to calculate the offset of an element with given indices (relative to the beginning of the array), we need all dimensions except the first. Let us look at an example:

P24: arr2dim.cpp Multidimensional arrays

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void exchange(int tab[][4], int w1, int w2) {
5.      int t,k;
6.      for (k = 0; k < 4; k++) {
7.          t           = tab[w1][k];
8.          tab[w1][k] = tab[w2][k];
9.          tab[w2][k] = t;
10.     }
11. }
12.
13. void printArray(int tab[][4], int dim1) {    ❶
14.     int w,k;
15.     for (w = 0; w < dim1; w++) {
16.         for (k = 0; k < 4; k++)
17.             cout << tab[w][k] << " ";
18.         cout << endl;
19.     }
20. }
21.
22. int main() {
23.     int tab[3][4] = { {1,2,3,4}, {5,6}, {1} };
24.
25.     cout << "Array before:\n"; printArray(tab,3);
26.     exchange(tab,0,1);
27.     cout << "Array  after:\n"; printArray(tab,3);
28. }
```

The program prints:

```
Array before:
```

```
1 2 3 4
5 6 0 0
1 0 0 0
Array  after:
5 6 0 0
1 2 3 4
1 0 0 0
```

In the main function we define a two-dimensional array with sizes 3×4. We initialize some of its elements; the rest will be initialized with zeros.

The function **exchange** exchanges two rows with indices passed as the last two arguments of the function. We do not need to understand all the details of the function; what is important is the form in which its first parameter was declared: `int[][4]`. No information on the first dimension of the function is passed. We could have specified it by declaring `int[3][4]`, but this would be just a documenting information for the reader of the code: the compiler would completely ignore it anyway.

However, information on the second dimension is necessary. Otherwise, the offset of elements relative to the beginning of the array could not be calculated inside the function.

The function **printArray** prints the elements of the array passed to it. This function needs to know how many rows to print, so the first dimension of the array must be passed (line ❶). Note, however, that it is passed separately through an additional argument, not as an element of the array's type.

Let us analyze the nature of the variable ***tab*** from the point of view of arrays/pointers correspondence. Suppose we define following definition:

```
int tab[dim1][dim2];
```

We remember from the section on pointer arithmetic that ***tab[i]*** is equivalent to `*(tab+i)`. Therefore, ***tab[i][j]*** will correspond to `*(tab[i]+j)` and should have an integer value, so `tab[i]+j` should be a pointer to an integer. As ***j*** is just an **int**, `tab[i]` should be a pointer to an integer (pointing to the beginning of the row with index ***i***). But this is just `*(tab+i)` and for this to be a pointer, `tab` must be a pointer to pointer to integer, that is something of the type **int****.

There is another problem with our arrays: the function **exchange** in the program **arr2dim.cpp** exchanges two rows with indices passed as arguments. However, it will only work for arrays with 4 columns, because of type declaration of its first parameter. If we had another matrix, with different number of columns, we would have to write another function, although its task would be identical. How could we write a more general function exchanging two given rows of a matrix?

One possibility is to define a function that accepts not a matrix but explicitly a one-dimensional array of pointers to rows, which are themselves one-dimensional arrays of **int** s. The advantage of this approach is the fact that passing an array we do not have to specify the size in its type: the type will be just a pointer. Therefore, we get:

P25: arr2dim2.cpp Matrix as an array of pointers

```
1. #include <iostream>
```

```
2.  using namespace std;
3.
4.  void exchange(int* arr[], int w1, int w2,
5.                            int dim1, int dim2) {
6.      for (int k = 0; k < dim2; k++) {
7.          int t      = arr[w1][k];
8.          arr[w1][k] = arr[w2][k];
9.          arr[w2][k] = t;
10.     }
11. }
12.
13. void printArray(int* arr[], int dim1, int dim2) {
14.     for (int w = 0; w < dim1; w++) {
15.         for (int k = 0; k < dim2; k++)
16.             cout << arr[w][k] << " ";
17.         cout << endl;
18.     }
19. }
20.
21. int main() {
22.     int tt[3][4] = { {1,2,3,4}, {7,6}, {1} };        ❶
23.
24.     const int dim1 = 3;
25.     const int dim2 = 4;
26.
27.     int* arr[dim1];                                  ❷
28.     for (int i = 0; i < dim1; i++) arr[i] = tt[i];
29.
30.     cout << "Before:\n"; printArray(arr,dim1,dim2);
31.     exchange(arr,0,1,dim1,dim2);
32.     cout << "After:\n";  printArray(arr,dim1,dim2);
33. }
```

The first parameter of the function **exchange** has the type *one-dimensional array of pointers to **int** s*. There is no information on any sizes here; sizes are sent to the function separately and can be different for different calls corresponding to matrices of different shapes.

There is, however a problem: given a matrix ***tt*** (line ❶), we have to construct an array of pointers pointing to rows — this is required by the function. Therefore, we have to define additional, auxiliary array ***arr*** of pointers (❷). Its size is equal to number of rows and its elements are assigned the values of addresses of rows of ***tt***, i.e., ***tt[0]***, ***tt[1]*** and ***tt[2]***. Thanks to pointer arithmetic, inside the function we can use ***arr*** as a matrix with two indices (because elements of ***tab*** are not of type **int** but of type **int***). The result

```
Before:
1 2 3 4
7 6 0 0
1 0 0 0
After:
7 6 0 0
1 2 3 4
1 0 0 0
```

shows that both functions, **exchange** and **printArray**, work properly although no information about sizes of arrays have been hard-coded in their definition.

5.5.2 Arrays of C-strings

In order to get used to all these complications, we will consider a special but important example of (one-dimensional) array of C-strings which can be treated as two-dimensional array of characters.

P26: arrstr.cpp Arrays of C-strings

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const char **v;
6.      const char *t[] = {"abcd", "efghi", "jklmno" };  ❶
7.      v = t;                                            ❷
8.      cout << "v+2          = " << v+2          << endl;
9.      cout << "v[2]         = " << v[2]         << endl;
10.     cout << "*(v+2)       = " << *(v+2)       << endl;
11.
12.     cout << "*(*(t+1)+2) = " << *(*(t+1)+2) << endl; ❸
13.     cout << "t[1][2]      = " << t[1][2]      << endl;
14.
15.     cout << "*(*(v+1)+2) = " << *(*(v+1)+2) << endl;
16.     cout << "v[1][2]      = " << v[1][2]      << endl;
17. }
```

What is ***t*** on line ❶? This in an array (because its name is followed by an opening square bracket) of pointers to **const char** s (because it follows type declaration **const char***). The pointers from the table point to arrays of characters (C-strings) initialized with strings given in literal form on the right hand side of the assignment (that is why we used **const** rather than "normal" **char** s). As *'array of elements of type **Type**'* corresponds to type **Type***, our ***t***, being an array of elements of type **const char***, corresponds to type **const char****. This makes the assignment from line ❷ legal.

Next we print the value of `v+2`. This is the address from ***v*** shifted by two lengths of elements of ***v***. Its type is **const char****, not **const char*** — therefore when printing it we get an address as such, not a string of characters.

In the next two lines we print the value of the variable *pointed to* by `v+2`: the type of this variable is **const char***, so a string is printed (all characters starting from the address `*(v+2)`, which is equivalent to ***v[2]***, to the first ***NUL*** character encountered). As ***v[2]*** is the address of the third string, we will see `jklmno` printed.

On lines starting from ❸, we do the same thing on the level of individual characters. The expression `*(t+1)` is equivalent to ***t[1]***, which is the pointer to the first character of the second string. Shifting it by two lengths of one **char**, we get `*(t+1)+2` which will be the address of the third character of the string pointed to by `*(t+1)`; dereferencing it, we get `*(*(t+1)+2)` which is equivalent to ***t[1][2]*** and this is the third character of the second string (the letter 'g'). We can check it looking at the output:

```
v+2          = 0x7fff364d7080
v[2]         = jklmno
*(v+2)       = jklmno
*(*(t+1)+2)  = g
t[1][2]      = g
*(*(v+1)+2)  = g
v[1][2]      = g
```

Last two lines show that all these operations can be performed on ***v*** (which was assigned the value of ***t***), although it has not been explicitly declared to be an array: it is just a pointer of the appropriate type.

Arrays of strings are used extensively in many C/C++ programs and it is very important to

understand what is going on “under the hood”. For example, command-line arguments are passed to the program as such an array — see the introductory section and the program arguments.cpp there.

Although the array of command-line arguments is declared as ' ***char* argv[]*** ' and not ' ***const char* argv[]*** ', we should never try to modify the C-strings pointed to by the elements of ***argv***!

To conclude this section, let us see how to read a C-string from standard input stream (usually connected to the keyboard). We have to remember that all white spaces are treated as separators and not parts of strings. Let us look at an example:

P27: readarr.cpp Reading C-strings

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      char nap1[100], nap2[100];
6.
7.      cout << "\nEnter two strings separated by spaces: ";
8.      cin  >>  nap1 >> nap2;
9.      cout << "\nString 1: " << nap1 << endl;
10.     cout <<   "String 2: " << nap2 << endl << endl;
11. }
```

We create character arrays ***nap1*** and ***nap2*** with sizes sufficient to store expected strings (by the way, it is a simple but generally *not* recommended way of reading strings). Then we read data to this arrays using normal mechanism provided by ***cin***. As we already noticed in section on reading input data , reading one piece of data terminates when a white space is encountered, so we can read two strings as below:

```
Enter two strings separated by a space: 'Linux Windows'

String 1: 'Linux
String 2: Windows'
```

Let us notice that enclosing a string in apostrophes does not help: a space is treated as a separator anyway. We can see also that the mechanism provided by using the object ***cin*** ensures that the terminating character ' \0' will be added automatically. Better ways of reading data from input streams will be covered in chapter on I/O operations .

Subsections

- 5.5.1 Matrices
- 5.5.2 Arrays of C-strings

5.5 Multidimensional arrays

C/C++ supports multidimensional arrays, although their implementation is not as efficient as in other languages (notably Fortran). Actually, any n-dimensional array is in fact a one-dimensional array of pointers which point to $(n - 1)$-dimensional arrays,... and so on recursively. Similarly to “normal” arrays, multidimensional *character* arrays are somewhat special, so we will discuss them separately. Of course, in this chapter we only talk about static arrays, i.e., allocated locally (on the stack) and with dimensions known at compile time.

5.5.1 Matrices

We will focus our attention on two-dimensional arrays (matrices). Let us consider the following definition:

```
int tab[2][4] = { {1,2,3}, {5,6,7,8} };
```

We declared a two-dimensional array (matrix) of integers with 2 rows and 4 columns (i.e., 8 elements). Conventionally, the first index corresponds to rows and the second to columns (what we call a row and what a column is a matter of convention, elements are located in memory linearly anyway).

The matrix ***tab*** is initialized with some initial values. The initializer on the right hand side of the assignment has a form of two arrays corresponding to rows of the matrix. It is important to remember the order in which elements will be stored: in C/C++ the order is row by row (*not* column by column, as in Fortran): the first row (with index 0) goes first

```
tab[0][0]  tab[0][1]  tab[0][2]  tab[0][3]
```

followed by the second row (with index 1)

```
tab[1][0]  tab[1][1]  tab[1][2]  tab[1][3]
```

In the initializer we had to enclose the elements of each row in braces: it was necessary here, because not all elements of the matrix were set. In the first row only three values have been set: the fourth will be assigned the value 0. We could have initialized all elements

```
int tab[2][4] = { 1,2,3,4,5,6,7,8 };
```

without braces indicating rows. This is, however, not very readable and should be avoided.

Additionally, clearly indicating consecutive rows clearly shows the true nature of a matrix as an array of arrays.

The elements of an array can be referenced by indicating row and column indices, each in separate pair of brackets — ***tab[1][2]*** has value 7 in our example; numbering starts from 0.

Suppose we defined a two-dimensional array with ***dim1*** rows and ***dim2*** columns.

```
constexpr int dim1 = ...;
constexpr int dim2 = ...;

int tab[dim1][dim2];
```

We know that elements are stored row by row. Given the address of the first element of the first row and indices ***m*** and ***n***, how to calculate the address of the element ***tab[m][n]***? This element belongs to $(m + 1)$-th row (with index ***m***), so to get to its location we have to jump over the first m rows, each with ***dim2*** elements; therefore we have to "skip" $m \cdot \mathrm{dim2}$ elements. Then we have to jump over the first ***n*** elements of the row that ***tab[m][n]*** belongs to, as it is $(n + 1)$-st element of this row. It follows that in total we have to skip

$$\mathrm{shift} = n + m \cdot \mathrm{dim2}$$

elements. What is important here is the fact that to calculate this offset, we do not need to know what the first dimension of our matrix is (i.e., we do not have to know ***dim1***). Generally, in order to be able to calculate the offset of an element with given indices (relative to the beginning of the array), we need all dimensions except the first. Let us look at an example:

P24: arr2dim.cpp Multidimensional arrays

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void exchange(int tab[][4], int w1, int w2) {
5.      int t,k;
6.      for (k = 0; k < 4; k++) {
7.          t           = tab[w1][k];
8.          tab[w1][k] = tab[w2][k];
9.          tab[w2][k] = t;
10.     }
11. }
12.
13. void printArray(int tab[][4], int dim1) {    ❶
14.     int w,k;
15.     for (w = 0; w < dim1; w++) {
16.         for (k = 0; k < 4; k++)
17.             cout << tab[w][k] << " ";
18.         cout << endl;
19.     }
20. }
21.
22. int main() {
23.     int tab[3][4] = { {1,2,3,4}, {5,6}, {1} };
24.
25.     cout << "Array before:\n"; printArray(tab,3);
26.     exchange(tab,0,1);
27.     cout << "Array  after:\n"; printArray(tab,3);
28. }
```

The program prints:

```
Array before:
```

```
1 2 3 4
5 6 0 0
1 0 0 0
Array  after:
5 6 0 0
1 2 3 4
1 0 0 0
```

In the main function we define a two-dimensional array with sizes 3×4. We initialize some of its elements; the rest will be initialized with zeros.

The function **exchange** exchanges two rows with indices passed as the last two arguments of the function. We do not need to understand all the details of the function; what is important is the form in which its first parameter was declared: `int[][4]`. No information on the first dimension of the function is passed. We could have specified it by declaring `int[3][4]`, but this would be just a documenting information for the reader of the code: the compiler would completely ignore it anyway.

However, information on the second dimension is necessary. Otherwise, the offset of elements relative to the beginning of the array could not be calculated inside the function.

The function **printArray** prints the elements of the array passed to it. This function needs to know how many rows to print, so the first dimension of the array must be passed (line ❶). Note, however, that it is passed separately through an additional argument, not as an element of the array's type.

Let us analyze the nature of the variable ***tab*** from the point of view of arrays/pointers correspondence. Suppose we define following definition:

```
int tab[dim1][dim2];
```

We remember from the section on pointer arithmetic that ***tab[i]*** is equivalent to `*(tab+i)`. Therefore, ***tab[i][j]*** will correspond to `*(tab[i]+j)` and should have an integer value, so `tab[i]+j` should be a pointer to an integer. As ***j*** is just an **int**, `tab[i]` should be a pointer to an integer (pointing to the beginning of the row with index ***i***). But this is just `*(tab+i)` and for this to be a pointer, `tab` must be a pointer to pointer to integer, that is something of the type **int****.

There is another problem with our arrays: the function **exchange** in the program **arr2dim.cpp** exchanges two rows with indices passed as arguments. However, it will only work for arrays with 4 columns, because of type declaration of its first parameter. If we had another matrix, with different number of columns, we would have to write another function, although its task would be identical. How could we write a more general function exchanging two given rows of a matrix?

One possibility is to define a function that accepts not a matrix but explicitly a one-dimensional array of pointers to rows, which are themselves one-dimensional arrays of **int** s. The advantage of this approach is the fact that passing an array we do not have to specify the size in its type: the type will be just a pointer. Therefore, we get:

P25: arr2dim2.cpp Matrix as an array of pointers

```
1. #include <iostream>
```

```
2.  using namespace std;
3.
4.  void exchange(int* arr[], int w1, int w2,
5.                            int dim1, int dim2) {
6.      for (int k = 0; k < dim2; k++) {
7.          int t      = arr[w1][k];
8.          arr[w1][k] = arr[w2][k];
9.          arr[w2][k] = t;
10.     }
11. }
12.
13. void printArray(int* arr[], int dim1, int dim2) {
14.     for (int w = 0; w < dim1; w++) {
15.         for (int k = 0; k < dim2; k++)
16.             cout << arr[w][k] << " ";
17.         cout << endl;
18.     }
19. }
20.
21. int main() {
22.     int tt[3][4] = { {1,2,3,4}, {7,6}, {1} };        ❶
23.
24.     const int dim1 = 3;
25.     const int dim2 = 4;
26.
27.     int* arr[dim1];                                  ❷
28.     for (int i = 0; i < dim1; i++) arr[i] = tt[i];
29.
30.     cout << "Before:\n"; printArray(arr,dim1,dim2);
31.     exchange(arr,0,1,dim1,dim2);
32.     cout << "After:\n";  printArray(arr,dim1,dim2);
33. }
```

The first parameter of the function **exchange** has the type *one-dimensional array of pointers to **int** s*. There is no information on any sizes here; sizes are sent to the function separately and can be different for different calls corresponding to matrices of different shapes.

There is, however a problem: given a matrix ***tt*** (line ❶), we have to construct an array of pointers pointing to rows — this is required by the function. Therefore, we have to define additional, auxiliary array ***arr*** of pointers (❷). Its size is equal to number of rows and its elements are assigned the values of addresses of rows of ***tt***, i.e., ***tt[0]***, ***tt[1]*** and ***tt[2]***. Thanks to pointer arithmetic, inside the function we can use ***arr*** as a matrix with two indices (because elements of ***tab*** are not of type **int** but of type **int***). The result

```
Before:
1 2 3 4
7 6 0 0
1 0 0 0
After:
7 6 0 0
1 2 3 4
1 0 0 0
```

shows that both functions, **exchange** and **printArray**, work properly although no information about sizes of arrays have been hard-coded in their definition.

5.5.2 Arrays of C-strings

In order to get used to all these complications, we will consider a special but important example of (one-dimensional) array of C-strings which can be treated as two-dimensional array of characters.

P26: arrstr.cpp Arrays of C-strings

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const char **v;
6.      const char *t[] = {"abcd", "efghi", "jklmno" };  ❶
7.      v = t;                                            ❷
8.      cout << "v+2          = " << v+2          << endl;
9.      cout << "v[2]         = " << v[2]         << endl;
10.     cout << "*(v+2)       = " << *(v+2)       << endl;
11.
12.     cout << "*(*(t+1)+2) = " << *(*(t+1)+2) << endl; ❸
13.     cout << "t[1][2]      = " << t[1][2]      << endl;
14.
15.     cout << "*(*(v+1)+2) = " << *(*(v+1)+2) << endl;
16.     cout << "v[1][2]      = " << v[1][2]      << endl;
17. }
```

What is ***t*** on line ❶? This in an array (because its name is followed by an opening square bracket) of pointers to **const char** s (because it follows type declaration **const char***). The pointers from the table point to arrays of characters (C-strings) initialized with strings given in literal form on the right hand side of the assignment (that is why we used **const** rather than "normal" **char** s). As *'array of elements of type **Type**'* corresponds to type **Type***, our ***t***, being an array of elements of type **const char***, corresponds to type **const char****. This makes the assignment from line ❷ legal.

Next we print the value of `v+2`. This is the address from ***v*** shifted by two lengths of elements of ***v***. Its type is **const char****, not **const char*** — therefore when printing it we get an address as such, not a string of characters.

In the next two lines we print the value of the variable *pointed to* by `v+2`: the type of this variable is **const char***, so a string is printed (all characters starting from the address `*(v+2)`, which is equivalent to ***v[2]***, to the first ***NUL*** character encountered). As ***v[2]*** is the address of the third string, we will see `jklmno` printed.

On lines starting from ❸, we do the same thing on the level of individual characters. The expression `*(t+1)` is equivalent to ***t[1]***, which is the pointer to the first character of the second string. Shifting it by two lengths of one **char**, we get `*(t+1)+2` which will be the address of the third character of the string pointed to by `*(t+1)`; dereferencing it, we get `*(*(t+1)+2)` which is equivalent to ***t[1][2]*** and this is the third character of the second string (the letter 'g'). We can check it looking at the output:

```
v+2          = 0x7fff364d7080
v[2]         = jklmno
*(v+2)       = jklmno
*(*(t+1)+2)  = g
t[1][2]      = g
*(*(v+1)+2)  = g
v[1][2]      = g
```

Last two lines show that all these operations can be performed on ***v*** (which was assigned the value of ***t***), although it has not been explicitly declared to be an array: it is just a pointer of the appropriate type.

Arrays of strings are used extensively in many C/C++ programs and it is very important to

understand what is going on “under the hood”. For example, command-line arguments are passed to the program as such an array — see the introductory section and the program arguments.cpp there.

Although the array of command-line arguments is declared as ' ***char* argv[]*** ' and not ' ***const char* argv[]*** ', we should never try to modify the C-strings pointed to by the elements of ***argv***!

To conclude this section, let us see how to read a C-string from standard input stream (usually connected to the keyboard). We have to remember that all white spaces are treated as separators and not parts of strings. Let us look at an example:

P27: readarr.cpp Reading C-strings

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      char nap1[100], nap2[100];
6.
7.      cout << "\nEnter two strings separated by spaces: ";
8.      cin  >>  nap1 >> nap2;
9.      cout << "\nString 1: " << nap1 << endl;
10.     cout <<   "String 2: " << nap2 << endl << endl;
11. }
```

We create character arrays ***nap1*** and ***nap2*** with sizes sufficient to store expected strings (by the way, it is a simple but generally *not* recommended way of reading strings). Then we read data to this arrays using normal mechanism provided by ***cin***. As we already noticed in section on reading input data , reading one piece of data terminates when a white space is encountered, so we can read two strings as below:

```
Enter two strings separated by a space: 'Linux Windows'

String 1: 'Linux
String 2: Windows'
```

Let us notice that enclosing a string in apostrophes does not help: a space is treated as a separator anyway. We can see also that the mechanism provided by using the object ***cin*** ensures that the terminating character ' \0' will be added automatically. Better ways of reading data from input streams will be covered in chapter on I/O operations .

5.6 Arrays of type *std::array*

The C++11 standard introduced in its standard library another form of an array: **std::array** (from header `array`). As we will see, its is rather a so called *template* of a class. For now all we have to know is that instead of creating a "normal", C-like array, we can create objects of type **array<Type,size>**, where **Type** is a type of elements and ***size*** is the size of the array. Variables of this type can be used as normal arrays, by indexing. Initialization is also similar; for example

```
1.  #include <iostream>
2.  #include <array>
3.  int main() {
4.      std::array<double,5> a1;
5.      std::array<int,3>    a2{1,2,3};
6.      for (int i = 0; i < 5; ++i) a1[i] = i+0.5;
7.      for (int i = 0; i < 5; ++i) std::cout << a1[i] << " ";
8.      std::cout << std::endl;
9.      for (int i = 0; i < 3; ++i) std::cout << a2[i] << " ";
10.     std::cout << std::endl;
11. }
```

prints

```
0.5 1.5 2.5 3.5 4.5
1 2 3
```

Arrays of this type are superior to normal arrays. First of all, as they are objects, they "know their size", also when passed as argument to a function. The size may be obtained by invoking, on such an object, the method: `arr.size()` (line ❶ in the example below), or by invoking global library function: `std::size(arr)` (line ❸), where ***arr*** is an array. As we can see in the example, this will work even if we pass an array to a function (**printArray** in our case). Individual elements can be accessed by indexing, as we have already seen, or by invoking a method: `arr.at()` (line ❹):

P28: aarrays.cpp Arrays of type *std::array*

```
1.  #include <array>
2.  #include <cstddef>   // size_t
3.  #include <iostream>
4.  #include <string>
5.  using std::array; using std::cout;
6.
7.  void printArray(const array<int,8>& a) {
8.      cout << "Version <int,EIGHT>: "
9.           << "a.size() = " << a.size() << "\n";          ❶
10.     for (const auto& e : a) cout << e << " ";           ❷
11.     cout << "\n";
12. }
13.
14. template <typename E, std::size_t SIZE>
15. void printArray(const array<E,SIZE>& a) {
16.     cout << "Template version: "
17.          << "std::size(a) = " << std::size(a) << "\n"; ❸
18.     for (std::size_t i = 0; i < a.size(); ++i)
19.         cout << a.at(i) << " ";                          ❹
```

```
20.     cout << "\n";
21. }
22.
23. int main() {
24.     array<int,8> ai{1,2,3,4,5};
25.     for (auto& e : ai) ++e;                    ❺
26.     printArray(ai);
27.
28.     array<std::string,5> as{"K","L","M","D"};
29.     for (auto& e : as) e += "!";
30.     printArray(as);
31. }
```

The example illustrates a few elements that we still don't know. The for of loop (for example in lines ❷ and ❺) is characteristic for collections from the standard library — we will cover it in section on loops . The second form of the **printArray** function is really a template — this allows it to be used for arrays of various types of elements and of different sizes. More about it in section on templates .

The program prints

```
Version <int,EIGHT>: a.size() = 8
2 3 4 5 6 1 1 1
Template version: std::size(a) = 5
K! L! M! D! !
```

It may happen that an array of type **std::array** should be passed to a function expecting a normal array (so actually a pointer). We can easily do it by invoking `arr.data()`, what returns the same data as a pointer to “normal” C-array.

Prev: 5.5 Multidimensional arrays **Up:** 5. Static arrays and pointers **Next:** 5.7 Vectors (std::vector)

T.R. Werner, September 27, 2023; 22:32

5.7 Vectors (*std::vector*)

Every C++ programmer must know (and sometimes use...) static arrays as those described above. However, they are not very convenient. Creating them, we have to know their size (and it must already be known when we write the source file, as sizes of C-arrays must be compilation constants). Passing C-arrays to functions is also problematic, because we have to pass their size separately, as what is really passed is not an array itself but just a pointer to its first element. More convenient alternative is provided by **vectors** (**vector** from the `vector` header). This is a collection very similar, and similarly implemented, to **ArrayList** that we know from Java. As type **array**, it is really a template which therefore must be parametrized (in angle brackets) by the type of elements. Vectors, as compared to C-arrays, have several advantages:

- it is possible to create empty vectors and add as many elements as needed;
- it is possible to create vectors with a given initial contents and still add new elements later;
- vectors “know” their size: one can always, for a vector ***vec***, find its current size by invoking `vec.size()`;
- the information about the size is retained when we pass a vector to a function (preferably by reference);
- vectors can even be “shrunk” when some of its elements are not needed any more;
- access to the i-th (counting from zero) element by `vec.at(i)` (but not `vec[i]`) is controlled at run time — illegal values of the index will be signaled (there is no such control when we use C-arrays).

We will cover collections, and among them vectors, later, as for the moment we don't know classes, methods, templates. Nevertheless, the example below (and documentation) should allow us to already use vectors in our programs.

P29: vecsimple.cpp Vectors

```
1.  #include <iostream>
2.  #include <string>
3.  #include <vector>
4.
5.  int main() {
6.      using std::vector; using std::cout;
7.
8.      vector<int> v1{2,1};                          ❶
9.      vector<int> v2(3,1);                          ❷
10.     v1.push_back(0);
11.     v2.push_back(1);
12.     cout << "v1: size = " << v1.size() << " -> ";
13.     for (const auto& e : v1) cout << e << " ";
14.     cout << "\n";
15.     cout << "v2: size = " << v2.size() << " -> ";
16.     for (const auto& e : v2) cout << e << " ";
17.     cout << "\n";
18.
19.     vector<std::string> v3;                       ❸
20.     // v3[0] = "A";  // WRONG!!
```

```
21.     v3.push_back("A");
22.     for (int i = 1; i < 5; ++i)
23.         v3.push_back(v3.at(i-1) + char('A' + i));    ❹
24.     for (const auto& e : v3) cout << e << " ";
25.     cout << "\n";
26.
27.     cout << "First: " << v3.front() << ", last : "    ❺
28.          << v3.back() << "\n";
29.     while (v3.size() > 0) {
30.         cout << "Removing " << v3.back() << "\n";
31.         v3.pop_back();                                ❻
32.     }
33. }
```

The above program prints

```
v1: size = 3 -> 2 1 0
v2: size = 4 -> 1 1 1 1
A AB ABC ABCD ABCDE
First: A, last : ABCDE
Removing ABCDE
Removing ABCD
Removing ABC
Removing AB
Removing A
```

One can create a vector with a given contents using the syntax with curly braces. For example, ***v1***, created in line ❶, contains two numbers of type **int** with values 2 and 1. One can also use round parentheses, as in line ❷. The meaning is, however, different. Now it means: initialize three elements of the vector (the first argument 3) with value 1 (the second argument 1). Finally, one can create an empty vector, as in line ❸. Note that it doesn't contain any elements, therefore the assignment `v3[0]="A"` would be illegal, as `v3[0]` means *the first* element, but there is no elements at all in an empty vector! New elements can be added to a vector by invoking **push_back**, as in the example (or, even better, by **emplace_back**, which we will cover later). New elements are then added – what, of course , modifies the size of the vector – at the end of the collection. There is a way to insert new elements somewhere in the middle, but it should be avoided as rather inefficient. Elements may be accessed by the syntax `vec[i]` (without control of indices) or `vec.at(i)` (❹, with control of validity of the index). The first and the last elements may be also accessed by `vec.front()` and `vec.back()`, respectively (❺). The last element can be removed by invoking `vec.pop_back()` (❻).

It's important to remember that for vectors, as for other collections from the Standard Library, adding an element really means adding its *copy*, while accessing an element means accessing the reference to its `original' located in the collection.

Prev: 5.6 Arrays of type std::array **Up:** 5. Static arrays and pointers **Next:** 6. “Compound” types

6. “Compound” types

The language provides a set of predefined basic data types (like **int**, **double**, etc.). Having at our disposal these types, we can build more complex types in two ways: defining classes or constructing types which are, in a sense, combinations of simple types that we already know. In this chapter we will focus our attention on the latter. We will also show how to use the keywords **typedef** and **using** to make building and using such types simpler.

Subsections

- 6.1 Defining “compound” types
- 6.2 *typedef* and *using* specifiers

6.1 Defining “compound” types

By “compound” types we mean types which are combinations of simpler types, like *array of pointers*, *reference to array*, *reference to pointer to array of function pointers*, etc. Such complicated types can be quite difficult to define and understand even for advanced programmers.

What are the types of variables ***x***, ***y***, ***z***, ***f*** after the following declarations/definitions:

```
int tab[] = {1,2,3};
int (&x)[3] = tab;
int *y[3] = {tab,tab,tab};
int *(&z)[3] = y;
int &(*f)(int*,int&);
```

The first line states that ***tab*** is a *'three-element array of **int** s'*, which in expressions is converted to type **int***. However, the remaining declarations are more difficult to understand. Let us state the general rules of “reading” such expressions:

1. Start from the name of a variable being defined.
2. Look to the right: if you see an opening round parenthesis, then the name denotes a *function*; you have to read the number and type of parameters. If it is an opening square bracket, then the name designates an *array* and you have to read its size.
3. If there is nothing to the right, or there is a closing round parenthesis, look to the left and read elements of the definition from right to left until there is nothing more or you encountered an opening round parenthesis.
4. If you encountered an opening round parenthesis, get out of the parentheses and continue from looking to the right.
5. Read an asterisk ('*') as *is a pointer to*.
6. Read an ampersand ('&') as *is a reference to*.
7. After you have read the number and types of parameters of a function, the rest defines the return type of this function.
8. After you have read the size of an array, the rest defines the type of its elements.

Let us have a closer look at the examples above:

```
int (&x)[3] = tab;
```

x IS

- there is a closing parenthesis to the right, so we look to the left (rule 3) and we see an ampersand — from rule 6 A REFERENCE TO
- further to the left there is a opening parenthesis, so from rule 4 we get out of the parentheses and look to the right: there is an opening bracket there, so from rule 2 we read A THREE-ELEMENT ARRAY OF
- going back to the left and applying rule 8: VARIABLES OF TYPE **int**.

As ***x*** is a reference, it must be initialized; in our case it is initialized with a reference to ***tab***, which is of the appropriate type.

```
int *y[3] = {tab,tab,tab};
```

y IS

- there is an opening bracket to the right, so A THREE-ELEMENT ARRAY OF
- we look to the left — from rules 5 and 8: POINTERS TO VARIABLES OF TYPE **int**.

No initialization is needed here, but the one that was used is correct: the initializer is a three-element array of arrays of **int** s, i.e., after standard conversions, pointers of type **int***. All three elements of ***y*** are initialized with the same value in this case.

```
int *(&z)[3] = y;
```

z IS

- a closing round parenthesis to the right, so we look to the left: A REFERENCE TO
- a round opening parenthesis to the left, so we get out of the parentheses and look to the right, we encounter an opening bracket: A THREE-ELEMENT ARRAY OF
- and now we go back to the left: POINTERS TO VARIABLES OF TYPE **int**.

It is a reference again, so it had to be initialized: the array ***y*** is of correct type and could be used here.

```
int &(*f)(int*,int&);
```

f IS

- a closing round parenthesis to the right, so we look to the left: A POINTER TO
- an opening round parenthesis to the left, so we get out of the parentheses and look to the right: A FUNCTION WITH TWO PARAMETERS OF TYPES **int*** AND **int&** WHICH RETURNS
- we look to the left: A REFERENCE TO VARIABLE OF TYPE **int**.

We will say more about pointers to functions in one of the next sections .

Let us look at these declarations in a program:

P30: decl.cpp "Compound" types

```
1. #include <iostream>
2.
3. int& fun(int *k, int &m) {
4.     return *k > m ? *k : m;
5. }
6.
7. int main() {
8.     using std::cout; using std::endl;
9.     int tab[]{1,2,3};
```

```
10.
11.     int (&x)[3] = tab;
12.     cout << "x[2]     = " << x[2]     << endl;
13.
14.     int *y[3] = {tab,tab,tab};
15.     cout << "y[2][0] = " << y[2][0] << endl;
16.
17.     int *(&z)[3] = y;
18.     cout << "z[2][0] = " << z[2][0] << endl;
19.
20.     int &(*f)(int*,int&);
21.     f = fun;
22.     int v1 =  f(&tab[1], tab[2]);            ❶
23.     int v2 = (*f)(&tab[1], tab[2]);          ❷
24.     cout << "v1      = " << v1      << endl;
25.     cout << "v2      = " << v2      << endl;
26. }
```

In view of the above analysis, the result

```
x[2]     = 3
y[2][0]  = 1
z[2][0]  = 1
v1       = 3
v2       = 3
```

should be clear. Let us notice that both forms of function invocation (lines ❶ and ❷) are equivalent: ***f*** is a pointer to function, but can be used as the name of the function without dereferencing (more in the section on pointers to functions).

Prev: 6. “Compound” types **Up:** 6. “Compound” types **Next:** 6.2 typedef and using specifiers

6.2 *typedef* and *using* specifiers

Complicated type like those that we encountered in the above section are hard to remember and using them can be error prone. However, there is a mechanism which allows us to give them a short, easy to remember name (an alias). This can be achieved with the help of the **typedef** keyword. We do it in the following way:

- we declare a variable of a type that we want to define an alias for;
- we add the keyword **typedef** at the beginning of this declaration. The name of the variable becomes the name of the *type* that this variable would have if there were no the **typedef** keyword.

It is important to remember that we do not “create” a new type in this way; we only give a name (an alias) to an existing (perhaps complicated) type. Creation of new types requires defining classes — we will postpone this issue to next chapters.

For example, after

```
long MY_INT;
```

the variable ***MY_INT*** would be of type **long**. Hence, after

```
typedef long MY_INT;
```

the name **MY_INT** is an alias of **long**, and declaration

```
MY_INT k;
```

is equivalent (at least in the scope where the **typedef** declaration is visible) to the declaration

```
long k;
```

Usually, we use **typedef** specifier in more complicated cases, quite often requiring nested **typedef** s (what is legal). In the following program

- **CH1** is an alias for the type *three-element array of characters*;
- **CH2** stands for *two-element array of three-element arrays of characters*;
- **CH3** is the name of the type *two-element array of two-element arrays of three-element arrays of characters*.

P31: typedef.cpp *typedef specifier*

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     typedef char CH1[3];
6.     typedef CH1  CH2[2];
7.     typedef CH2  CH3[2];
8.
```

```
    CH1 ch1 = {'a','b','c'};

    CH2 ch2 = {{'a','b','c'},{'d','e','f'}};

    CH3 ch3 = {
                  {{'a','b','c'},{'d','e','f'}},
                  {{'g','h','i'},{'j','k','l'}},
              };

    cout << "sizeof(CH1)  = " << sizeof(CH1)  << endl
         << "sizeof(CH2)  = " << sizeof(CH2)  << endl
         << "sizeof(CH3)  = " << sizeof(CH3)  << endl
         << "ch1[2]       = " << ch1[2]       << endl
         << "ch2[1][1]    = " << ch2[1][1]    << endl
         << "ch3[1][0][2] = " << ch3[1][0][2] << endl;
}
```

The type named **CH3** is essentially equivalent to *three-dimensional array of characters with dimensions* 2×2×3. The program prints

```
sizeof(CH1)  = 3
sizeof(CH2)  = 6
sizeof(CH3)  = 12
ch1[2]       = c
ch2[1][1]    = e
ch3[1][0][2] = i
```

Notice that sizes printed by the program indicate that declared types have been correctly interpreted.

The **typedef** specifier is often used to give a name to types that are used in a program as types of parameters or return types of functions, especially if a type is complicated and appears in many places of the program. As a (not so complicated) example let us consider the following program:

P32: typedef1.cpp *typedef specifiers and functions*

```
#include <iostream>
using namespace std;

typedef int IN3[][2][2];

int fun(IN3 t) {
    int max = t[0][0][0];
    for (int k = 0; k < 2; ++k)
        for (int j = 0; j < 2; ++j)
            for (int i = 0; i < 2; ++i)
                if (t[k][j][i] > max)
                    max = t[k][j][i];
    return max;
}

int main() {
    IN3 in3 = { {{4,3},{2,1}},
                {{7,8},{5,6}} };

    int max = fun(in3);

    cout << "max = " << max << endl;
}
```

The type **IN3** stands for *three-dimensional array of **int** s with the second and third dimensions equal to 2*. Having established this, we do not need to repeat the detailed specification of this type when using it in a function definition or declaration of the variable ***in3***. The function **fun** finds the maximum element of a three-dimensional array passed as the argument — the program prints '`max = 8`'.

Starting from the C++11 standard a new, probably more readable, syntax can be used in lieu of **typedef**. We begin with the keyword **using**, after which we write a sort of an assignment: the name of an alias on the left and specification of the type on the right. On the right hand side the name of a “variable” is omitted. For example, the following pairs of alias definitions are equivalent:

```
typedef int TAB[3];
using TAB = int[3];

typedef int& (*FUN)(int&,double);
using FUN = int& (*)(int&,double);

typedef double* (&T)[3]
using T =  double* (&)[3]
```

As we will see later, when talking about templates, this form is more flexible, because it allows us to define an alias for a family of types (aliases may be parametrized by types).

Prev: 6.1 Defining “compound” types **Up:** 6. “Compound” types **Next:** 7. Variables

7. Variables

This chapter will be devoted to variables: how to declare and define them, what is the meaning of various modifiers used when declaring variables. The notion of scope, visibility and lvalue will also be explained.

Subsections

- 7.1 Keywords and names
- 7.2 Scope and visibility
- 7.3 Storage classes
  - 7.3.1 Static variables
  - 7.3.2 External variables
- 7.4 Type modifiers
  - 7.4.1 Volatile variables
  - 7.4.2 Constants
- 7.5 L-values

7.1 Keywords and names

Names are introduced into a translation unit (i.e., a file together with other files added by `#include`) by **declarations**. Declarations specify the interpretation and attributes of these names. Declarations are often (but not always, as we will see) also definitions, i.e., loosely speaking, they are connected with creation of the named objects physically in the computer's memory. The following *"one definition rule"* must be observed:

In any compilation unit there can be at most one definition of any variable, function, class, enumeration type or template.

However, this restriction does not apply to *declarations*, which *may* be repeated.

When introducing new names (**identifiers**) into our programs, we have to remember that there are some words, called **keywords**, that are reserved and cannot be used for other purposes (e.g., as names of variables, functions, classes, etc.). All the 92 keywords of the C++ programming language are summarized in the table below.

alignas	const_cast	module	static_cast
alignof	continue	mutable	struct
and	decltype	namespace	switch
and_eq	default	new	synchronized
asm	delete	noexcept	template
atomic_cancel	do	not	this
atomic_commit	double	not_eq	thread_local
atomic_noexcept	dynamic_cast	nullptr	trrow
auto	else	operator	true
bitand	enum	or	try
bitor	explicit	or_eq	typedef
bool	export	private	typeid
break	extern	protected	typename
case	false	public	union
catch	float	register	unsigned
char	for	reinterpret_cast	using
char16_t	friend	requires	virtual
char32_t	goto	return	void
class	if	short	volatile
compl	import	signed	wchar_t
concept	inline	sizeof	while
const	int	static	xor
constexpr	long	static_assert	xor_eq

Table 7.1: C++ keywords

Additionally, four identifiers *are* keywords, but only when used in specific contexts; these

are **override**, **final**, **transaction_safe** and **transaction_safe_dynamic**.

All names in a program must be different than any of the keywords. They can be composed of letters, digits and underscores ('_'). Formally, non-Latin Unicode characters are allowed, but using them may cause some problems, so it is better to avoid them. The first character must not be a digit. Also, names containing anywhere two consecutive underscores, or an underscore followed by an uppercase letter at the beginning are reserved and should not be used. Finally, names beginning with an underscore not followed by an uppercase letter may be used only in global scope.

Lower- and upper-case letters are *distinct*: the names ***val_X*** and ***val_x*** are both legal but different names and do not have to be related in any way.

There are no strict naming conventions in C++, although most programmers use lower-case identifiers to denote variables and functions; names of classes start with an upper-case letter, and constants are traditionally written in upper case.

If a name consists of several words, they are usually separated by underscore (***customer_id***), or each word (except, perhaps, the first) starts with an upper case letter (***fileName***) — this is called CamelCase notation.

Prev: 7. Variables **Up:** 7. Variables **Next:** 7.2 Scope and visibility

7.2 Scope and visibility

Every identifier declared in a program has its **scope**. This is the portion of the program where this declaration is active, i.e., the name declared can be used. The identifier is **visible** where it is associated by the compiler with this declaration without any qualification, i.e., the name alone can be used without specifying class name, namespace name, etc. These two notions are not synonymic: a name can be hidden (shadowed) by declaring another entity with the same name (see below).

In C/C++ it is possible to declare variables outside of all functions (and classes). They have the **file scope**, i.e., their declaration is active from the point where they appear in the source file to the end of this file (and including all functions which are defined in this portion of the program). Their scope can even be extended to other files (modules) where they are redeclared as **extern** (see section on external variables). We then say that they are **exported**. Variables which are defined outside of functions and classes are called **global**. Unlike local variables, defined inside a function (generally: a block), they *are* automatically initialized with 0 (zero) of the appropriate type (this applies also to pointers). They are alive (exist in memory) until the termination of the program.

Local variables are defined inside blocks delimited by braces; in particular in the body of functions. Their scope starts from their declaration/definition and extends to the end of the narrowest block containing this definition, i.e., at most to the end of the function they are defined in. Therefore, variables with the same name but declared in different functions are completely independent. Parameters of functions can be treated as defined locally at the very beginning of the body of the function: they are initialized with the values of arguments used when invoking the function. Local variables are alive from the moment they are defined until the end of their scope (the narrowest surrounding block). When the flow of control exits the block, local variables are removed from the memory. This means, in particular, that when the program reenters the same function (or, more generally, a block), all local variables are created afresh — they do not “remember” their previous values. Variables declared in the initialization part of a **for**-loop and condition part of a **if**, **while**, **for** and **switch** statements are local to the body of these statements: they disappear when the loop terminates (the details will be explained when we come to loops).

Global variables can be hidden (shadowed) by declaring another entity with the same name locally in a block (e.g., a function) contained in this variable's scope. Inside this block, starting from the point of declaration, the unqualified name will refer to the *local* variable; we say that the local variable hides (shadows) the global one. The global variable still exists and is accessible, but we have to qualify its name with a special **scope resolution operator** which is denoted by a double colon, i.e., by two consecutive colons '`::`'. If a global variable ***k*** has been shadowed in a block by another variable with the same name, then it still can be accessed by its qualified name ***::k***.

The program below illustrates visibilty of variables and hiding.

P33: hide.cpp Visibility and hiding variables

```
1.  #include <iostream>
2.
3.  int k;                                  ❶
4.
5.  int main() {
6.      using std::cout; using std::endl;
7.
8.      cout << "  k: " <<   k << endl;     ❷
9.      cout << "::k: " << ::k << endl;
10.
11.     int k = 10;                         ❸
12.
13.     cout << "  k: " <<   k << endl;
14.     cout << "::k: " << ::k << endl;
15.
16.     ::k = 1;                            ❹
17.
18.     cout << "  k: " <<   k << endl;
19.     cout << "::k: " << ::k << endl;
20.
21.     {                                   ❺
22.         int k = 77;
23.         cout << "Inside block:" << endl;
24.         cout << "  k: " <<    k << endl;
25.         cout << "::k: " <<  ::k << endl;
26.     }
27.
28.     cout << "After  block:" << endl;
29.     cout << "  k: " <<   k  << endl;
30.     cout << "::k: " << ::k  << endl;
31. }
```

The variable ***k*** declared on line ❶ is a global variable. Therefore, it is also visible in function **main**. As it is a global variable, it *is* initialized with zero, what we can see printing its value in lines ❷. As it was not hidden, both forms, unqualified, ***k***, and qualified, ***::k***, refer to the same variable — the global one.

On line ❸ we introduce a *local* variable, ***k***. It shadows the global ***k***, so the name ***k*** refers now to this local variable. The global ***k*** is still accessible, but its name must now be qualified with the scope resolution operator; we refer to it by the name ***::k***. On line ❹, we change the value of *global* ***k*** — the value of the local ***k*** remains intact.

On line ❺, we create a block, which is therefore nested in the block constituting the body of the function **main**. Inside this block we define another variable with the name ***k*** (what would be impossible in Java). Inside the block we can access this newly created ***k*** (with the value 77) and the global ***::k*** - — the ***k*** declared on line ❸ is now completely shadowed: there is no way we could refer to it. However, when we leave the block, the ***k*** declared in this block is removed, so again the name ***k*** refers to the variable declared in line ❸. The program prints

```
  k: 0
::k: 0
  k: 10
::k: 0
  k: 10
::k: 1
Inside block:
  k: 77
::k: 1
After  block:
  k: 10
::k: 1
```

Subsections

- 7.3.1 Static variables
- 7.3.2 External variables

7.3 Storage classes

The storage class of a variable can be specified by one of the specifiers **extern** or **static** (the third one, **register**, is obsolete, although it is still a keyword). Generally, it specifies how and where a variable will be located in memory.

7.3.1 Static variables

Static variables are declared with specifier **static**. Both global and local variables can be declared as static.

Static variables are created only once and initialized with zero of an appropriate type. If it is declared locally, it is created when the flow of control encounters the declaration for the first time. It will exist until termination of the program. It follows, that this is a kind of data which persists between function calls: when the flow of control reenters a function, the values of static variables declared in this function will have the values that they had when the function was called previously, as physically they are still the same variables. They exist between function calls, but, as they are declared locally in a function, they are not visible outside of these functions.

Global variables can also be declared as **static**. They are also created only once (before calling **main**) and live until termination of the program. What is then the difference between "normal" and static global variables? If a global variable is static, it is not exported to the linker, so it will be "invisible" in other compilation units (what normally is not the case: we will explain it when we come to **extern** keyword). However, using global static variables is generally not recommended — instead, one can use the mechanism of namespaces (see section on namespaces).

The names of static variables can be hidden, both global and local ones. All such variables retain their identity when the program reenters the scope they are declared in. Let us look at an example:

P34: stat.cpp Static variables

```
1. #include <iostream>
2. using namespace std;
3.
4. int stat = 10;
5.
```

```
6. void fun() {
7.     static int stat;
8.     cout << "local  stat " << stat++ << endl;
9.     cout << "global stat " << ::stat << endl;
10.    {                                          ❶
11.        static int stat;
12.        cout << "block  stat " << stat--
13.                               << "\n\n";
14.    }
15. }
16.
17. int main() {
18.     fun();
19.     fun();
20.     fun();
21. }
```

The program prints:

```
local  stat 0
global stat 10
block  stat 0

local  stat 1
global stat 10
block  stat -1

local  stat 2
global stat 10
block  stat -2
```

As we can see, all three variables ***stat*** are independent and “persistent”. In particular, both local variables ***stat*** declared in the function retain independently their values between calls, also the one inside a somewhat artificial block starting in line ❶.

We can use static variables with the same name in different functions: they are of course completely unrelated. There are three static variables ***counter*** in the program below — two in two different functions and one global. All three are initialized with zero automatically. As the local counters retain their values between calls, they can be used to count how many times the corresponding function was called. In both functions the global variable ***::counter*** is visible (by its qualified name, as it is hidden by local variables of the same name), so it is used to count calls to both **fun1** and **fun2**:

P35: static.cpp Static call counters

```
1. #include <iostream>
2. using namespace std;
3.
4. int counter;
5.
6. void fun1() {
7.     static int counter;
8.     counter++;     // local
9.     ::counter++;   // global
10.    cout << "Call count   fun1: " << counter << endl;
11. }
12.
13. void fun2() {
14.     static int counter;
15.     counter++;     // local
16.     ::counter++;   // global
17.     cout << "Call count   fun2: " << counter << endl;
18. }
19.
```

```
20. int main() {
21.     fun1(); fun1(); fun2(); fun1(); fun2();
22.     cout << "Call count fun1/2: " << counter << endl;
23. }
```

The program prints:

```
Call count   fun1: 1
Call count   fun1: 2
Call count   fun2: 1
Call count   fun1: 3
Call count   fun2: 2
Call count fun1/2: 5
```

7.3.2 External variables

External variables are declared with the **extern** specifier,

```
extern double x;
```

They can be declared only in global (file) scope, i.e., outside of any functions. Declaring a variable as external means that the variable of such name is (or will be) *defined* in another compilation unit. In other words, this is a message for the compiler that we do not want to create any variable: it will be created elsewhere and here, in this module, we only want to have access to it. Therefore, the compiler does not try to allocate memory for this variable; it is the task for the linker to provide the address of a variable of this name from another module (note that this is a rare case when a variable is declared but not defined). External variables can be declared many times in several modules. However, they can be defined only once in exactly one compilation unit of the whole program. The definition, and possibly explicit initialization, must have the form of a declaration *without* the **extern** or **static** specifier. It should be obvious that when a variable is declared as external, it cannot be initialized — this is only declaration, no memory is allocated, so there is no place to write the initial value to. Therefore, a line like this

```
extern double x = 1.5;
```

is equivalent to

```
double x = 1.5;
```

as initialization forces the compiler to allocate memory and hence *define* the variable. In such a case the keyword **extern** will be simply ignored.

All this pertained to external *variables*; the same keyword **extern** may also be used in declaration of functions defined in another module. However, in function declarations it is superfluous, as functions are linked externally anyway (if they were not declared **static**). More about this in section on functions .

Consider a program which consists of two files. The first contains the **main** function

P36: exter1.cpp External variables. File 1

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. double x1 = 11;
5. extern double x2;
6. void func();
7.
8. int main()
9. {
10.     cout << "main: x1 = " << x1 << endl;
11.     cout << "main: x2 = " << x2 << endl;
12.     func();
13. }
```

and the second a function **func**

P37: exter2.cpp External variables. File 2

```
1. #include <iostream>
2. using namespace std;
3.
4. extern double x1;
5. double x2 = 22;
6.
7. void func()
8. {
9.     cout << "func: x1 = " << x1 << endl;
10.     cout << "func: x2 = " << x2 << endl;
11. }
```

The variable ***x1*** is declared *and defined* in the first file as a global variable. In the second file it is declared as **extern** but not defined: therefore, the name ***x1*** in the second file will refer to ***x1*** defined in the first one. With ***x2*** the situation is just opposite. The program prints

```
cpp> g++ -o extern exter1.cpp exter2.cpp
cpp> ./extern
main: x1 = 11
main: x2 = 22
func: x1 = 11
func: x2 = 22
```

After the program has been linked, the names ***x1*** and ***x2*** from the two files refer to the same two variables. The function **func** was not declared as external in the first file (although it could have been), because, as we have said, nonstatic functions are linked externally anyway.

Subsections

- 7.3.1 Static variables
- 7.3.2 External variables

7.3 Storage classes

The storage class of a variable can be specified by one of the specifiers **extern** or **static** (the third one, **register**, is obsolete, although it is still a keyword). Generally, it specifies how and where a variable will be located in memory.

7.3.1 Static variables

Static variables are declared with specifier **static**. Both global and local variables can be declared as static.

Static variables are created only once and initialized with zero of an appropriate type. If it is declared locally, it is created when the flow of control encounters the declaration for the first time. It will exist until termination of the program. It follows, that this is a kind of data which persists between function calls: when the flow of control reenters a function, the values of static variables declared in this function will have the values that they had when the function was called previously, as physically they are still the same variables. They exist between function calls, but, as they are declared locally in a function, they are not visible outside of these functions.

Global variables can also be declared as **static**. They are also created only once (before calling **main**) and live until termination of the program. What is then the difference between "normal" and static global variables? If a global variable is static, it is not exported to the linker, so it will be "invisible" in other compilation units (what normally is not the case: we will explain it when we come to **extern** keyword). However, using global static variables is generally not recommended — instead, one can use the mechanism of namespaces (see section on namespaces).

The names of static variables can be hidden, both global and local ones. All such variables retain their identity when the program reenters the scope they are declared in. Let us look at an example:

P34: stat.cpp Static variables

```
1. #include <iostream>
2. using namespace std;
3.
4. int stat = 10;
5.
```

```
6. void fun() {
7.     static int stat;
8.     cout << "local  stat " << stat++ << endl;
9.     cout << "global stat " << ::stat << endl;
10.    {                                          ❶
11.        static int stat;
12.        cout << "block  stat " << stat--
13.                               << "\n\n";
14.    }
15. }
16.
17. int main() {
18.     fun();
19.     fun();
20.     fun();
21. }
```

The program prints:

```
local  stat 0
global stat 10
block  stat 0

local  stat 1
global stat 10
block  stat -1

local  stat 2
global stat 10
block  stat -2
```

As we can see, all three variables ***stat*** are independent and “persistent”. In particular, both local variables ***stat*** declared in the function retain independently their values between calls, also the one inside a somewhat artificial block starting in line ❶.

We can use static variables with the same name in different functions: they are of course completely unrelated. There are three static variables ***counter*** in the program below — two in two different functions and one global. All three are initialized with zero automatically. As the local counters retain their values between calls, they can be used to count how many times the corresponding function was called. In both functions the global variable ***::counter*** is visible (by its qualified name, as it is hidden by local variables of the same name), so it is used to count calls to both **fun1** and **fun2**:

P35: static.cpp Static call counters

```
1. #include <iostream>
2. using namespace std;
3.
4. int counter;
5.
6. void fun1() {
7.     static int counter;
8.     counter++;     // local
9.     ::counter++;   // global
10.    cout << "Call count   fun1: " << counter << endl;
11. }
12.
13. void fun2() {
14.     static int counter;
15.     counter++;     // local
16.     ::counter++;   // global
17.     cout << "Call count   fun2: " << counter << endl;
18. }
19.
```

```
20. int main() {
21.     fun1(); fun1(); fun2(); fun1(); fun2();
22.     cout << "Call count fun1/2: " << counter << endl;
23. }
```

The program prints:

```
Call count   fun1: 1
Call count   fun1: 2
Call count   fun2: 1
Call count   fun1: 3
Call count   fun2: 2
Call count fun1/2: 5
```

7.3.2 External variables

External variables are declared with the **extern** specifier,

```
extern double x;
```

They can be declared only in global (file) scope, i.e., outside of any functions. Declaring a variable as external means that the variable of such name is (or will be) *defined* in another compilation unit. In other words, this is a message for the compiler that we do not want to create any variable: it will be created elsewhere and here, in this module, we only want to have access to it. Therefore, the compiler does not try to allocate memory for this variable; it is the task for the linker to provide the address of a variable of this name from another module (note that this is a rare case when a variable is declared but not defined). External variables can be declared many times in several modules. However, they can be defined only once in exactly one compilation unit of the whole program. The definition, and possibly explicit initialization, must have the form of a declaration *without* the **extern** or **static** specifier. It should be obvious that when a variable is declared as external, it cannot be initialized — this is only declaration, no memory is allocated, so there is no place to write the initial value to. Therefore, a line like this

```
extern double x = 1.5;
```

is equivalent to

```
double x = 1.5;
```

as initialization forces the compiler to allocate memory and hence *define* the variable. In such a case the keyword **extern** will be simply ignored.

All this pertained to external *variables*; the same keyword **extern** may also be used in declaration of functions defined in another module. However, in function declarations it is superfluous, as functions are linked externally anyway (if they were not declared **static**). More about this in section on functions .

Consider a program which consists of two files. The first contains the **main** function

P36: exter1.cpp External variables. File 1

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. double x1 = 11;
5. extern double x2;
6. void func();
7.
8. int main()
9. {
10.     cout << "main: x1 = " << x1 << endl;
11.     cout << "main: x2 = " << x2 << endl;
12.     func();
13. }
```

and the second a function **func**

P37: exter2.cpp External variables. File 2

```
1. #include <iostream>
2. using namespace std;
3.
4. extern double x1;
5. double x2 = 22;
6.
7. void func()
8. {
9.     cout << "func: x1 = " << x1 << endl;
10.     cout << "func: x2 = " << x2 << endl;
11. }
```

The variable ***x1*** is declared *and defined* in the first file as a global variable. In the second file it is declared as **extern** but not defined: therefore, the name ***x1*** in the second file will refer to ***x1*** defined in the first one. With ***x2*** the situation is just opposite. The program prints

```
cpp> g++ -o extern exter1.cpp exter2.cpp
cpp> ./extern
main: x1 = 11
main: x2 = 22
func: x1 = 11
func: x2 = 22
```

After the program has been linked, the names ***x1*** and ***x2*** from the two files refer to the same two variables. The function **func** was not declared as external in the first file (although it could have been), because, as we have said, nonstatic functions are linked externally anyway.

Subsections

- 7.3.1 Static variables
- 7.3.2 External variables

7.3 Storage classes

The storage class of a variable can be specified by one of the specifiers **extern** or **static** (the third one, **register**, is obsolete, although it is still a keyword). Generally, it specifies how and where a variable will be located in memory.

7.3.1 Static variables

Static variables are declared with specifier **static**. Both global and local variables can be declared as static.

Static variables are created only once and initialized with zero of an appropriate type. If it is declared locally, it is created when the flow of control encounters the declaration for the first time. It will exist until termination of the program. It follows, that this is a kind of data which persists between function calls: when the flow of control reenters a function, the values of static variables declared in this function will have the values that they had when the function was called previously, as physically they are still the same variables. They exist between function calls, but, as they are declared locally in a function, they are not visible outside of these functions.

Global variables can also be declared as **static**. They are also created only once (before calling **main**) and live until termination of the program. What is then the difference between "normal" and static global variables? If a global variable is static, it is not exported to the linker, so it will be "invisible" in other compilation units (what normally is not the case: we will explain it when we come to **extern** keyword). However, using global static variables is generally not recommended — instead, one can use the mechanism of namespaces (see section on namespaces).

The names of static variables can be hidden, both global and local ones. All such variables retain their identity when the program reenters the scope they are declared in. Let us look at an example:

P34: stat.cpp Static variables

```
1. #include <iostream>
2. using namespace std;
3.
4. int stat = 10;
5.
```

```
6. void fun() {
7.     static int stat;
8.     cout << "local  stat " << stat++ << endl;
9.     cout << "global stat " << ::stat << endl;
10.    {                                          ❶
11.        static int stat;
12.        cout << "block  stat " << stat--
13.                               << "\n\n";
14.    }
15. }
16.
17. int main() {
18.     fun();
19.     fun();
20.     fun();
21. }
```

The program prints:

```
local  stat 0
global stat 10
block  stat 0

local  stat 1
global stat 10
block  stat -1

local  stat 2
global stat 10
block  stat -2
```

As we can see, all three variables ***stat*** are independent and “persistent”. In particular, both local variables ***stat*** declared in the function retain independently their values between calls, also the one inside a somewhat artificial block starting in line ❶.

We can use static variables with the same name in different functions: they are of course completely unrelated. There are three static variables ***counter*** in the program below — two in two different functions and one global. All three are initialized with zero automatically. As the local counters retain their values between calls, they can be used to count how many times the corresponding function was called. In both functions the global variable ***::counter*** is visible (by its qualified name, as it is hidden by local variables of the same name), so it is used to count calls to both **fun1** and **fun2**:

P35: static.cpp Static call counters

```
1. #include <iostream>
2. using namespace std;
3.
4. int counter;
5.
6. void fun1() {
7.     static int counter;
8.     counter++;     // local
9.     ::counter++;   // global
10.    cout << "Call count   fun1: " << counter << endl;
11. }
12.
13. void fun2() {
14.     static int counter;
15.     counter++;     // local
16.     ::counter++;   // global
17.     cout << "Call count   fun2: " << counter << endl;
18. }
19.
```

```
20. int main() {
21.     fun1(); fun1(); fun2(); fun1(); fun2();
22.     cout << "Call count fun1/2: " << counter << endl;
23. }
```

The program prints:

```
Call count   fun1: 1
Call count   fun1: 2
Call count   fun2: 1
Call count   fun1: 3
Call count   fun2: 2
Call count fun1/2: 5
```

7.3.2 External variables

External variables are declared with the **extern** specifier,

```
extern double x;
```

They can be declared only in global (file) scope, i.e., outside of any functions. Declaring a variable as external means that the variable of such name is (or will be) *defined* in another compilation unit. In other words, this is a message for the compiler that we do not want to create any variable: it will be created elsewhere and here, in this module, we only want to have access to it. Therefore, the compiler does not try to allocate memory for this variable; it is the task for the linker to provide the address of a variable of this name from another module (note that this is a rare case when a variable is declared but not defined). External variables can be declared many times in several modules. However, they can be defined only once in exactly one compilation unit of the whole program. The definition, and possibly explicit initialization, must have the form of a declaration *without* the **extern** or **static** specifier. It should be obvious that when a variable is declared as external, it cannot be initialized — this is only declaration, no memory is allocated, so there is no place to write the initial value to. Therefore, a line like this

```
extern double x = 1.5;
```

is equivalent to

```
double x = 1.5;
```

as initialization forces the compiler to allocate memory and hence *define* the variable. In such a case the keyword **extern** will be simply ignored.

All this pertained to external *variables*; the same keyword **extern** may also be used in declaration of functions defined in another module. However, in function declarations it is superfluous, as functions are linked externally anyway (if they were not declared **static**). More about this in section on functions .

Consider a program which consists of two files. The first contains the **main** function

P36: exter1.cpp External variables. File 1

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. double x1 = 11;
5. extern double x2;
6. void func();
7.
8. int main()
9. {
10.     cout << "main: x1 = " << x1 << endl;
11.     cout << "main: x2 = " << x2 << endl;
12.     func();
13. }
```

and the second a function **func**

P37: exter2.cpp External variables. File 2

```
1. #include <iostream>
2. using namespace std;
3.
4. extern double x1;
5. double x2 = 22;
6.
7. void func()
8. {
9.     cout << "func: x1 = " << x1 << endl;
10.     cout << "func: x2 = " << x2 << endl;
11. }
```

The variable ***x1*** is declared *and defined* in the first file as a global variable. In the second file it is declared as **extern** but not defined: therefore, the name ***x1*** in the second file will refer to ***x1*** defined in the first one. With ***x2*** the situation is just opposite. The program prints

```
cpp> g++ -o extern exter1.cpp exter2.cpp
cpp> ./extern
main: x1 = 11
main: x2 = 22
func: x1 = 11
func: x2 = 22
```

After the program has been linked, the names ***x1*** and ***x2*** from the two files refer to the same two variables. The function **func** was not declared as external in the first file (although it could have been), because, as we have said, nonstatic functions are linked externally anyway.

Subsections

- 7.4.1 Volatile variables
- 7.4.2 Constants

7.4 Type modifiers

Modifiers **volatile** and **const** are called *type* modifiers because they change the type of objects being declared/defined (although their binary representation remains unchanged).

7.4.1 Volatile variables

Volatile variables are declared with modifier **volatile**, e.g.,

```
volatile double x;
```

where **volatile double** is the name of a type which is different than just **double**. Declaring a variable as volatile means that its value can be modified in such a way that the program "does not know" about it. This can happen if, for example, the value can be changed by some external sensors. Volatile variables are treated in a special way: the compiler must ensure that their value stored in memory is always valid and current; all modification have to be executed immediately without buffering new values in registers or cache. Every time the value of a volatile variable is needed, it must be read from memory and not taken from registers or cache, because it is potentially possible that it was changed even if the compiler does not see any "reason" for such a change.

Volatile variables are never needed in "normal" programs and one has to remember that their use deteriorates efficiency of the program and makes the process of its optimization much harder.

Also, it is not true that volatile variables are useful in multi-threaded applications, as the details of their behavior can depend on implementation and are therefore not portable (in such situations, use rather **atomic** variables).

7.4.2 Constants

Constants are used very often and in many contexts. They are declared with modifier **const** (before or after the name of a type). They have to be initialized at the moment of creation; after that their value cannot be changed.

The value of constants cannot be changed after their creation.

For example, we can define a constant representing π as

```
const double PI = 3.1415926536;
```

but the construction like

```
const double PI;
PI = 3.1415926536; // WRONG !!!
```

would be illegal because in the second line we try to change the value of a constant that has already been created.

Some constant can be evaluated at compilation time (if the initializers contain literal values or other constants that can be evaluated at compile time — simple arithmetic on integral values is also tractable. More complex initializations will be performed at run time.

```
const int a = 2*11;
const int b = 5*a*a;
const int c = fun(a,b);
```

Variables ***a*** and ***b*** are compilation constants, but ***c*** is not, although its type is **const int**, because it will be initialized at run time, as it involves invoking a function.

In the example above, all variables are of type **const int** which is different than just **int**. For that reason, the following

```
const int i = 25;
int *pi = &i; // WRONG !!!
```

is illegal, as the type of '`&i`' is *pointer to constant integer* while the declared type of ***pi*** is *pointer to integer*.

However, an assignment in the opposite direction would be legal. If a pointer is declared as a pointer to a constant, it can be assigned the value of the address of a non- **const** variable. After that, this value cannot be changed if we refer to the variable using the name of the pointer: it can be changed, however, if we can refer to the same variable by another name. Let us look at an example:

P38: constants.cpp Constants

```
1. #include <iostream>
2. using namespace std;
3.
4. int fun(const int * pi) {
5.     //*pi = 2 * (*pi); // WRONG !!!
6.     return *pi;
7. }
8.
9. int main() {
10.     int i = 2;
11.     int res = fun(&i);                ❶
12.     cout << "res = " << res << endl;
13. }
```

The line commented out in the function would be illegal. We pass to the function the address of a “normal” integer (❶). However, the parameter of the function is declared as a *pointer to constant integer*, so the compiler will not allow to change the value of the variable pointed to by pointer ***pi*** (which is just non-const integer ***i*** from the **main** function).

Of course the same variable is modifiable in **main** because here it has type **int** (and not **const int**).

This behavior may lead to some confusion. For example, in the following program

P39: constants1.cpp More on constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int i = 2;
6.      const int *pi = &i;                ❶
7.
8.      i = 2*i;                           ❷
9.      cout << "  i = " <<   i << endl;
10.     cout << "*pi = " << *pi << endl;
11.     //*pi = -1;
12. }
```

variable ***i*** is not declared as a constant. But ***pi*** is a pointer of type **const int***, which means that the variable pointed to by ***pi*** should be treated as a constant. We initialize ***pi*** with the address of ***i*** (line ❶). Is ***i*** constant or not? The answer is:

- yes, if we refer to it through the name ***pi*** (by dereferencing it: ***pi**) — that is why the commented out last line would be illegal;
- no, if we refer to it by name ***i*** (as can be seen on line ❷, which is legal).

With the last line commented out, the program is correct and prints

```
  i = 4
*pi = 4
```

One has to clearly distinguish between pointers to constants and constant pointers. In all the above examples, pointers pointed to constants, but were not constant themselves. Variables pointed to by these pointers could not be modified, but it was possible to change the value of pointers, i.e., the addresses stored in them. Let us see one more example

P40: constants2.cpp Pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const int i = 2, j = 3;
6.
7.      const int *p = &i;                 ❶
8.      cout << "*p = " << *p << endl;
9.
10.     p = &j;
11.     cout << "*p = " << *p << endl;
12. }
```

Both variables, ***i*** and ***j*** are declared constant. The pointer ***p*** is a pointer to constant. But itself, it is not a constant: we initialize it with the address of ***i*** (❶), but then we change its value by assigning to it the address of ***j***. Note, that it was not necessary to initialize the

pointer on line ❶: we define a pointer which itself is not constant.

However, one can define constant pointers as well. The **const** modifier should be then added *after* the asterisk. Therefore

```
const int *pi;
```

or

```
int const *pi;
```

declares “normal” pointer to constant integer. Therefore, initialization is not needed. Let us note that **const** can appear before or after the name of type **int**. But

```
int i;
int *const pi = &i;
```

declares and defines (in the second line) a *constant pointer*. Now an initialization is necessary — we have to assign a value (the address of an existing **int** variable) to the pointer being defined, because it is a constant. The value of ***i*** can be modified — we can refer to it by the name ***i*** or by dereferencing ***pi*** (e.g., `*pi=7;`). But we cannot change the value of the pointer, i.e., it is now impossible to put the address of another **int** as the value of the pointer ***pi***.

Another example illustrating the discussion:

P41: constpoint.cpp Constant pointers and pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int *p, k = 5, m = 7;
6.
7.      const int cnts = 3;         // constant
8.      const int *q = &k;          // q - pointer to constant
9.      int *const r = &k;          // r - constant pointer
10.     const int arr[] = {1,2,3}; // array of constants
11.
12.     p = &cnts;                              ❶
13.     cnts = 1;                               ❷
14.     *q = m;                                 ❸
15.     q = &m;                                 ❹
16.     r = &m;                                 ❺
17.     k = 10;                                 ❻
18.     arr[1] = 9;                             ❼
19. }
```

Let us have a closer look at some lines of the program below (which isr illformed and will not compile!):

- Line ❶ is be illegal, as ***p*** is of type **int*** while ***&cnts*** is an address of a constant (its type is **const int***).
- On line ❷ we attempt to modify the value of a constant, what is illegal.
- On line ❸ we try to modify the value of the variable ***k*** (as currently ***q*** points to ***k***). This variable is not a constant. However, we try to modify ***k*** through the pointer ***q*** which *was* declared as a pointer to a constant — this is not allowed.
- Line ❹: the variable ***q*** is a pointer to a constant, but the pointer itself is not a constant, so changing its value is legal.

- Line ❺ would be illegal, as ***r*** is a constant pointer. It points to ***k*** and this cannot be changed.
- Line ❻ is legal: ***r*** is a constant pointer and points to ***k***: however, the value it points to is *not* constant and can be modified.
- Line ❼: ***arr[1]*** is a constant, because it is an element of an array of constants.

We can also define *constant pointers to constants*:

```
int i;
const int *const pi = &i;
```

Obviously, being constant, the pointer (***pi*** in our example) must be initialized — not necessarily with an address of a constant, however. The value pointed to by this pointer will be treated as a constant. Although it can be changed if we can refer to it by another name, it will be impossible to modify it by dereferencing this pointer. Also the value of the pointer itself cannot be changed: it will point to the variable ***i*** "forever". For example, in the program

P42: cnstcnst.cpp Constant pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int i = 1, m = 2;
6.      const int *const pi = &i;
7.
8.      cout << "Before *pi = " << *pi << endl;
9.      i = 3;                                    ❶
10.     cout << "After  *pi = " << *pi << endl;
11.
12.     *pi = 4; // NO: pi points to a constant
13.     pi = &m; // NO: pi is a constant pointer
14. }
```

it is possible to change the value of ***i*** by referring to it by the name ***i***, as this is not a constant (line ❶). But it would be illegal to modify the same value by referring to it by the name of the pointer ***pi***, or to modify the value of this pointer itself (last two lines).

Under the new C++11 standard, there is a stronger form of declaring **const** antness — instead of using **const**, we use the keyword **constexpr**. It means that it must be possible for the compiler to calculate the initializing value (of integer, floating point, object type.) This value can depend on values of literals, but also on values of other variables if they are also declared as **constexpr**. Compiler will check if this is the case and report an error if it is not. In the example below

P43: constexpr.cpp *constexpr* values

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      constexpr int hourfee = 7;
6.      constexpr int tim = 5;
```

```
7.
8.      int arr[10 + (tim-1)*hourfee];        ❶
9.
10.     cout << "number of elements in arr "
11.          << sizeof(arr)/sizeof(arr[0]) << endl;
12. }
```

the expression used to specify the dimension of an array (❶) is a compile-time constant which is really evaluated at compile time — otherwise the program could not have been compiled (with '`-pedantic`' option, at least), as static arrays require a compile-time constant initializer for their dimensions.

We will see in the following chapters that function invocations, object construction and invocations of methods on objects can also be **constexpr** — this was impossible under the old standard.

Prev: 7.3 Storage classes **Up:** 7. Variables **Next:** 7.5 L-values

T.R. Werner, September 27, 2023; 22:32

Subsections

- 7.4.1 Volatile variables
- 7.4.2 Constants

7.4 Type modifiers

Modifiers **volatile** and **const** are called *type* modifiers because they change the type of objects being declared/defined (although their binary representation remains unchanged).

7.4.1 Volatile variables

Volatile variables are declared with modifier **volatile**, e.g.,

```
volatile double x;
```

where **volatile double** is the name of a type which is different than just **double**. Declaring a variable as volatile means that its value can be modified in such a way that the program "does not know" about it. This can happen if, for example, the value can be changed by some external sensors. Volatile variables are treated in a special way: the compiler must ensure that their value stored in memory is always valid and current; all modification have to be executed immediately without buffering new values in registers or cache. Every time the value of a volatile variable is needed, it must be read from memory and not taken from registers or cache, because it is potentially possible that it was changed even if the compiler does not see any "reason" for such a change.

Volatile variables are never needed in "normal" programs and one has to remember that their use deteriorates efficiency of the program and makes the process of its optimization much harder.

Also, it is not true that volatile variables are useful in multi-threaded applications, as the details of their behavior can depend on implementation and are therefore not portable (in such situations, use rather **atomic** variables).

7.4.2 Constants

Constants are used very often and in many contexts. They are declared with modifier **const** (before or after the name of a type). They have to be initialized at the moment of creation; after that their value cannot be changed.

The value of constants cannot be changed after their creation.

For example, we can define a constant representing π as

```
const double PI = 3.1415926536;
```

but the construction like

```
const double PI;
PI = 3.1415926536; // WRONG !!!
```

would be illegal because in the second line we try to change the value of a constant that has already been created.

Some constant can be evaluated at compilation time (if the initializers contain literal values or other constants that can be evaluated at compile time — simple arithmetic on integral values is also tractable. More complex initializations will be performed at run time.

```
const int a = 2*11;
const int b = 5*a*a;
const int c = fun(a,b);
```

Variables ***a*** and ***b*** are compilation constants, but ***c*** is not, although its type is **const int**, because it will be initialized at run time, as it involves invoking a function.

In the example above, all variables are of type **const int** which is different than just **int**. For that reason, the following

```
const int i = 25;
int *pi = &i; // WRONG !!!
```

is illegal, as the type of '`&i`' is *pointer to constant integer* while the declared type of ***pi*** is *pointer to integer*.

However, an assignment in the opposite direction would be legal. If a pointer is declared as a pointer to a constant, it can be assigned the value of the address of a non- **const** variable. After that, this value cannot be changed if we refer to the variable using the name of the pointer: it can be changed, however, if we can refer to the same variable by another name. Let us look at an example:

P38: constants.cpp Constants

```
1. #include <iostream>
2. using namespace std;
3.
4. int fun(const int * pi) {
5.     //*pi = 2 * (*pi); // WRONG !!!
6.     return *pi;
7. }
8.
9. int main() {
10.     int i = 2;
11.     int res = fun(&i);                ❶
12.     cout << "res = " << res << endl;
13. }
```

The line commented out in the function would be illegal. We pass to the function the address of a “normal” integer (❶). However, the parameter of the function is declared as a *pointer to constant integer*, so the compiler will not allow to change the value of the variable pointed to by pointer ***pi*** (which is just non-const integer ***i*** from the **main** function).

Of course the same variable is modifiable in **main** because here it has type **int** (and not **const int**).

This behavior may lead to some confusion. For example, in the following program

P39: constants1.cpp More on constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int i = 2;
6.      const int *pi = &i;                ❶
7.
8.      i = 2*i;                           ❷
9.      cout << "  i = " <<   i << endl;
10.     cout << "*pi = " << *pi << endl;
11.     //*pi = -1;
12. }
```

variable ***i*** is not declared as a constant. But ***pi*** is a pointer of type **const int***, which means that the variable pointed to by ***pi*** should be treated as a constant. We initialize ***pi*** with the address of ***i*** (line ❶). Is ***i*** constant or not? The answer is:

- yes, if we refer to it through the name ***pi*** (by dereferencing it: ***pi**) — that is why the commented out last line would be illegal;
- no, if we refer to it by name ***i*** (as can be seen on line ❷, which is legal).

With the last line commented out, the program is correct and prints

```
  i = 4
*pi = 4
```

One has to clearly distinguish between pointers to constants and constant pointers. In all the above examples, pointers pointed to constants, but were not constant themselves. Variables pointed to by these pointers could not be modified, but it was possible to change the value of pointers, i.e., the addresses stored in them. Let us see one more example

P40: constants2.cpp Pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const int i = 2, j = 3;
6.
7.      const int *p = &i;                 ❶
8.      cout << "*p = " << *p << endl;
9.
10.     p = &j;
11.     cout << "*p = " << *p << endl;
12. }
```

Both variables, ***i*** and ***j*** are declared constant. The pointer ***p*** is a pointer to constant. But itself, it is not a constant: we initialize it with the address of ***i*** (❶), but then we change its value by assigning to it the address of ***j***. Note, that it was not necessary to initialize the

pointer on line ❶: we define a pointer which itself is not constant.

However, one can define constant pointers as well. The **const** modifier should be then added *after* the asterisk. Therefore

```
const int *pi;
```

or

```
int const *pi;
```

declares “normal” pointer to constant integer. Therefore, initialization is not needed. Let us note that **const** can appear before or after the name of type **int**. But

```
int i;
int *const pi = &i;
```

declares and defines (in the second line) a *constant pointer*. Now an initialization is necessary — we have to assign a value (the address of an existing **int** variable) to the pointer being defined, because it is a constant. The value of ***i*** can be modified — we can refer to it by the name ***i*** or by dereferencing ***pi*** (e.g., `*pi=7;`). But we cannot change the value of the pointer, i.e., it is now impossible to put the address of another **int** as the value of the pointer ***pi***.

Another example illustrating the discussion:

P41: constpoint.cpp Constant pointers and pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int *p, k = 5, m = 7;
6.
7.      const int cnts = 3;         // constant
8.      const int *q = &k;          // q - pointer to constant
9.      int *const r = &k;          // r - constant pointer
10.     const int arr[] = {1,2,3}; // array of constants
11.
12.     p = &cnts;                              ❶
13.     cnts = 1;                               ❷
14.     *q = m;                                 ❸
15.     q = &m;                                 ❹
16.     r = &m;                                 ❺
17.     k = 10;                                 ❻
18.     arr[1] = 9;                             ❼
19. }
```

Let us have a closer look at some lines of the program below (which isr illformed and will not compile!):

- Line ❶ is be illegal, as ***p*** is of type **int*** while ***&cnts*** is an address of a constant (its type is **const int***).
- On line ❷ we attempt to modify the value of a constant, what is illegal.
- On line ❸ we try to modify the value of the variable ***k*** (as currently ***q*** points to ***k***). This variable is not a constant. However, we try to modify ***k*** through the pointer ***q*** which *was* declared as a pointer to a constant — this is not allowed.
- Line ❹: the variable ***q*** is a pointer to a constant, but the pointer itself is not a constant, so changing its value is legal.

- Line ❺ would be illegal, as ***r*** is a constant pointer. It points to ***k*** and this cannot be changed.
- Line ❻ is legal: ***r*** is a constant pointer and points to ***k***: however, the value it points to is *not* constant and can be modified.
- Line ❼: ***arr[1]*** is a constant, because it is an element of an array of constants.

We can also define *constant pointers to constants*:

```
int i;
const int *const pi = &i;
```

Obviously, being constant, the pointer (***pi*** in our example) must be initialized — not necessarily with an address of a constant, however. The value pointed to by this pointer will be treated as a constant. Although it can be changed if we can refer to it by another name, it will be impossible to modify it by dereferencing this pointer. Also the value of the pointer itself cannot be changed: it will point to the variable ***i*** "forever". For example, in the program

P42: cnstcnst.cpp Constant pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int i = 1, m = 2;
6.      const int *const pi = &i;
7.
8.      cout << "Before *pi = " << *pi << endl;
9.      i = 3;                                    ❶
10.     cout << "After  *pi = " << *pi << endl;
11.
12.     *pi = 4; // NO: pi points to a constant
13.     pi = &m; // NO: pi is a constant pointer
14. }
```

it is possible to change the value of ***i*** by referring to it by the name ***i***, as this is not a constant (line ❶). But it would be illegal to modify the same value by referring to it by the name of the pointer ***pi***, or to modify the value of this pointer itself (last two lines).

Under the new C++11 standard, there is a stronger form of declaring **const** antness — instead of using **const**, we use the keyword **constexpr**. It means that it must be possible for the compiler to calculate the initializing value (of integer, floating point, object type.) This value can depend on values of literals, but also on values of other variables if they are also declared as **constexpr**. Compiler will check if this is the case and report an error if it is not. In the example below

P43: constexpr.cpp *constexpr* values

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      constexpr int hourfee = 7;
6.      constexpr int tim = 5;
```

```
7.
8.      int arr[10 + (tim-1)*hourfee];        ❶
9.
10.     cout << "number of elements in arr "
11.          << sizeof(arr)/sizeof(arr[0]) << endl;
12. }
```

the expression used to specify the dimension of an array (❶) is a compile-time constant which is really evaluated at compile time — otherwise the program could not have been compiled (with '`-pedantic`' option, at least), as static arrays require a compile-time constant initializer for their dimensions.

We will see in the following chapters that function invocations, object construction and invocations of methods on objects can also be **constexpr** — this was impossible under the old standard.

Prev: 7.3 Storage classes **Up:** 7. Variables **Next:** 7.5 L-values

T.R. Werner, September 27, 2023; 22:32

Subsections

- 7.4.1 Volatile variables
- 7.4.2 Constants

7.4 Type modifiers

Modifiers **volatile** and **const** are called *type* modifiers because they change the type of objects being declared/defined (although their binary representation remains unchanged).

7.4.1 Volatile variables

Volatile variables are declared with modifier **volatile**, e.g.,

```
volatile double x;
```

where **volatile double** is the name of a type which is different than just **double**. Declaring a variable as volatile means that its value can be modified in such a way that the program "does not know" about it. This can happen if, for example, the value can be changed by some external sensors. Volatile variables are treated in a special way: the compiler must ensure that their value stored in memory is always valid and current; all modification have to be executed immediately without buffering new values in registers or cache. Every time the value of a volatile variable is needed, it must be read from memory and not taken from registers or cache, because it is potentially possible that it was changed even if the compiler does not see any "reason" for such a change.

Volatile variables are never needed in "normal" programs and one has to remember that their use deteriorates efficiency of the program and makes the process of its optimization much harder.

Also, it is not true that volatile variables are useful in multi-threaded applications, as the details of their behavior can depend on implementation and are therefore not portable (in such situations, use rather **atomic** variables).

7.4.2 Constants

Constants are used very often and in many contexts. They are declared with modifier **const** (before or after the name of a type). They have to be initialized at the moment of creation; after that their value cannot be changed.

The value of constants cannot be changed after their creation.

For example, we can define a constant representing π as

```
const double PI = 3.1415926536;
```

but the construction like

```
const double PI;
PI = 3.1415926536; // WRONG !!!
```

would be illegal because in the second line we try to change the value of a constant that has already been created.

Some constant can be evaluated at compilation time (if the initializers contain literal values or other constants that can be evaluated at compile time — simple arithmetic on integral values is also tractable. More complex initializations will be performed at run time.

```
const int a = 2*11;
const int b = 5*a*a;
const int c = fun(a,b);
```

Variables ***a*** and ***b*** are compilation constants, but ***c*** is not, although its type is **const int**, because it will be initialized at run time, as it involves invoking a function.

In the example above, all variables are of type **const int** which is different than just **int**. For that reason, the following

```
const int i = 25;
int *pi = &i; // WRONG !!!
```

is illegal, as the type of '`&i`' is *pointer to constant integer* while the declared type of ***pi*** is *pointer to integer*.

However, an assignment in the opposite direction would be legal. If a pointer is declared as a pointer to a constant, it can be assigned the value of the address of a non- **const** variable. After that, this value cannot be changed if we refer to the variable using the name of the pointer: it can be changed, however, if we can refer to the same variable by another name. Let us look at an example:

P38: constants.cpp Constants

```
1. #include <iostream>
2. using namespace std;
3.
4. int fun(const int * pi) {
5.     //*pi = 2 * (*pi); // WRONG !!!
6.     return *pi;
7. }
8.
9. int main() {
10.     int i = 2;
11.     int res = fun(&i);                ❶
12.     cout << "res = " << res << endl;
13. }
```

The line commented out in the function would be illegal. We pass to the function the address of a “normal” integer (❶). However, the parameter of the function is declared as a *pointer to constant integer*, so the compiler will not allow to change the value of the variable pointed to by pointer ***pi*** (which is just non-const integer ***i*** from the **main** function).

Of course the same variable is modifiable in **main** because here it has type **int** (and not **const int**).

This behavior may lead to some confusion. For example, in the following program

P39: constants1.cpp More on constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int i = 2;
6.      const int *pi = &i;                ❶
7.
8.      i = 2*i;                           ❷
9.      cout << "  i = " <<   i << endl;
10.     cout << "*pi = " << *pi << endl;
11.     //*pi = -1;
12. }
```

variable ***i*** is not declared as a constant. But ***pi*** is a pointer of type **const int***, which means that the variable pointed to by ***pi*** should be treated as a constant. We initialize ***pi*** with the address of ***i*** (line ❶). Is ***i*** constant or not? The answer is:

- yes, if we refer to it through the name ***pi*** (by dereferencing it: ***pi**) — that is why the commented out last line would be illegal;
- no, if we refer to it by name ***i*** (as can be seen on line ❷, which is legal).

With the last line commented out, the program is correct and prints

```
  i = 4
*pi = 4
```

One has to clearly distinguish between pointers to constants and constant pointers. In all the above examples, pointers pointed to constants, but were not constant themselves. Variables pointed to by these pointers could not be modified, but it was possible to change the value of pointers, i.e., the addresses stored in them. Let us see one more example

P40: constants2.cpp Pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const int i = 2, j = 3;
6.
7.      const int *p = &i;                 ❶
8.      cout << "*p = " << *p << endl;
9.
10.     p = &j;
11.     cout << "*p = " << *p << endl;
12. }
```

Both variables, ***i*** and ***j*** are declared constant. The pointer ***p*** is a pointer to constant. But itself, it is not a constant: we initialize it with the address of ***i*** (❶), but then we change its value by assigning to it the address of ***j***. Note, that it was not necessary to initialize the

pointer on line ❶: we define a pointer which itself is not constant.

However, one can define constant pointers as well. The **const** modifier should be then added *after* the asterisk. Therefore

```
const int *pi;
```

or

```
int const *pi;
```

declares “normal” pointer to constant integer. Therefore, initialization is not needed. Let us note that **const** can appear before or after the name of type **int**. But

```
int i;
int *const pi = &i;
```

declares and defines (in the second line) a *constant pointer*. Now an initialization is necessary — we have to assign a value (the address of an existing **int** variable) to the pointer being defined, because it is a constant. The value of ***i*** can be modified — we can refer to it by the name ***i*** or by dereferencing ***pi*** (e.g., `*pi=7;`). But we cannot change the value of the pointer, i.e., it is now impossible to put the address of another **int** as the value of the pointer ***pi***.

Another example illustrating the discussion:

P41: constpoint.cpp Constant pointers and pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int *p, k = 5, m = 7;
6.
7.      const int cnts = 3;         // constant
8.      const int *q = &k;          // q - pointer to constant
9.      int *const r = &k;          // r - constant pointer
10.     const int arr[] = {1,2,3}; // array of constants
11.
12.     p = &cnts;                              ❶
13.     cnts = 1;                               ❷
14.     *q = m;                                 ❸
15.     q = &m;                                 ❹
16.     r = &m;                                 ❺
17.     k = 10;                                 ❻
18.     arr[1] = 9;                             ❼
19. }
```

Let us have a closer look at some lines of the program below (which isr illformed and will not compile!):

- Line ❶ is be illegal, as ***p*** is of type **int*** while ***&cnts*** is an address of a constant (its type is **const int***).
- On line ❷ we attempt to modify the value of a constant, what is illegal.
- On line ❸ we try to modify the value of the variable ***k*** (as currently ***q*** points to ***k***). This variable is not a constant. However, we try to modify ***k*** through the pointer ***q*** which *was* declared as a pointer to a constant — this is not allowed.
- Line ❹: the variable ***q*** is a pointer to a constant, but the pointer itself is not a constant, so changing its value is legal.

- Line ❺ would be illegal, as ***r*** is a constant pointer. It points to ***k*** and this cannot be changed.
- Line ❻ is legal: ***r*** is a constant pointer and points to ***k***: however, the value it points to is *not* constant and can be modified.
- Line ❼: ***arr[1]*** is a constant, because it is an element of an array of constants.

We can also define *constant pointers to constants*:

```
int i;
const int *const pi = &i;
```

Obviously, being constant, the pointer (***pi*** in our example) must be initialized — not necessarily with an address of a constant, however. The value pointed to by this pointer will be treated as a constant. Although it can be changed if we can refer to it by another name, it will be impossible to modify it by dereferencing this pointer. Also the value of the pointer itself cannot be changed: it will point to the variable ***i*** "forever". For example, in the program

P42: cnstcnst.cpp Constant pointers to constants

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int i = 1, m = 2;
6.      const int *const pi = &i;
7.
8.      cout << "Before *pi = " << *pi << endl;
9.      i = 3;                                    ❶
10.     cout << "After  *pi = " << *pi << endl;
11.
12.     *pi = 4; // NO: pi points to a constant
13.     pi = &m; // NO: pi is a constant pointer
14. }
```

it is possible to change the value of ***i*** by referring to it by the name ***i***, as this is not a constant (line ❶). But it would be illegal to modify the same value by referring to it by the name of the pointer ***pi***, or to modify the value of this pointer itself (last two lines).

Under the new C++11 standard, there is a stronger form of declaring **const** antness — instead of using **const**, we use the keyword **constexpr**. It means that it must be possible for the compiler to calculate the initializing value (of integer, floating point, object type.) This value can depend on values of literals, but also on values of other variables if they are also declared as **constexpr**. Compiler will check if this is the case and report an error if it is not. In the example below

P43: constexpr.cpp *constexpr* values

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      constexpr int hourfee = 7;
6.      constexpr int tim = 5;
```

```
7.
8.      int arr[10 + (tim-1)*hourfee];        ❶
9.
10.     cout << "number of elements in arr "
11.          << sizeof(arr)/sizeof(arr[0]) << endl;
12. }
```

the expression used to specify the dimension of an array (❶) is a compile-time constant which is really evaluated at compile time — otherwise the program could not have been compiled (with '`-pedantic`' option, at least), as static arrays require a compile-time constant initializer for their dimensions.

We will see in the following chapters that function invocations, object construction and invocations of methods on objects can also be **constexpr** — this was impossible under the old standard.

Prev: 7.3 Storage classes **Up:** 7. Variables **Next:** 7.5 L-values

T.R. Werner, September 27, 2023; 22:32

7.5 L-values

Every “datum” in the program must be stored somewhere in the computer. However, this place does not have to have a well defined, programmatically accessible address. It can be a register of the processor, or a special region used for temporary values as they are dynamically created and destroyed.

For example, when this fragment is executed

```
int x, y = 1;
// ...
x = y + 1;
```

the result of the addition '`y+1`' undoubtedly has to appear somewhere before it can be copied to the location of the variable ***x*** (perhaps in a register). Anyway, we cannot access this variable; all we can do is to store it somewhere or use it in another operation. A value of this sort, called a **temporary**, usually appears on the *right* hand side of assignments, as

in assignment statement the right hand side determines what is to be calculated (and represents a *value*), while the left hand side specifies a *location* where the result is to be stored.

Therefore, only expressions identifying programmatically accessible addresses can appear on the left hand side of assignments (their current value is irrelevant — it will be overwritten anyway). This leads us to the notion of **l-values** and **r-values**.

Expression which identifies a datum which has a well defined type and defined, programmatically accessible address is called an *l-value*.

This does not mean that l-values always represent modifiable variables; a variable declared as **const** is an l-value, but cannot be modified (at least if we refer to it by a name declared as **const** — it can be modifiable under another name). Generally, if something appears on the left hand side of an assignment, it has to be an l-value; the opposite is not true: there are l-values which cannot appear there.

Expression which has a well defined *value* is called an *r-value*.

An l-value is simultaneously an r-value, but an r-value does not have to be an l-value. The expression '`x+2`' has a value and is an r-value, but it is not an l-value.

Let us enumerate expressions which are l-values (the exact meaning of some of them will be explained in the next chapters):

- Identifier of a variable (possibly qualified, e.g., ***::x***).
- The result of function invocation, if that function returns a reference.

- The assignment statement as a whole: its value is the value of the left hand side *after* the assignment has been completed; the location it identifies is the address of the left hand side (which itself must be an l-value).
- Expressions with *prefix* increment or decrement operators (but *not* expressions with postfix decrement and increment operators: `++k` is, but `k++` is not an l-value).
- Expression with pointer dereference (`*p`) or dereferencing an element of an array using subscript operator (`tab[k]`).
- The result of the member selection operator if it does not refer to a function-like member of a class (`anObject.field` or `aPointer->field`).
- The result of the conditional expression, if both values are l-values (so, for example, `a>0?++i:++j` is, but `a>0?i++:j++` is not an l-value).
- The result of a conversion to a reference type (`(int&)k`).
- The result of the comma operator if the right argument in an l-value.

For example, in the last line of

```
int x = 2, y = x + 1;
// ...
int j = x = y;
```

the expression `x=y` is an l-value (and r-value as well) whose value is equal to the value of ***x*** after the assignment. If so, it can be put on the left hand side of an assignment:

```
(x=y-2) = 5;
```

The expression `(x=y-2)` has value of ***x*** after assignment (that is 1) and the address of this variable; the value 5 will be therefore assigned to ***x*** (erasing the 2 which has just been put there).

After

```
int *p = &( x>y ? x : y);
```

the pointer ***p*** will contain the address of the larger of variables ***x*** and ***y*** (the address operator, i.e., '*', can only be applied to l-values).

Other examples:

P44: lval.cpp L-values

```
1. #include <iostream>
2. using namespace std;
3.
4. int& timestwo(int& m) {
5.     static int count;
6.     cout << " In timestwo: count = " << count << endl;
7.     m *= 2;
8.     return count;
9. }
10.
11. void printTab(const int *tab, int size) {
12.     cout << "[ ";
13.     for (int i = 0; i < size; i++)
14.         cout << tab[i] << " ";
15.     cout << "]" << endl;
16. }
17.
18. int main() {
19.     int i = 1, j = 2, k = 3;
20.
```

```
21.     // assignment as an l-value
22.     (i=j) = k;
23.     cout << " A: i = " << i << " j = " << j
24.          <<    " k = " << k << endl;          // 3,2,3
25.
26.     int tab[] = {1,2,3,4}, *p = tab;
27.     cout << " B: before "; printTab(tab,4);
28.     *++++++p = 8;
29.     cout << " B:  after "; printTab(tab,4);
30.
31.     // now p points to the last element!
32.     cout << " C: ++*----p = " << ++*----p << endl; // 3
33.     cout << " C:    tab "; printTab(tab,4);
34.
35.     // conversion as an l-value
36.     int m = 7;
37.     timestwo( (int&)m=8 )++;  // conversion unnecessary!
38.     cout << "D1: m = " << m << endl;
39.     timestwo(m)++;
40.     cout << "D2: m = " << m << endl;
41.     int n = timestwo(m) = 10;
42.     cout << "D3: m = " << m << endl;
43.     cout << "D4: n = " << n << endl;
44.
45.     // comma operator
46.     k = (i=1, j=2) + 1;
47.     cout << " E: i = " << i << " j = " << j
48.          <<    " k = " << k << endl;          // 1,2,3
49.
50.     // conditional expression as an l-value
51.     (k > 2 ? i : j) = 5;
52.     cout << " F: i = " << i << " j = " << j
53.          <<    " k = " << k << endl;          // 5,2,3
54. }
```

The program prints:

```
 A: i = 3 j = 2 k = 3
 B: before [ 1 2 3 4 ]
 B:  after [ 1 2 3 8 ]
 C: ++*----p = 3
 C:    tab [ 1 3 3 8 ]
 In timestwo: count = 0
D1: m = 16
 In timestwo: count = 1
D2: m = 32
 In timestwo: count = 2
D3: m = 64
D4: n = 10
 E: i = 1 j = 2 k = 3
 F: i = 5 j = 2 k = 3
```

The assignment on line 22, '`(i=j)`', plays the rôle of an l-value — its address in the address of the variable ***i***. Therefore, the value of ***k*** will be assigned to ***i***, leaving the values of ***k*** and ***j*** unchanged. Let us notice, that this is different from '`i=j=k`', which would be interpreted as '`i=(j=k)`' and thus change the values of both ***i*** and ***j***.

On line 26 we define a four-element array; its address is assigned to the pointer ***p***. Let us analyze a somewhat bizarre expression `*++++++p` On line 28. First, we increment ***p*** three times. But ***p*** is a pointer, so its incrementation means the same as `((p+1)+1)+1` and amounts to shifting the address stored in ***p*** by three units: the pointer pointed to ***tab[0]***, and now it points to ***tab[3]***, the last element of the array. Dereferencing it will yield l-value ***tab[3]*** and the value 8 will be assigned to this element (see the line of the printout denoted by 'B').

A similar construction was used on line 32. First we decrement ***p*** twice — as ***p*** pointed to

tab[3], it now points to ***tab[1]***. Now we dereference this pointer: the expression `*--p` is another name of ***tab[1]*** The value of this element, which is of type **int**, is now incremented by one. Therefore, as an effect of this expression, the value of ***tab[1]*** will be incremented and pointer ***p*** will be shifted two units to the left (see line 'C' of the printout).

On line 37 we explicitly convert the argument of **timestwo** to the reference type **int&** — the argument is the l-value of assignment `m=8`, which is just ***m***. This conversion is in fact not needed here, as it would be performed automatically anyway.

Since we passed the reference to ***m*** to the function, all operations on this variable performed inside the function will be performed on the original variable ***m*** and will be visible after the function returns to **main** (see line 'D' of the printout).

The function **timestwo** returns a reference to a locally declared variable ***count***. It is locally declared, but static, so it exists after the function returns (it is created and initialized with 0 when the flow of control enters the function for the first time). The function returns the reference, therefore an l-value: the expression `timestwo(m)` is just another name of ***count*** and we can increment it (lines 27 and 39) or even put it on the left hand side of an assignment (line 41).

The expression '`(i=1, j=2)`' on line 46 is an l-value equivalent to ***j*** after the assignment `j=2` (so its value is 2). We then add one to this value and the result is assigned to ***k***.

The conditional expression `(k>2?i:j)` on line 51 is an l-value, because its second and third arguments (***i*** and ***j***) are l-values. As in our case the condition `k>2` is true, the whole expression is the l-value of ***i***, so the value 5 will be assigned to this variable.

---

8. Statements

Statements are the building blocks of every program. We will describe all kinds of statements available in C/C++.

Subsections

- 8.1 Categories of statements
- 8.2 Labels
- 8.3 Declarations
- 8.4 Null statement
- 8.5 Compound statement
- 8.6 Expression statement
- 8.7 Conditional statement
- 8.8 Selection (*switch*)statement (*switch*)
- 8.9 Iteration statements (loops)
  - 8.9.1 *while* loop
  - 8.9.2 *do-while* loop
  - 8.9.3 *for* loop
  - 8.9.4 *Foreach* loop

- 8.10 *continue* and *break* statements
- 8.11 *goto* statement
- 8.12 *return* statement
- 8.13 Exception handling statements

8.1 Categories of statements

Statements in C/C++ fall in several categories:

- null (empty) statement
- declarations
- compound statement
- expression statement
- conditional statements (**if**, **if...else**)
- selection (**switch**) statement
- iteration statements (**for**, **while**, **do...while**)
- break statement
- continue statement
- goto statement
- return statement
- exception-handling statements

Statements end with a semicolon; compound statements end with a closing brace. Statements can be labeled.

T.R. Werner, September 27, 2023; 22:32

8.2 Labels

Any active statement (i.e., not a declaration) can be labeled. The label is denoted by any legal identifier followed by a colon, which in turn is followed by the statement it labels. Labels can be used as targets of **goto** jumps, as in the following example:

P45: labincpp.cpp Jump to a labeled statement

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int tab[2][2][2]{{{1,2},{3,4}},{{5,6},{7,8}}};
6.      bool present = false;
7.      for (int i = 0; i < 2; i++)
8.          for (int j = 0; j < 2; j++)
9.              for (int k = 0; k < 2; k++)
10.                 if (tab[i][j][k] == 5) {     ❶
11.                     present = true;
12.                     goto LAB;
13.                 }
14.     LAB:if(present)                          ❷
15.         cout <<  "5 is present in the array" << endl;
16.     else
17.         cout << "there is no 5 in the array" << endl;
18. }
```

The triply nested loop in the program searches for an element of the array which is equal to 5. If such an element is found (❶), the variable ***present*** is set to **true** and the **goto** statement transfers the flow of control directly to the statement in line ❷, labeled with the identifier ***LAB:*** (outside of all loops). Note that **break** statement would *only* break from the innermost loop. If 5 is not found, all iterations of all loops are executed and ***present*** remains **false**. Note also that in the **goto** statement, only the identifier of the label is specified (*without* a colon).

Another type of labels may appear only in **switch** statement. There are only two such labels: **case** and **default** — see the section on switch statement.

8.3 Declarations

We will talk here about declarations of variables (declarations of functions will be explained in the section on functions).

Declaration specifies the type of a variable and its attributes. Usually it is connected with **definition** of this variable, i.e., with allocating a region of memory where the variable will be stored (physically created). Declarations have the form:

```
modifiers Type ident_list;
```

where:

- **modifiers** (which are optional) specify attributes of declared variables (e.g., **const**, **static**, which we know from the previous chapter). If there is more than one modifier, they can be specified in any order with no commas in-between.
- **Type** specifies the base type of declared variable (e.g., **double**, **string**, **unsigned short** etc. — see section on types).
- **ident_list** is a comma separated list of identifiers of declared variables. If a declaration is connected with definition, one can put an initializer after the identifier — an equal sign and an expression whose value should be used as the initial value or the brace-init.

Examples

```
double x{17.5}, y, z = 1;
static const double PI{3.14};
extern double PIHALF;
const double * const pPi = &PI;
```

The third line is an example of declaration without definition and hence without initialization. Of course, the variable ***PIHALF*** must be defined in another module of the program.

8.4 Null statement

Null statement consists of the single semicolon alone. Execution of such a statement has no effect but the statement is sometimes needed because of syntactic reasons. An example: in the *quick sort* procedure, which sorts an array, we look for the index of the first element of the array which is not greater than a certain value ***v***. This can be achieved by the following loop:

```
while (a[++i] < v);
```

where the body of the loop is empty, because the whole task is performed by the code checking the condition of the loop. The semicolon (null statement) is needed here; otherwise any statement which follows the loop would be erroneously considered the definition of its body.

8.5 Compound statement

Compound statements must be used when the syntax requires one statement, but we need a sequence of them. By enclosing a sequence of statements inside curly braces, we form one compound statement (grouping statement) out of, possibly many, simple (or compound) statements. Of course, each of the statements constituting a compound statement can itself be a compound statement. Remember that there is *no* semicolon after the closing brace — semicolons are only required after every simple statement separately.

Note also that definitions of functions have the form of a compound statement, as their definitions (bodies) are enclosed in curly braces, thus forming a block.

As we have already mentioned, the body of a compound statement constitutes a block: all variables declared inside such a block exist only from the place of their declaration/definition to the end of this block — when the flow of control leaves the block, they are removed from memory (the stack), so their names can be reused in declarations of other, completely independent variables. Thus

```
{
    int i = 5;
    {
        int k = fun(i);
        i += k;
    }
    cout << "i=" << i << endl;
}
```

is one compound statement consisting of three statements, one of which is itself a compound statement. After the whole statement has been executed, variables ***i*** and ***k*** do not exist any more and their names can be used for declaring other variables (actually, the variable ***k*** will be removed from the stack right after leaving the inner block, before the last line).

8.6 Expression statement

Each expression (meaningful sequence of lexemes) which has a well defined value can be treated as a statement — an **expression statement**. During execution its value is evaluated; the result, in some contexts, can be simply discarded. Therefore, in such situations, the expression has to have some side effects — if it does not, the compiler could simply ignore it (if it is absolutely sure that no side effect is possible). An expression may be an assignment, an increment or decrement operation, a function call, etc. Let us look at an example:

P46: exprstat.cpp Expression statements

```
1.  #include <iostream>
2.  #include <cstdio>
3.  using namespace std;
4.
5.  int main() {
6.      int k = 7, m = 8;
7.      ++k;                  ❶
8.      k+1;                  ❷
9.      k = 5;                ❸
10.     printf("OK?");        ❹
11.     k > m ? ++k : --m;    ❺
12.     new double(3.5);      ❻
13. }
```

We have several expression statements in this program: incrementation on line ❶, assignment in ❸, function invocation in ❹(**printf** returns **int**), conditional selection in ❺, creation of an object in ❻. The statement on line ❷ ('`k+1`') is a legal expression statement although it has no effect whatsoever and, most probably, will be completely ignored by the compiler (for a Java compiler, such expressions would be simply illegal). Creation of a variable on line ❻ does not have much sense either, as we do not store the returned address; the newly created variable of type **double** will not be accessible, but there will be no way to delete it. Generally, however, the compiler cannot ignore such senseless creation of objects since, in principle, it is possible that the constructor does produce some side effects, which are rightly expected by the programmer to occur.

8.7 Conditional statement

Conditional statement has two forms. The simpler form is

```
if ( b ) stmt
```

where ***b*** is an expression with logical value and ***stmt*** is a statement. As we know, a logical value does not have to be of type **bool** — any integer nonzero value and any nonempty pointer value will be considered **true**, while null values (zero or **nullptr** pointer) is interpreted as **false**. After evaluating the value of ***b***

- the conditional statement terminates without executing ***stmt*** if ***b*** has been evaluated to **false**;
- the statement ***stmt*** is executed if ***b*** has been evaluated to **true**.

Let us note that the syntax requires exactly *one* statement after the closing parenthesis. If we need more, we have to form a compound statement. For example, execution of the following code

```
if ( s != 0 )
    cerr << "Something went wrong. Quitting." << endl;
    exit(1);
```

will *always* terminate the program, as the `exit` statement is *not* "under" **if** here! We should rather have written

```
if ( s != 0 ) {
    cerr << "Something went wrong. Quitting." << endl;
    exit(1);
}
```

A very common mistake is to put a semicolon right after the closing parenthesis of the condition part of a conditional statement:

```
if ( s != 0 );
{
    cerr << "Something went wrong. Quitting." << endl;
    exit(1);
}
```

Compilation of this code will succeed: after `if (...)` there *is* a statement here, namely the null statement (semicolon). The block after the conditional statement will always be executed, as it has nothing to do with the preceding **if**.

Another pitfall is the fact that in C/C++ any integer or pointer value can be treated as a logical value (see the section on logical type): **false** corresponds to zero or null value, **true** to any non-zero or non-null value. This can lead to subtle mistakes that are not detected as errors by the compiler (such errors are impossible in Java). For example, in

```
if ( a = b ) stmt  // probably WRONG !!
```

if ***a*** and ***b*** are integer variables, the assignment expression ' `a=b`' has an (integer) value

equal to the value of the left-hand side *after* assignment has been performed. This value, zero or non-zero, will be interpreted as logical **false** or **true**, respectively, what might be, but might be not what was intended. Usually, what we mean is a comparison: *is the value of **a** equal to the value of **b** or not*. If this is the case, the form `a==b` should have been used.

In the new standard, just before the condition in parentheses, one can add the so called *init-statement* (with a semicolon after it). This can be an expression (something that has a value) or a declaration of one or more variables of the same type. In the latter case, declared variable(s) will be seen only inside the **if** statement (also in its **else** branch). For example, in the snippet below, ***delta*** can be used in all branches of the **if** statement, but doesn't exist outside it:

```
double a, b, c;
cout << "Enter three numbers: ";
std::cin >> a >> b >> c;

if (auto delta = b*b-4*a*c; delta > 0)
    cout << "Two roots, delta = " << delta << endl;
else if (delta < 0)
    cout << "No roots, delta = " << delta << endl;
else
    cout << "Two equal roots, delta = 0" << endl;
```

Usually this is advantageous, as we don't pollute our namespace with names of variables which are needed only locally.

The second form of the conditional statement contains additional **else** clause:

```
if ( b ) stmt1
else     stmt2
```

where, as before, ***b*** is an expression with logical value and ***stmt1*** and ***stmt2*** are statements (possibly compound). First, the value of ***b*** is evaluated and then:

- statement ***stmt1*** is executed and ***stmt2*** ignored if the value of ***b*** is **true**;
- statement ***stmt2*** is executed and ***stmt1*** ignored if the value of ***b*** is **false**.

Again, both ***stmt1*** and ***stmt2*** are single statements: if more are needed, blocking a sequence of statements into one compound statement has to be used.

Statements ***stmt1*** and ***stmt2*** can be conditional statements themselves. This can lead to complicated expressions, sometimes hard to understand. To make them clear, appropriate indentation should always be used. To find out which **if** corresponds to which **else** and *vice versa* one should observe the rule: an **else** clause corresponds to the closest **if** preceding it which was not followed by its corresponding **else** and which is in the same block and at the same level of nesting as the **else** clause. For example,

```
if ( b1 ) instr0 else if ( b2 ) instr1 if ( b3 ) instr2
else instr3 else instr4
```

is equivalent to

```
if ( b1 )           // 1
    stmt0
else                // 1
    if ( b2 )       // 2
        stmt1
        if ( b3 )   // 3
```

```
            stmt2
        else        // 3
            stmt3
    else            // 2
        stmt4
```

where **if** s and their corresponding **else** clauses are marked with the same numbers in comments.

Another example

```
1.      if ( val >= 0 )
2.      {
3.          if ( val > 9 ) cout << "Too large"<< endl;
4.      }
5.      else
6.          cout << "Too small" << endl;
```

Let us notice that **if** clause in the third line must have been put into a block in spite of the fact that it is a single statement and not a sequence of statements. Otherwise, the **else** clause from the fifth line would have been interpreted as the pair to this **if** and not, what was intended, as the pair to the **if** from the first line. To avoid such errors, keeping correct indentation may be very helpful.

Prev: 8.6 Expression statement **Up:** 8. Statements **Next:** 8.8 Selection (switch)statement (switch)

8.8 Selection (*switch*)statement (*switch*)

Selection statement (**switch** statement) can always be replaced by conditional statements (**if..else**) but is sometimes more convenient and enhances the clarity of code (also, may be more efficient). Its most general form is

```
switch (integ_expr) {
    case const1: list1
    case const2: list2
    // ...
    dafault: list
}
```

where ***integ_exp*** is an expression with integer value, ***const1***, ***const2***, ..., are *constant* (known at compile time) expressions with integer values, and ***list1***, ***list2***, ..., are lists of statements (possibly empty). Constants ***const1***, ***const2***, etc. can be given as integer literals, names of integer **constexpr** variables, or expressions involving subexpressions of this type. The number of **case** clauses is unlimited. The constants appearing in them must be all different. The **default** clause is optional and cannot appear more than once. It does *not* have to be the last clause (although usually it is).

When the **switch** statement is executed, the value of ***integ_exp*** is evaluated first. If this value is equal to the value of one of the constants appearing in **case** clauses, control is passed to the statement immediately following the matched **case**. If the value of ***integ_exp*** is not equal to any of the constants, control is passed to the statement immediately following the **default** clause if it exists; if it does not exist, none of the statements is executed.

When the flow of control is passed to one of the **case** or **default** clauses, it executes all statements from the list belonging to this clause and continues with *all* lists that follow (the **case** and **default** labels are then “transparent” — only statements from the lists are seen). If we want to break out of the **switch** statement earlier, we have to use **break**, **return** or **goto** statements (see below).

Selection statement is illustrated in the following program. The function **sw** prints (by calling the function **g**) a series of asterisks of the length depending on the value of function's argument: 4 asterisks for argument value equal to 1, 2 asterisks for argument 5, zero for 2 or 3, and 3 asterisks for any other value of the argument.

P47: switch.cpp *Switch* statement (1)

```
1. #include <iostream>
2. using namespace std;
3.
4. void g( ) {
5.     cout << '*';
6. }
```

```
7.
8. void sw(int k) {
9.     cout << k << ": ";
10.    switch ( k ) {
11.        default: g( );                    ❶
12.        case  5: g( ); g( );              ❷
13.        case  3:
14.        case  2: break;                   ❸
15.        case  1: g( ); g( ); g( ); g( );
16.    }
17.    cout << endl;
18. }
19.
20. int main() {
21.    sw(9);
22.    sw(5);
23.    sw(4);
24.    sw(3);
25.    sw(2);
26.    sw(1);
27.    sw(0);
28. }
```

The program prints the following results:

```
9: ***
5: **
4: ***
3:
2:
1: ****
0: ***
```

Let us note that for arguments other than 1, 2, 3, 5 the flow of control is transferred to **default** clause (line ❶); the function **g** is invoked (one asterisk), then control is transferred to the list following '`case 5`' (two asterisks), then to the list following '`case 3`' (which is empty), and finally to the list after '`case 2`' (line ❸) which terminates the execution of the whole **switch** statement by executing **break** (the **break** transfers the flow of control to the first statement *after* the **switch** construct). Therefore, for ***k*** different than 1, 2, 3, or 5, three asterisks will be printed. If the **switch** statement appears in a function, we can also use **return** to break out of it (and out of the function).

The function **hexVal** from the program below returns the value of a character passed as argument treating it as a hexadecimal digit (or `-1` if the character does not correspond to any hexadecimal digit):

P48: hex.cpp *Switch statement (2)*

```
1. #include <iostream>
2. using namespace std;
3.
4. int hexVal(char c) {
5.     switch ( c ) {
6.         case '0': case '1': case '2':   ❶
7.         case '3': case '4': case '5':
8.         case '6': case '7': case '8':
9.         case '9':
10.            return c - '0';             ❷
11.
12.        case 'a': case 'b': case 'c':
13.        case 'd': case 'e':
14.        case 'f':
15.            return 10 + c - 'a';        ❸
16.
```

```
17.         case 'A': case 'B': case 'C':
18.         case 'D': case 'E':
19.         case 'F':
20.             return 10 + c - 'A';        ❹
21.
22.         default: return -1;             ❺
23.     }
24. }
25.
26. int main() {
27.     cout << "A = " << hexVal('A') << endl
28.          << "f = " << hexVal('f') << endl
29.          << "9 = " << hexVal('9') << endl
30.          << "b = " << hexVal('b') << endl
31.          << "Z = " << hexVal('Z') << endl;
32. }
```

On line ❶ and three next lines, we group many empty **case** clauses, except for the last which contain a **return** statement. As we can see, for any value of the variable ***c*** which corresponds to a “normal” digit, flow of control will reach the **return** statement on line ❷ and the correct value will be returned by the function (***c*** is of type **char**, so will in fact contain the ASCII code of a digit: by subtracting the ASCII code of the character '0' we get its numerical value). Similarly, for any lower-case letter corresponding to a hexadecimal digit, the control will reach line ❸, and for upper-case letters — line ❹. For any other values of ***c*** the **default** clause will be used, returning `-1` on line ❺. The program prints:

```
A = 10
f = 15
9 = 9
b = 11
Z = -1
```

As with the **if** statement, the new standard allows us to use an *init-statement* in **switch**:

```
srand(time(nullptr));
switch (auto r = std::rand()%30; r/10) {
    case 0:  cout << r << " -> first ten\n"; break;
    case 1:  cout << r << " -> second ten\n"; break;
    default: cout << r << " -> third ten\n";
}
```

Prev: 8.7 Conditional statement **Up:** 8. Statements **Next:** 8.9 Iteration statements (loops)

Subsections

- 8.9.1 *while* loop
- 8.9.2 *do-while* loop
- 8.9.3 *for* loop
- 8.9.4 *Foreach* loop

8.9 Iteration statements (loops)

Iteration statements (or **loops**) are used when the repetition of a portion of the code is needed. The same fragment of code is executed cyclically as long as some logical condition holds true. It is the responsibility of the programmer to ensure that this condition will eventually become false so the loop can terminate. Another way to terminate the execution of a loop is by using **return**, **break** or **goto** statements.

There are three basic forms of iteration statements. Any of them can be easily transformed to any other, so programmers can choose a form best suited for a particular purpose at hand.

In each of these forms, loops can be modified or terminated by instructions **break** or **continue**.

8.9.1 *while* loop

The **while** loop has the form

```
while ( b ) stmt
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. The syntax says about *one* statement: if more are required, they have to be “packed” into one compound statement by enclosing them in braces (see the section on compound statements).

Execution of the **while** statement proceeds in a cyclic way as follows:

- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **while** statement terminates and the flow of control passes to the next statement after the **while** statement (***stmt*** is *not* executed);
- if ***b*** is **true** the statement ***stmt*** is executed and the flow of control passes again to the beginning of the **while** statement.

For example, the following snippet will find (in ***num***) the first natural number of the form 2^n which is larger than a given natural number ***lim***:

```
int numb = 1;
while ( numb <= lim ) numb *= 2;
```

while this code

```
int age = 0;
while ( age < 5 || age > 100) cin >> age;
```

will read from the standard input an **int** until a number from the range $[5, 100]$ is entered.

8.9.2 *do-while* loop

The **do** (or **do-while**) loop has the form

```
do stmt while ( b )
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. Again, the syntax says about *one* statement, so if more are required, they have to be gathered into one compound statement.

The execution goes (cyclically) as follows:

- the statement ***stmt*** is executed;
- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **do-while** statement terminates and the flow of control passes to the next statement after the **do-while** statement;
- if ***b*** is **true** the flow of control passes again to the beginning of the **do-while** statement.

In the **do...while** statement the statement ***stmt*** is executed first and then the termination condition ***b*** is checked; in the **while** loop the order was opposite: first the termination condition was checked, and then ***stmt*** was executed. Therefore, in the **do-while** loop, ***stmt*** will *always* be executed at least once, whereas in the **while** loop it is possible that ***stmt*** is not executed at all (when ***b*** is **false** right from the beginning).

The program below simulates a series of rolls of two dice. The outcome of each roll is obtained with the help of pseudo-random number generator (**rand** and initialization with **srand** from the header **<cstdlib>**). The game ends when we get six spots on both dice:

P49: dice.cpp *do...while loop*

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  using namespace std;
4.
5.  int main() {
6.      int x, y, roll = 0;
7.
8.      srand(unsigned(time(0)));
9.
10.     do {
11.         x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
12.         y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
13.         cout << "Roll no " << ++roll   << ": ("
14.              <<  x << ", " << y << ")" << endl;
15.     } while (x + y != 12);
16. }
```

As the number of rolls is not known in advance and there must be at least one roll to end the game, the form **do...while** is the most natural form of loop here. The outcome of the program was

```
Roll no 1: (1, 5)
Roll no 2: (5, 3)
Roll no 3: (6, 2)
Roll no 4: (5, 3)
Roll no 5: (1, 5)
Roll no 6: (3, 2)
Roll no 7: (3, 3)
Roll no 8: (5, 4)
Roll no 9: (3, 2)
Roll no 10: (4, 5)
Roll no 11: (6, 6)
```

Of course, as we use random number generator, the outcome will be different each time we run the program.

8.9.3 *for* loop

The **for** loop has the form

```
for ( init ; b ; incr ) stmt
```

where ***stmt*** is a statement (possibly compound, possibly empty — just a semicolon). The expression ***b*** must have a value convertible to **bool** — this expression can be omitted and then its value defaults to **true**. The expression ***init*** can be

- omitted;
- *one* declaration statement, possibly declaring/defining several variables of the same type with or without initialization;
- any number of comma-separated expression statements (see the section on expression statements).

The “incrementing” part (denoted by ***incr*** here) can be omitted or be a comma-separated list of any number of expression statements.

Even if one or more parts, ***init***, ***b*** or ***incr*** is omitted, the semicolons and parentheses must be present.

The execution of the **for** loop proceeds as follows;

1. If ***init*** is not empty, it is executed. If it is a sequence of expression statements, they are executed from left to right and their values are ignored. If it is a declaration, the scope of the variables declared will contain the ***b*** and ***incr*** parts and the body of the loop (the statement ***stmt***, which can be a compund statement) — they will not exist after the termination of the loop. Statements in ***init*** are executed always, but once only: when the flow of control enters the loop.
2. The value of ***b*** is evaluated and converted to type **bool** (if ***b*** is empty, **true** is assumed). If the value of ***b*** is **false**, the loop terminates.
3. The body of the loop (statement ***stmt***) is executed.
4. If ***incr*** is not empty, expression statements from this part are executed and their values are ignored.
5. Flow of control returns to item 2.

It is not possible to declare in the ***init*** part variables of different type: that would require more than one declaration statements:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
    // ... WRONG !!!
}
```

It is, however, possible to declare more than one variables of the same type:

```
for (int i = 0, k = size-1; i < k; i++, k--) {
    // OK
}
```

The **for** loop is the most general form of loop. It is particularly well suited in cases where one knows in advance how many iterations will be needed. Usually we initialize values of some kind of "counters" in the ***init*** part and increment (or decrement) them in the ***incr*** part.

As an example let us consider a simple program where function **reverse** reverses the order of elements in an array of integers:

P50: revers.cpp *for loop*

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void reverse(int *arr, int size) {
5.      if ( size < 2 ) return;
6.
7.      for (int i = 0, k = size-1, aux; i < k; i++, k--) { ❶
8.          aux    = arr[i];
9.          arr[i] = arr[k];
10.         arr[k] = aux;
11.     }
12. }
13.
14. void printArr(int *arr, int size) {
15.     cout << "[ ";
16.     for (int i = 0; i < size; i++)
17.         cout << arr[i] << " ";
18.     cout << "]" << endl;
19. }
20.
21. int main() {
22.     int arr[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
23.     int size = sizeof(arr)/sizeof(arr[0]);
24.
25.     printArr(arr,size);
26.     reverse (arr,size);
27.     printArr(arr,size);
28. }
```

The loop on line ❶ uses two indices — one, ***i***, runs over elements of table ***tab*** from the beginning forwards and the other, ***k***, from the last element backwards; the loop terminates when these indices "cross" each other. In this way the first and the last elements are exchanged, then the second with the last but one, and so forth. In the ***init*** part, we declared also an auxiliary variable ***aux***, which is used when exchanging the values of any two elements. The result of this program is

```
[ 1 3 5 7 2 4 -9 12 ]
[ 12 -9 4 2 7 5 3 1 ]
```

8.9.4 *Foreach* loop

The C++11 standard introduced another form of looping, the so called „foreach" loop. It can be used to iterate over a collection from the standard library (for example **std::array**, but is also applicable to normal static arrays — under the condition that in a given scope the array is visible as having the array type (and not pointer type, as is the case after passing it to a function). Let us look at the syntax:

P51: foreach.cpp "Foreach" loop

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int arr[] = {1,2,3,4,9,8,7,6};
6.
7.      for (int e  : arr) cout << e << " ";    ❶
8.      cout << endl;
9.      for (int& e : arr) e -= 1;              ❷
10.     for (auto e : arr) cout << e << " ";    ❸
11.     cout << endl;
12. }
```

On line ❶ the variable ***e*** will be initialized in each iteration of the loop with an unmodifiable *copy* of subsequent element of the array ***arr***. On line ❷, however, in each iteration ***e*** will be the *reference* to subsequent element of the array, so elements can be modified (here we just subtract 1 from each element). As we can see on line ❸, we do not have to specify the type of the elements explicitly — the type can be deduced from the type of the array (more generally, of the collection). The same applies to references: on line ❷, we could have used `auto&` instead of `int&`. The output of the program is

```
1 2 3 4 9 8 7 6
0 1 2 3 8 7 6 5
```

It is also possible to iterate over a collection created *ad hoc* — as surrounded in braces sequence of values of the same type, as in the program

P52: adhocforeach.cpp Loop over *ad hoc* created collection

```
1.  #include <iostream>
2.
3.  long long factorial(int n) {
4.      return n < 2 ? 1 : n*factorial(n-1);
5.  }
6.
7.  int main() {
8.      for (auto n : {12, 14, 16, 18, 20}) {
9.          if (auto f = factorial(n); f > 1e17)
10.             std::cout << f << " is really big\n";
11.         else
12.             std::cout << f << " is not so big\n";
13.     }
14. }
```

which prints

```
479001600 is not so big
87178291200 is not so big
20922789888000 is not so big
6402373705728000 is not so big
2432902008176640000 is really big
```

Prev: 8.8 Selection (switch)statement (switch) **Up:** 8. Statements **Next:** 8.10 continue and break statements

T.R. Werner, September 27, 2023; 22:32

Subsections

- 8.9.1 *while* loop
- 8.9.2 *do-while* loop
- 8.9.3 *for* loop
- 8.9.4 *Foreach* loop

8.9 Iteration statements (loops)

Iteration statements (or **loops**) are used when the repetition of a portion of the code is needed. The same fragment of code is executed cyclically as long as some logical condition holds true. It is the responsibility of the programmer to ensure that this condition will eventually become false so the loop can terminate. Another way to terminate the execution of a loop is by using **return**, **break** or **goto** statements.

There are three basic forms of iteration statements. Any of them can be easily transformed to any other, so programmers can choose a form best suited for a particular purpose at hand.

In each of these forms, loops can be modified or terminated by instructions **break** or **continue**.

8.9.1 *while* loop

The **while** loop has the form

```
while ( b ) stmt
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. The syntax says about *one* statement: if more are required, they have to be “packed” into one compound statement by enclosing them in braces (see the section on compound statements).

Execution of the **while** statement proceeds in a cyclic way as follows:

- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **while** statement terminates and the flow of control passes to the next statement after the **while** statement (***stmt*** is *not* executed);
- if ***b*** is **true** the statement ***stmt*** is executed and the flow of control passes again to the beginning of the **while** statement.

For example, the following snippet will find (in ***num***) the first natural number of the form 2^n which is larger than a given natural number ***lim***:

```
int numb = 1;
while ( numb <= lim ) numb *= 2;
```

while this code

```
int age = 0;
while ( age < 5 || age > 100) cin >> age;
```

will read from the standard input an **int** until a number from the range $[5, 100]$ is entered.

8.9.2 *do-while* loop

The **do** (or **do-while**) loop has the form

```
do stmt while ( b )
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. Again, the syntax says about *one* statement, so if more are required, they have to be gathered into one compound statement.

The execution goes (cyclically) as follows:

- the statement ***stmt*** is executed;
- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **do-while** statement terminates and the flow of control passes to the next statement after the **do-while** statement;
- if ***b*** is **true** the flow of control passes again to the beginning of the **do-while** statement.

In the **do...while** statement the statement ***stmt*** is executed first and then the termination condition ***b*** is checked; in the **while** loop the order was opposite: first the termination condition was checked, and then ***stmt*** was executed. Therefore, in the **do-while** loop, ***stmt*** will *always* be executed at least once, whereas in the **while** loop it is possible that ***stmt*** is not executed at all (when ***b*** is **false** right from the beginning).

The program below simulates a series of rolls of two dice. The outcome of each roll is obtained with the help of pseudo-random number generator (**rand** and initialization with **srand** from the header **<cstdlib>**). The game ends when we get six spots on both dice:

P49: dice.cpp *do...while loop*

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  using namespace std;
4.
5.  int main() {
6.      int x, y, roll = 0;
7.
8.      srand(unsigned(time(0)));
9.
10.     do {
11.         x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
12.         y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
13.         cout << "Roll no " << ++roll   << ": ("
14.              <<  x << ", " << y << ")" << endl;
15.     } while (x + y != 12);
16. }
```

As the number of rolls is not known in advance and there must be at least one roll to end the game, the form **do...while** is the most natural form of loop here. The outcome of the program was

```
Roll no 1: (1, 5)
Roll no 2: (5, 3)
Roll no 3: (6, 2)
Roll no 4: (5, 3)
Roll no 5: (1, 5)
Roll no 6: (3, 2)
Roll no 7: (3, 3)
Roll no 8: (5, 4)
Roll no 9: (3, 2)
Roll no 10: (4, 5)
Roll no 11: (6, 6)
```

Of course, as we use random number generator, the outcome will be different each time we run the program.

8.9.3 *for* loop

The **for** loop has the form

```
for ( init ; b ; incr ) stmt
```

where ***stmt*** is a statement (possibly compound, possibly empty — just a semicolon). The expression ***b*** must have a value convertible to **bool** — this expression can be omitted and then its value defaults to **true**. The expression ***init*** can be

- omitted;
- *one* declaration statement, possibly declaring/defining several variables of the same type with or without initialization;
- any number of comma-separated expression statements (see the section on expression statements).

The “incrementing” part (denoted by ***incr*** here) can be omitted or be a comma-separated list of any number of expression statements.

Even if one or more parts, ***init***, ***b*** or ***incr*** is omitted, the semicolons and parentheses must be present.

The execution of the **for** loop proceeds as follows;

1. If ***init*** is not empty, it is executed. If it is a sequence of expression statements, they are executed from left to right and their values are ignored. If it is a declaration, the scope of the variables declared will contain the ***b*** and ***incr*** parts and the body of the loop (the statement ***stmt***, which can be a compund statement) — they will not exist after the termination of the loop. Statements in ***init*** are executed always, but once only: when the flow of control enters the loop.
2. The value of ***b*** is evaluated and converted to type **bool** (if ***b*** is empty, **true** is assumed). If the value of ***b*** is **false**, the loop terminates.
3. The body of the loop (statement ***stmt***) is executed.
4. If ***incr*** is not empty, expression statements from this part are executed and their values are ignored.
5. Flow of control returns to item 2.

It is not possible to declare in the ***init*** part variables of different type: that would require more than one declaration statements:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
    // ... WRONG !!!
}
```

It is, however, possible to declare more than one variables of the same type:

```
for (int i = 0, k = size-1; i < k; i++, k--) {
    // OK
}
```

The **for** loop is the most general form of loop. It is particularly well suited in cases where one knows in advance how many iterations will be needed. Usually we initialize values of some kind of "counters" in the ***init*** part and increment (or decrement) them in the ***incr*** part.

As an example let us consider a simple program where function **reverse** reverses the order of elements in an array of integers:

P50: revers.cpp *for loop*

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void reverse(int *arr, int size) {
5.      if ( size < 2 ) return;
6.
7.      for (int i = 0, k = size-1, aux; i < k; i++, k--) { ❶
8.          aux    = arr[i];
9.          arr[i] = arr[k];
10.         arr[k] = aux;
11.     }
12. }
13.
14. void printArr(int *arr, int size) {
15.     cout << "[ ";
16.     for (int i = 0; i < size; i++)
17.         cout << arr[i] << " ";
18.     cout << "]" << endl;
19. }
20.
21. int main() {
22.     int arr[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
23.     int size = sizeof(arr)/sizeof(arr[0]);
24.
25.     printArr(arr,size);
26.     reverse (arr,size);
27.     printArr(arr,size);
28. }
```

The loop on line ❶ uses two indices — one, ***i***, runs over elements of table ***tab*** from the beginning forwards and the other, ***k***, from the last element backwards; the loop terminates when these indices "cross" each other. In this way the first and the last elements are exchanged, then the second with the last but one, and so forth. In the ***init*** part, we declared also an auxiliary variable ***aux***, which is used when exchanging the values of any two elements. The result of this program is

```
[ 1 3 5 7 2 4 -9 12 ]
[ 12 -9 4 2 7 5 3 1 ]
```

8.9.4 *Foreach* loop

The C++11 standard introduced another form of looping, the so called „foreach" loop. It can be used to iterate over a collection from the standard library (for example **std::array**, but is also applicable to normal static arrays — under the condition that in a given scope the array is visible as having the array type (and not pointer type, as is the case after passing it to a function). Let us look at the syntax:

P51: foreach.cpp "Foreach" loop

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int arr[] = {1,2,3,4,9,8,7,6};
6.
7.      for (int e  : arr) cout << e << " ";    ❶
8.      cout << endl;
9.      for (int& e : arr) e -= 1;              ❷
10.     for (auto e : arr) cout << e << " ";    ❸
11.     cout << endl;
12. }
```

On line ❶ the variable ***e*** will be initialized in each iteration of the loop with an unmodifiable *copy* of subsequent element of the array ***arr***. On line ❷, however, in each iteration ***e*** will be the *reference* to subsequent element of the array, so elements can be modified (here we just subtract 1 from each element). As we can see on line ❸, we do not have to specify the type of the elements explicitly — the type can be deduced from the type of the array (more generally, of the collection). The same applies to references: on line ❷, we could have used `auto&` instead of `int&`. The output of the program is

```
1 2 3 4 9 8 7 6
0 1 2 3 8 7 6 5
```

It is also possible to iterate over a collection created *ad hoc* — as surrounded in braces sequence of values of the same type, as in the program

P52: adhocforeach.cpp Loop over *ad hoc* created collection

```
1.  #include <iostream>
2.
3.  long long factorial(int n) {
4.      return n < 2 ? 1 : n*factorial(n-1);
5.  }
6.
7.  int main() {
8.      for (auto n : {12, 14, 16, 18, 20}) {
9.          if (auto f = factorial(n); f > 1e17)
10.             std::cout << f << " is really big\n";
11.         else
12.             std::cout << f << " is not so big\n";
13.     }
14. }
```

which prints

```
479001600 is not so big
87178291200 is not so big
20922789888000 is not so big
6402373705728000 is not so big
2432902008176640000 is really big
```

Prev: 8.8 Selection (switch)statement (switch) **Up:** 8. Statements **Next:** 8.10 continue and break statements

T.R. Werner, September 27, 2023; 22:32

Subsections

- 8.9.1 *while* loop
- 8.9.2 *do-while* loop
- 8.9.3 *for* loop
- 8.9.4 *Foreach* loop

8.9 Iteration statements (loops)

Iteration statements (or **loops**) are used when the repetition of a portion of the code is needed. The same fragment of code is executed cyclically as long as some logical condition holds true. It is the responsibility of the programmer to ensure that this condition will eventually become false so the loop can terminate. Another way to terminate the execution of a loop is by using **return**, **break** or **goto** statements.

There are three basic forms of iteration statements. Any of them can be easily transformed to any other, so programmers can choose a form best suited for a particular purpose at hand.

In each of these forms, loops can be modified or terminated by instructions **break** or **continue**.

8.9.1 *while* loop

The **while** loop has the form

```
while ( b ) stmt
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. The syntax says about *one* statement: if more are required, they have to be “packed” into one compound statement by enclosing them in braces (see the section on compound statements).

Execution of the **while** statement proceeds in a cyclic way as follows:

- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **while** statement terminates and the flow of control passes to the next statement after the **while** statement (***stmt*** is *not* executed);
- if ***b*** is **true** the statement ***stmt*** is executed and the flow of control passes again to the beginning of the **while** statement.

For example, the following snippet will find (in ***num***) the first natural number of the form 2^n which is larger than a given natural number ***lim***:

```
int numb = 1;
while ( numb <= lim ) numb *= 2;
```

while this code

```
int age = 0;
while ( age < 5 || age > 100) cin >> age;
```

will read from the standard input an **int** until a number from the range $[5, 100]$ is entered.

8.9.2 *do-while* loop

The **do** (or **do-while**) loop has the form

```
do stmt while ( b )
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. Again, the syntax says about *one* statement, so if more are required, they have to be gathered into one compound statement.

The execution goes (cyclically) as follows:

- the statement ***stmt*** is executed;
- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **do-while** statement terminates and the flow of control passes to the next statement after the **do-while** statement;
- if ***b*** is **true** the flow of control passes again to the beginning of the **do-while** statement.

In the **do...while** statement the statement ***stmt*** is executed first and then the termination condition ***b*** is checked; in the **while** loop the order was opposite: first the termination condition was checked, and then ***stmt*** was executed. Therefore, in the **do-while** loop, ***stmt*** will *always* be executed at least once, whereas in the **while** loop it is possible that ***stmt*** is not executed at all (when ***b*** is **false** right from the beginning).

The program below simulates a series of rolls of two dice. The outcome of each roll is obtained with the help of pseudo-random number generator (**rand** and initialization with **srand** from the header **<cstdlib>**). The game ends when we get six spots on both dice:

P49: dice.cpp *do...while loop*

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  using namespace std;
4.
5.  int main() {
6.      int x, y, roll = 0;
7.
8.      srand(unsigned(time(0)));
9.
10.     do {
11.         x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
12.         y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
13.         cout << "Roll no " << ++roll   << ": ("
14.              <<  x << ", " << y << ")" << endl;
15.     } while (x + y != 12);
16. }
```

As the number of rolls is not known in advance and there must be at least one roll to end the game, the form **do...while** is the most natural form of loop here. The outcome of the program was

```
Roll no 1: (1, 5)
Roll no 2: (5, 3)
Roll no 3: (6, 2)
Roll no 4: (5, 3)
Roll no 5: (1, 5)
Roll no 6: (3, 2)
Roll no 7: (3, 3)
Roll no 8: (5, 4)
Roll no 9: (3, 2)
Roll no 10: (4, 5)
Roll no 11: (6, 6)
```

Of course, as we use random number generator, the outcome will be different each time we run the program.

8.9.3 *for* loop

The **for** loop has the form

```
for ( init ; b ; incr ) stmt
```

where ***stmt*** is a statement (possibly compound, possibly empty — just a semicolon). The expression ***b*** must have a value convertible to **bool** — this expression can be omitted and then its value defaults to **true**. The expression ***init*** can be

- omitted;
- *one* declaration statement, possibly declaring/defining several variables of the same type with or without initialization;
- any number of comma-separated expression statements (see the section on expression statements).

The “incrementing” part (denoted by ***incr*** here) can be omitted or be a comma-separated list of any number of expression statements.

Even if one or more parts, ***init***, ***b*** or ***incr*** is omitted, the semicolons and parentheses must be present.

The execution of the **for** loop proceeds as follows;

1. If ***init*** is not empty, it is executed. If it is a sequence of expression statements, they are executed from left to right and their values are ignored. If it is a declaration, the scope of the variables declared will contain the ***b*** and ***incr*** parts and the body of the loop (the statement ***stmt***, which can be a compund statement) — they will not exist after the termination of the loop. Statements in ***init*** are executed always, but once only: when the flow of control enters the loop.
2. The value of ***b*** is evaluated and converted to type **bool** (if ***b*** is empty, **true** is assumed). If the value of ***b*** is **false**, the loop terminates.
3. The body of the loop (statement ***stmt***) is executed.
4. If ***incr*** is not empty, expression statements from this part are executed and their values are ignored.
5. Flow of control returns to item 2.

It is not possible to declare in the ***init*** part variables of different type: that would require more than one declaration statements:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
    // ... WRONG !!!
}
```

It is, however, possible to declare more than one variables of the same type:

```
for (int i = 0, k = size-1; i < k; i++, k--) {
    // OK
}
```

The **for** loop is the most general form of loop. It is particularly well suited in cases where one knows in advance how many iterations will be needed. Usually we initialize values of some kind of "counters" in the ***init*** part and increment (or decrement) them in the ***incr*** part.

As an example let us consider a simple program where function **reverse** reverses the order of elements in an array of integers:

P50: revers.cpp *for loop*

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void reverse(int *arr, int size) {
5.      if ( size < 2 ) return;
6.
7.      for (int i = 0, k = size-1, aux; i < k; i++, k--) { ❶
8.          aux    = arr[i];
9.          arr[i] = arr[k];
10.         arr[k] = aux;
11.     }
12. }
13.
14. void printArr(int *arr, int size) {
15.     cout << "[ ";
16.     for (int i = 0; i < size; i++)
17.         cout << arr[i] << " ";
18.     cout << "]" << endl;
19. }
20.
21. int main() {
22.     int arr[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
23.     int size = sizeof(arr)/sizeof(arr[0]);
24.
25.     printArr(arr,size);
26.     reverse (arr,size);
27.     printArr(arr,size);
28. }
```

The loop on line ❶ uses two indices — one, ***i***, runs over elements of table ***tab*** from the beginning forwards and the other, ***k***, from the last element backwards; the loop terminates when these indices "cross" each other. In this way the first and the last elements are exchanged, then the second with the last but one, and so forth. In the ***init*** part, we declared also an auxiliary variable ***aux***, which is used when exchanging the values of any two elements. The result of this program is

```
[ 1 3 5 7 2 4 -9 12 ]
[ 12 -9 4 2 7 5 3 1 ]
```

8.9.4 *Foreach* loop

The C++11 standard introduced another form of looping, the so called „foreach" loop. It can be used to iterate over a collection from the standard library (for example **std::array**, but is also applicable to normal static arrays — under the condition that in a given scope the array is visible as having the array type (and not pointer type, as is the case after passing it to a function). Let us look at the syntax:

P51: foreach.cpp "Foreach" loop

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int arr[] = {1,2,3,4,9,8,7,6};
6.
7.      for (int e  : arr) cout << e << " ";    ❶
8.      cout << endl;
9.      for (int& e : arr) e -= 1;              ❷
10.     for (auto e : arr) cout << e << " ";    ❸
11.     cout << endl;
12. }
```

On line ❶ the variable ***e*** will be initialized in each iteration of the loop with an unmodifiable *copy* of subsequent element of the array ***arr***. On line ❷, however, in each iteration ***e*** will be the *reference* to subsequent element of the array, so elements can be modified (here we just subtract 1 from each element). As we can see on line ❸, we do not have to specify the type of the elements explicitly — the type can be deduced from the type of the array (more generally, of the collection). The same applies to references: on line ❷, we could have used `auto&` instead of `int&`. The output of the program is

```
1 2 3 4 9 8 7 6
0 1 2 3 8 7 6 5
```

It is also possible to iterate over a collection created *ad hoc* — as surrounded in braces sequence of values of the same type, as in the program

P52: adhocforeach.cpp Loop over *ad hoc* created collection

```
1.  #include <iostream>
2.
3.  long long factorial(int n) {
4.      return n < 2 ? 1 : n*factorial(n-1);
5.  }
6.
7.  int main() {
8.      for (auto n : {12, 14, 16, 18, 20}) {
9.          if (auto f = factorial(n); f > 1e17)
10.             std::cout << f << " is really big\n";
11.         else
12.             std::cout << f << " is not so big\n";
13.     }
14. }
```

which prints

```
479001600 is not so big
87178291200 is not so big
20922789888000 is not so big
6402373705728000 is not so big
2432902008176640000 is really big
```

Prev: 8.8 Selection (switch)statement (switch) **Up:** 8. Statements **Next:** 8.10 continue and break statements

T.R. Werner, September 27, 2023; 22:32

Subsections

- 8.9.1 *while* loop
- 8.9.2 *do-while* loop
- 8.9.3 *for* loop
- 8.9.4 *Foreach* loop

8.9 Iteration statements (loops)

Iteration statements (or **loops**) are used when the repetition of a portion of the code is needed. The same fragment of code is executed cyclically as long as some logical condition holds true. It is the responsibility of the programmer to ensure that this condition will eventually become false so the loop can terminate. Another way to terminate the execution of a loop is by using **return**, **break** or **goto** statements.

There are three basic forms of iteration statements. Any of them can be easily transformed to any other, so programmers can choose a form best suited for a particular purpose at hand.

In each of these forms, loops can be modified or terminated by instructions **break** or **continue**.

8.9.1 *while* loop

The **while** loop has the form

```
while ( b ) stmt
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. The syntax says about *one* statement: if more are required, they have to be “packed” into one compound statement by enclosing them in braces (see the section on compound statements).

Execution of the **while** statement proceeds in a cyclic way as follows:

- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **while** statement terminates and the flow of control passes to the next statement after the **while** statement (***stmt*** is *not* executed);
- if ***b*** is **true** the statement ***stmt*** is executed and the flow of control passes again to the beginning of the **while** statement.

For example, the following snippet will find (in ***num***) the first natural number of the form 2^n which is larger than a given natural number ***lim***:

```
int numb = 1;
while ( numb <= lim ) numb *= 2;
```

while this code

```
int age = 0;
while ( age < 5 || age > 100) cin >> age;
```

will read from the standard input an **int** until a number from the range $[5, 100]$ is entered.

8.9.2 *do-while* loop

The **do** (or **do-while**) loop has the form

```
do stmt while ( b )
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. Again, the syntax says about *one* statement, so if more are required, they have to be gathered into one compound statement.

The execution goes (cyclically) as follows:

- the statement ***stmt*** is executed;
- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **do-while** statement terminates and the flow of control passes to the next statement after the **do-while** statement;
- if ***b*** is **true** the flow of control passes again to the beginning of the **do-while** statement.

In the **do...while** statement the statement ***stmt*** is executed first and then the termination condition ***b*** is checked; in the **while** loop the order was opposite: first the termination condition was checked, and then ***stmt*** was executed. Therefore, in the **do-while** loop, ***stmt*** will *always* be executed at least once, whereas in the **while** loop it is possible that ***stmt*** is not executed at all (when ***b*** is **false** right from the beginning).

The program below simulates a series of rolls of two dice. The outcome of each roll is obtained with the help of pseudo-random number generator (**rand** and initialization with **srand** from the header **<cstdlib>**). The game ends when we get six spots on both dice:

P49: dice.cpp *do...while loop*

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  using namespace std;
4.
5.  int main() {
6.      int x, y, roll = 0;
7.
8.      srand(unsigned(time(0)));
9.
10.     do {
11.         x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
12.         y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
13.         cout << "Roll no " << ++roll   << ": ("
14.              <<  x << ", " << y << ")" << endl;
15.     } while (x + y != 12);
16. }
```

As the number of rolls is not known in advance and there must be at least one roll to end the game, the form **do...while** is the most natural form of loop here. The outcome of the program was

```
Roll no 1: (1, 5)
Roll no 2: (5, 3)
Roll no 3: (6, 2)
Roll no 4: (5, 3)
Roll no 5: (1, 5)
Roll no 6: (3, 2)
Roll no 7: (3, 3)
Roll no 8: (5, 4)
Roll no 9: (3, 2)
Roll no 10: (4, 5)
Roll no 11: (6, 6)
```

Of course, as we use random number generator, the outcome will be different each time we run the program.

8.9.3 *for* loop

The **for** loop has the form

```
for ( init ; b ; incr ) stmt
```

where ***stmt*** is a statement (possibly compound, possibly empty — just a semicolon). The expression ***b*** must have a value convertible to **bool** — this expression can be omitted and then its value defaults to **true**. The expression ***init*** can be

- omitted;
- *one* declaration statement, possibly declaring/defining several variables of the same type with or without initialization;
- any number of comma-separated expression statements (see the section on expression statements).

The “incrementing” part (denoted by ***incr*** here) can be omitted or be a comma-separated list of any number of expression statements.

Even if one or more parts, ***init***, ***b*** or ***incr*** is omitted, the semicolons and parentheses must be present.

The execution of the **for** loop proceeds as follows;

1. If ***init*** is not empty, it is executed. If it is a sequence of expression statements, they are executed from left to right and their values are ignored. If it is a declaration, the scope of the variables declared will contain the ***b*** and ***incr*** parts and the body of the loop (the statement ***stmt***, which can be a compund statement) — they will not exist after the termination of the loop. Statements in ***init*** are executed always, but once only: when the flow of control enters the loop.
2. The value of ***b*** is evaluated and converted to type **bool** (if ***b*** is empty, **true** is assumed). If the value of ***b*** is **false**, the loop terminates.
3. The body of the loop (statement ***stmt***) is executed.
4. If ***incr*** is not empty, expression statements from this part are executed and their values are ignored.
5. Flow of control returns to item 2.

It is not possible to declare in the ***init*** part variables of different type: that would require more than one declaration statements:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
    // ... WRONG !!!
}
```

It is, however, possible to declare more than one variables of the same type:

```
for (int i = 0, k = size-1; i < k; i++, k--) {
    // OK
}
```

The **for** loop is the most general form of loop. It is particularly well suited in cases where one knows in advance how many iterations will be needed. Usually we initialize values of some kind of "counters" in the ***init*** part and increment (or decrement) them in the ***incr*** part.

As an example let us consider a simple program where function **reverse** reverses the order of elements in an array of integers:

P50: revers.cpp *for loop*

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void reverse(int *arr, int size) {
5.      if ( size < 2 ) return;
6.
7.      for (int i = 0, k = size-1, aux; i < k; i++, k--) { ❶
8.          aux    = arr[i];
9.          arr[i] = arr[k];
10.         arr[k] = aux;
11.     }
12. }
13.
14. void printArr(int *arr, int size) {
15.     cout << "[ ";
16.     for (int i = 0; i < size; i++)
17.         cout << arr[i] << " ";
18.     cout << "]" << endl;
19. }
20.
21. int main() {
22.     int arr[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
23.     int size = sizeof(arr)/sizeof(arr[0]);
24.
25.     printArr(arr,size);
26.     reverse (arr,size);
27.     printArr(arr,size);
28. }
```

The loop on line ❶ uses two indices — one, ***i***, runs over elements of table ***tab*** from the beginning forwards and the other, ***k***, from the last element backwards; the loop terminates when these indices "cross" each other. In this way the first and the last elements are exchanged, then the second with the last but one, and so forth. In the ***init*** part, we declared also an auxiliary variable ***aux***, which is used when exchanging the values of any two elements. The result of this program is

```
[ 1 3 5 7 2 4 -9 12 ]
[ 12 -9 4 2 7 5 3 1 ]
```

8.9.4 *Foreach* loop

The C++11 standard introduced another form of looping, the so called „foreach" loop. It can be used to iterate over a collection from the standard library (for example **std::array**, but is also applicable to normal static arrays — under the condition that in a given scope the array is visible as having the array type (and not pointer type, as is the case after passing it to a function). Let us look at the syntax:

P51: foreach.cpp "Foreach" loop

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int arr[] = {1,2,3,4,9,8,7,6};
6.
7.      for (int e  : arr) cout << e << " ";    ❶
8.      cout << endl;
9.      for (int& e : arr) e -= 1;              ❷
10.     for (auto e : arr) cout << e << " ";    ❸
11.     cout << endl;
12. }
```

On line ❶ the variable ***e*** will be initialized in each iteration of the loop with an unmodifiable *copy* of subsequent element of the array ***arr***. On line ❷, however, in each iteration ***e*** will be the *reference* to subsequent element of the array, so elements can be modified (here we just subtract 1 from each element). As we can see on line ❸, we do not have to specify the type of the elements explicitly — the type can be deduced from the type of the array (more generally, of the collection). The same applies to references: on line ❷, we could have used `auto&` instead of `int&`. The output of the program is

```
1 2 3 4 9 8 7 6
0 1 2 3 8 7 6 5
```

It is also possible to iterate over a collection created *ad hoc* — as surrounded in braces sequence of values of the same type, as in the program

P52: adhocforeach.cpp Loop over *ad hoc* created collection

```
1.  #include <iostream>
2.
3.  long long factorial(int n) {
4.      return n < 2 ? 1 : n*factorial(n-1);
5.  }
6.
7.  int main() {
8.      for (auto n : {12, 14, 16, 18, 20}) {
9.          if (auto f = factorial(n); f > 1e17)
10.             std::cout << f << " is really big\n";
11.         else
12.             std::cout << f << " is not so big\n";
13.     }
14. }
```

which prints

```
479001600 is not so big
87178291200 is not so big
20922789888000 is not so big
6402373705728000 is not so big
2432902008176640000 is really big
```

Prev: 8.8 Selection (switch)statement (switch) **Up:** 8. Statements **Next:** 8.10 continue and break statements

T.R. Werner, September 27, 2023; 22:32

Subsections

- 8.9.1 *while* loop
- 8.9.2 *do-while* loop
- 8.9.3 *for* loop
- 8.9.4 *Foreach* loop

8.9 Iteration statements (loops)

Iteration statements (or **loops**) are used when the repetition of a portion of the code is needed. The same fragment of code is executed cyclically as long as some logical condition holds true. It is the responsibility of the programmer to ensure that this condition will eventually become false so the loop can terminate. Another way to terminate the execution of a loop is by using **return**, **break** or **goto** statements.

There are three basic forms of iteration statements. Any of them can be easily transformed to any other, so programmers can choose a form best suited for a particular purpose at hand.

In each of these forms, loops can be modified or terminated by instructions **break** or **continue**.

8.9.1 *while* loop

The **while** loop has the form

```
while ( b ) stmt
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. The syntax says about *one* statement: if more are required, they have to be “packed” into one compound statement by enclosing them in braces (see the section on compound statements).

Execution of the **while** statement proceeds in a cyclic way as follows:

- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **while** statement terminates and the flow of control passes to the next statement after the **while** statement (***stmt*** is *not* executed);
- if ***b*** is **true** the statement ***stmt*** is executed and the flow of control passes again to the beginning of the **while** statement.

For example, the following snippet will find (in ***num***) the first natural number of the form 2^n which is larger than a given natural number ***lim***:

```
int numb = 1;
while ( numb <= lim ) numb *= 2;
```

while this code

```
int age = 0;
while ( age < 5 || age > 100) cin >> age;
```

will read from the standard input an **int** until a number from the range $[5, 100]$ is entered.

8.9.2 *do-while* loop

The **do** (or **do-while**) loop has the form

```
do stmt while ( b )
```

where ***b*** is an expression which has a logical value and ***stmt*** is a statement. Again, the syntax says about *one* statement, so if more are required, they have to be gathered into one compound statement.

The execution goes (cyclically) as follows:

- the statement ***stmt*** is executed;
- the value of ***b*** is evaluated and converted to type **bool**;
- if ***b*** is **false**, then the **do-while** statement terminates and the flow of control passes to the next statement after the **do-while** statement;
- if ***b*** is **true** the flow of control passes again to the beginning of the **do-while** statement.

In the **do...while** statement the statement ***stmt*** is executed first and then the termination condition ***b*** is checked; in the **while** loop the order was opposite: first the termination condition was checked, and then ***stmt*** was executed. Therefore, in the **do-while** loop, ***stmt*** will *always* be executed at least once, whereas in the **while** loop it is possible that ***stmt*** is not executed at all (when ***b*** is **false** right from the beginning).

The program below simulates a series of rolls of two dice. The outcome of each roll is obtained with the help of pseudo-random number generator (**rand** and initialization with **srand** from the header **<cstdlib>**). The game ends when we get six spots on both dice:

P49: dice.cpp *do...while loop*

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  using namespace std;
4.
5.  int main() {
6.      int x, y, roll = 0;
7.
8.      srand(unsigned(time(0)));
9.
10.     do {
11.         x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
12.         y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
13.         cout << "Roll no " << ++roll   << ": ("
14.              <<  x << ", " << y << ")" << endl;
15.     } while (x + y != 12);
16. }
```

As the number of rolls is not known in advance and there must be at least one roll to end the game, the form **do...while** is the most natural form of loop here. The outcome of the program was

```
Roll no 1: (1, 5)
Roll no 2: (5, 3)
Roll no 3: (6, 2)
Roll no 4: (5, 3)
Roll no 5: (1, 5)
Roll no 6: (3, 2)
Roll no 7: (3, 3)
Roll no 8: (5, 4)
Roll no 9: (3, 2)
Roll no 10: (4, 5)
Roll no 11: (6, 6)
```

Of course, as we use random number generator, the outcome will be different each time we run the program.

8.9.3 *for* loop

The **for** loop has the form

```
for ( init ; b ; incr ) stmt
```

where ***stmt*** is a statement (possibly compound, possibly empty — just a semicolon). The expression ***b*** must have a value convertible to **bool** — this expression can be omitted and then its value defaults to **true**. The expression ***init*** can be

- omitted;
- *one* declaration statement, possibly declaring/defining several variables of the same type with or without initialization;
- any number of comma-separated expression statements (see the section on expression statements).

The “incrementing” part (denoted by ***incr*** here) can be omitted or be a comma-separated list of any number of expression statements.

Even if one or more parts, ***init***, ***b*** or ***incr*** is omitted, the semicolons and parentheses must be present.

The execution of the **for** loop proceeds as follows;

1. If ***init*** is not empty, it is executed. If it is a sequence of expression statements, they are executed from left to right and their values are ignored. If it is a declaration, the scope of the variables declared will contain the ***b*** and ***incr*** parts and the body of the loop (the statement ***stmt***, which can be a compund statement) — they will not exist after the termination of the loop. Statements in ***init*** are executed always, but once only: when the flow of control enters the loop.
2. The value of ***b*** is evaluated and converted to type **bool** (if ***b*** is empty, **true** is assumed). If the value of ***b*** is **false**, the loop terminates.
3. The body of the loop (statement ***stmt***) is executed.
4. If ***incr*** is not empty, expression statements from this part are executed and their values are ignored.
5. Flow of control returns to item 2.

It is not possible to declare in the ***init*** part variables of different type: that would require more than one declaration statements:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
    // ... WRONG !!!
}
```

It is, however, possible to declare more than one variables of the same type:

```
for (int i = 0, k = size-1; i < k; i++, k--) {
    // OK
}
```

The **for** loop is the most general form of loop. It is particularly well suited in cases where one knows in advance how many iterations will be needed. Usually we initialize values of some kind of "counters" in the ***init*** part and increment (or decrement) them in the ***incr*** part.

As an example let us consider a simple program where function **reverse** reverses the order of elements in an array of integers:

P50: revers.cpp *for loop*

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void reverse(int *arr, int size) {
5.      if ( size < 2 ) return;
6.
7.      for (int i = 0, k = size-1, aux; i < k; i++, k--) { ❶
8.          aux    = arr[i];
9.          arr[i] = arr[k];
10.         arr[k] = aux;
11.     }
12. }
13.
14. void printArr(int *arr, int size) {
15.     cout << "[ ";
16.     for (int i = 0; i < size; i++)
17.         cout << arr[i] << " ";
18.     cout << "]" << endl;
19. }
20.
21. int main() {
22.     int arr[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
23.     int size = sizeof(arr)/sizeof(arr[0]);
24.
25.     printArr(arr,size);
26.     reverse (arr,size);
27.     printArr(arr,size);
28. }
```

The loop on line ❶ uses two indices — one, ***i***, runs over elements of table ***tab*** from the beginning forwards and the other, ***k***, from the last element backwards; the loop terminates when these indices "cross" each other. In this way the first and the last elements are exchanged, then the second with the last but one, and so forth. In the ***init*** part, we declared also an auxiliary variable ***aux***, which is used when exchanging the values of any two elements. The result of this program is

```
[ 1 3 5 7 2 4 -9 12 ]
[ 12 -9 4 2 7 5 3 1 ]
```

8.9.4 *Foreach* loop

The C++11 standard introduced another form of looping, the so called „foreach" loop. It can be used to iterate over a collection from the standard library (for example **std::array**, but is also applicable to normal static arrays — under the condition that in a given scope the array is visible as having the array type (and not pointer type, as is the case after passing it to a function). Let us look at the syntax:

P51: foreach.cpp "Foreach" loop

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int arr[] = {1,2,3,4,9,8,7,6};
6.
7.      for (int e  : arr) cout << e << " ";    ❶
8.      cout << endl;
9.      for (int& e : arr) e -= 1;              ❷
10.     for (auto e : arr) cout << e << " ";    ❸
11.     cout << endl;
12. }
```

On line ❶ the variable ***e*** will be initialized in each iteration of the loop with an unmodifiable *copy* of subsequent element of the array ***arr***. On line ❷, however, in each iteration ***e*** will be the *reference* to subsequent element of the array, so elements can be modified (here we just subtract 1 from each element). As we can see on line ❸, we do not have to specify the type of the elements explicitly — the type can be deduced from the type of the array (more generally, of the collection). The same applies to references: on line ❷, we could have used `auto&` instead of `int&`. The output of the program is

```
1 2 3 4 9 8 7 6
0 1 2 3 8 7 6 5
```

It is also possible to iterate over a collection created *ad hoc* — as surrounded in braces sequence of values of the same type, as in the program

P52: adhocforeach.cpp Loop over *ad hoc* created collection

```
1.  #include <iostream>
2.
3.  long long factorial(int n) {
4.      return n < 2 ? 1 : n*factorial(n-1);
5.  }
6.
7.  int main() {
8.      for (auto n : {12, 14, 16, 18, 20}) {
9.          if (auto f = factorial(n); f > 1e17)
10.             std::cout << f << " is really big\n";
11.         else
12.             std::cout << f << " is not so big\n";
13.     }
14. }
```

which prints

```
479001600 is not so big
87178291200 is not so big
20922789888000 is not so big
6402373705728000 is not so big
2432902008176640000 is really big
```

Prev: 8.8 Selection (switch)statement (switch) **Up:** 8. Statements **Next:** 8.10 continue and break statements

T.R. Werner, September 27, 2023; 22:32

8.10 *continue* and *break* statements

These two statements are used inside a loop of any type: **for**, **do...while** or **while** (the **break** statement can also be used inside the **switch** statement).

The **continue** statement enforces termination of the current iteration of the loop it is nested in. Then the loop continues normally. If it is a **for** loop, the ***incr*** part (see above) will be executed as if the flow of control reached the end of the body of the loop. The rule applies only to the narrowest loop embracing the statement: it is not possible to terminate the current cycle of any outer loop in this way. In other words, after encountering **continue** statement, the flow of controls behaves as if all remaining statements of the body of the innermost loop were substituted by an empty statement.

In the following program, the function **sumPos** sums up all positive elements of an array (the result is 17); negative elements are skipped due to the **continue** statement on line ❶:

P53: sumpos.cpp *continue* statement

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int sumPos(int *arr, int size) {
5.      int sum = 0;
6.      for (int i = 0; i < size; i++) {
7.          if ( arr[i] <= 0 ) continue;    ❶
8.          sum += arr[i];
9.      }
10.     return sum;
11. }
12.
13. int main() {
14.     int arr[] = { 1, -3, 5, -7, 2, 0, 9 };
15.     int sum = sumPos(arr,sizeof(arr)/sizeof(arr[0]));
16.     cout << "Sum: " << sum << endl;
17. }
```

The **break** statement terminates the narrowest loop it is nested in. The flow of control passes to the statement after the loop. Modifying the previous program, we can now write a function which sums up all positive elements of an array until a non-positive element is encountered (the result is 16); all remaining elements are ignored:

P54: sumuntil.cpp *break* statement

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int sumuntil(int *arr, int size) {
5.      int sum = 0;
6.      for (int i = 0; i < size; i++) {
```

```
7.          if ( arr[i] <= 0 ) break;
8.          sum += arr[i];
9.      }
10.     return sum;
11. }
12.
13. int main() {
14.     int arr[] = { 1, 3, 5, 7, 0, 4, 9 };
15.     int sum = sumuntil(arr, sizeof(arr)/sizeof(arr[0]));
16.     cout << "Sum: " << sum << endl;
17. }
```

On line ❶ the loop is terminated when a non-positive element is encountered.

The **break** statement can also be used inside **switch**, where it terminates the execution of the whole **switch** statement (see the section on switch statement). Of course, **continue** would not have any sense in a **switch**, as this is not an iteration statement.

Those who know Java remember that in this language both **break** and **continue** statements have the labeled form, so it is possible to break from or continue an outer loop as well. There is no such construct in C/C++.

Prev: 8.9 Iteration statements (loops) **Up:** 8. Statements **Next:** 8.11 goto statement

T.R. Wemer, September 27, 2023; 22:32

8.11 *goto* statement

The “jump” statement, **goto**, transfers the flow of control to another place of the code. It has the form

```
goto label;
```

where ***label*** is a label of a statement in the same function. After execution of the **goto** the flow of control will pass to this statement.

The language allows for even somewhat strange jumps (it is illegal, however, to jump into a block in such a way that some declarations with initializations are skipped). This flexibility should not be abused. Actually, it is believed that **goto** statement should be avoided altogether — it makes the program harder to understand and maintain.

A construct that is quite often used and which uses **goto** statement is the **goto** which “jumps out” from nested loops. Due to absence of labeled **break** and **continue** statements, this is the simplest way to do it. Let us look at an example:

P55: students.cpp *goto statement*

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const int st_size = 5;
6.
7.      char grades[][st_size] = {{ 'A', 'A', 'B', 'C', 'B' },
8.                                { 'A', 'C', 'C', 'F', 'D' },
9.                                { 'A', 'C', 'B', 'B', 'A' }};
10.
11.     int gr_size = sizeof(grades)/sizeof(grades[0]);
12.
13.     bool isF = false;
14.
15.     for (int group = 0; group < gr_size; group++)      ❶
16.         for (int student = 0; student < st_size; student++)
17.             if ( grades[group][student] == 'F' ) {
18.                 isF = true;
19.                 goto THEEND;
20.             }
21. THEEND:                                                ❷
22.     if (isF) cout << "There was an \'F\'" << endl;
23.     else     cout << "There was no \'F\'" << endl;
24. }
```

In an nested loop starting on line ❶, we look for an 'F' note in a two-dimensional array of notes indexed with group number and student number. We are only interested whether there is a student with at least one 'F' note. If we encounter an 'F', we want to break from both loops. The **break** statement would break only from the inner loop over students; the loop over groups would be continued. That is why we used **goto**, which jumps out of both loops, to line ❷.

8.12 *return* statement

The **return** statement terminates the execution of the function it was encountered in. The flow of control returns to the calling function, or the whole program terminates, if **return** was encountered in the **main** function. If the function was declared as **void**, a **return** is assumed as the last statement (just before the closing brace which marks the end of function's body). There is one exception: the **main** function is not a **void** function (it must return an integer) but still **return** can be omitted: '`return 0`' is assumed. If a function is not **void** (and not **main**), it must return a value, so the last statement executed *must* be **return** (the other possibility is to throw an exception):

```
return val;
```

where ***val*** is an expression the value of which is of the type declared as the return type of the function (or there is a standard conversion leading from the type of ***val*** the type of the function). The value returned by the **return** statement (possibly after a conversion) is then used to initialize the function's result which becomes accessible in the calling function.

8.13 Exception handling statements

There are three statements connected with exception handling:

- **try** statement which declares a portion of the code as possible source of exceptions;
- **catch** statement which defines a block of code where exeptions are handled;
- **throw** statement which permits the user to throw exceptions “by hand”.

We will postpone the discussion of exceptions to the chapter on exceptions .

Prev: 8.12 return statement **Up:** 8. Statements **Next:** 9. Operators

9. Operators

In this chapter, we will describe operators appearing in the C++ language. More detailed discussion of some of them will be deferred to subsequent chapters. **Operators** are non-alphanumeric tokens appearing in the text of programs (plus sign, asterisk, per cent sign, etc.) which are interpreted as a request to invoke appropriate functions operating on data defined by expressions adjacent to the tokens-operators. The data an operator acts on are called its **arguments** (or **operands**).

Generally, operators fall into two categories: operators with one argument (**unary operator**) and with two arguments (**binary operator**). As in most (but not all) other languages, for binary operators we use **infix** notation: they are placed *between* their arguments. For unary operators, the **prefix** notation is used (with some exceptions): they act on expressions following them. There is also one **ternary operator**: the conditional operator which has three arguments.

Subsections

- 9.1 Precedence and associativity of operators
- 9.2 Overview of operators
  - 9.2.1 Scope-resolution operators
  - 9.2.2 Operators of precedence 15
  - 9.2.3 Operators of precedence 14
  - 9.2.4 Operators of precedence 13
  - 9.2.5 Arithmetic operators
  - 9.2.6 Relational operators and comparisons
  - 9.2.7 Bitwise operators
  - 9.2.8 Logical operators
  - 9.2.9 Assignment operators
  - 9.2.10 Conditional operator
  - 9.2.11 Exception-throw operator
  - 9.2.12 Comma operator
  - 9.2.13 Alternative operator names

9.1 Precedence and associativity of operators

The infix notation we are accustomed to seems natural for humans but in fact leads to some ambiguities which must be resolved. For example in

```
a + b / c
```

the variable ***b*** could be considered the left argument od division operator (slash) or the right argument of addition. In the first case we would calculate $a \cdot (b/c)$ and in the second case $(a + b)/c$ — the result will be, of course, different.

To solve this problem we could require that parentheses be used in such cases. This certainly always works, but would be very inconvenient. Therefore, as in mathematics, we attach to each operator its **precedence** (**priority**). In situations like the one described above, if an expression can be considered as an argument of two operators, the one with higher precedence will be performed first (in our example it would be division, which has higher priority than addition). If precedence of both operators is the same (as is the case for addition and subtraction), then they will be executed in the order determined by their **associativity**: from left to right for left-associative operators and from right to left for right-associative operators. For this rule to work, operators with the same precedence must have the same associativity (which is the case). Therefore in

```
a + b / c
```

division will be performed first, as it has higher precedence than addition, while in

```
a + b - c
```

addition will be performed first, as addition and subtraction have the same precedence and both are left-associative. The assignment operator is right-associative. Therefore, in the expression `a=b=c` the assignment `b=c` will be performed first and its result (which is the value of ***b*** after the operation) will be assigned to ***a***. But, e.g., the operator of direct selection (a dot) is left associative. Consequently, `object.memb1.memb2` means: *first select the member **memb1** of object **object** and then, from the resulting object, select the member **memb2**.*

All prefix unary operators are right-associative: the right operator, closer to the argument, acts first. Therefore, `*++p` is equivalent to `*(++p)`, while `++*p` to `++(*p)` because both operators, '`++`' and unary '`*`', are right associative.

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

Subsections

- 9.2.1 Scope-resolution operators
- 9.2.2 Operators of precedence 15
- 9.2.3 Operators of precedence 14
- 9.2.4 Operators of precedence 13
- 9.2.5 Arithmetic operators
- 9.2.6 Relational operators and comparisons
- 9.2.7 Bitwise operators
- 9.2.8 Logical operators
- 9.2.9 Assignment operators
- 9.2.10 Conditional operator
- 9.2.11 Exception-throw operator
- 9.2.12 Comma operator
- 9.2.13 Alternative operator names

9.2 Overview of operators

The table below presents operators in C++. The meaning of abbreviations used in its rightmost column is as follows:

class: name of a class	*obj*: object
memb: member of a class or namespace	*ptr*: pointer
expr: expression	*lval*: l-value
nsname: name of a namespace	*type*: name of a type
name: name (identifier)	

All operator are divided into 18 groups — each group includes operators of the same precedence. Groups are ordered from the highest precedence to the lowest. Associativity is indicated in the first column: 'R' for right and 'L' for left.

L/R	Function	Use
Precedence 16		
L	scope resolution	class::memb,nsname::memb,::name
Precedence 15		
L	member access	ob.memb,ptr->memb
L	subscripting	expr[expr]
L	function call	expr(list_expr)
L	value construction	type(list_expr), type{list_expr}
L	postfix decrement, increment	lval- -,lval++
Precedence 14		
R	object sizeof operator	**sizeof** expr
R	type sizeof operator	**sizeof**(type)

R	package sizeof operator	**sizeof...**(name)
R	prefix decrement,increment	- -lval,++lval
R	bitwise, logical NOT	~expr,!expr
R	unary minus, plus	-expr, +expr
R	address-of operator	&lval
R	indirection (dereference)	*ptr
R	object allocation	**new** type
R	array allocation	**new** type[expr]
R	object deallocation	**delete** expr
R	array deallocation	**delete** [] expr
R	C-like cast	(type)expr
Precedence 13		
L	pointer to member	ptr->*memb_ptr,obj.*memb_ptr
Precedence 12		
L	multiplication,division	expr*expr,expr/expr
L	remainder (modulo)	expr % expr
Precedence 11		
L	addition,subtraction	expr+expr,expr-expr
Precedence 10		
L	left, right shift	expr<<expr,expr>>expr
Precedence 9		
L	less than	expr < expr
L	less than or equal	expr <= expr
L	greater than	expr > expr
L	greater than or equal	expr >= expr
Precedence 8		
L	equality	expr == expr
L	nonequality	expr != expr
Precedence 7		
L	bitwise AND	expr&expr
Precedence 6		
L	bitwise XOR	expr^expr
Precedence 5		
L	bitwise OR	expr | expr
Precedence 4		
L	logical AND	expr && expr
Precedence 3		
L	logical OR	expr | | expr
Precedence 2		
R	conditional operator	expr ? expr : expr
R	assignment	lval = expr
R	assignment with addition	lval += expr
R	assignment with subtraction	lval -= expr
R	assignment with multiplication	lval *= expr
R	assignment with division	lval /= expr
R	assignment with remainder	lval %= expr
R	assignment with left shift	lval <<= expr
R	assignment with right shift	lval >>= expr

R	assignment with bitwise AND	lval &= expr
R	assignment with bitwise OR	lval |= expr
R	assignment with bitwise XOR	lval ^= expr
R	exception throwing	**throw** expr
Precedence 1		
L	comma operator	expr,expr

Table 9.1: Operators in C++

Operators **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** and **alignof** are not mentioned above, as they are never ambiguous.

Some of these operator are related to classes, namespaces, conversions, exceptions; we will discuss them in subsequent chapters.

9.2.1 Scope-resolution operators

The **scope-resolution operators** have the highest precedence (16 in the table). They are denoted by double-colon ('::'). We already know the global scope resolution operator (see section on scopes): notation ***::x*** refers to a global variable named ***x***. Using the double-colon symbol is only needed if a global name, say ***x***, is shadowed by the name of another variable declared in a narrower scope (in a function or block).

The same symbol, '::', is used as the class scope resolution operator. In this way we can access names declared inside class declarations (names of static methods, enumerations, inner classes and unions, etc.). More details can be found in chapter on classes . Similarly, using '::' we can access names declared in namespaces (for details, see chapter on namespaces).

9.2.2 Operators of precedence 15

The first two, “dot” and “arrow” operators, are used with objects of structures and classes: we will come to them in chapter on structures .

The subscript (or indexing) operator ' `[]`', is already known to us (see section on pointer aruthmetic).

The function call operator is denoted by round parentheses ' `()`':

```
func(k, m, 5)
```

A name followed by parentheses triggers a function call if it appears in an executable statement (not in declaration/definition). The name does not have to be the name of a function — it can also be (in C++) the name of an object (more about it in the chapter on the Standard Library). The name of a function without parentheses denotes the pointer to this function (see section on function pointers).

The **value construction operator** (`Type(list_expr)`, `Typ{list_expr}`) is not known to us yet. It is connected with classes and will be discussed in section on object creation . Let us mention here that in C++ it is possible to create objects of primitive types (**int**, **double**, ...) as if they were objects of a class. For example, `int(3)` creates an **int** initialized with the value 3 — the syntax is exactly the same as if we would use to create an object of class **int** passing the value 3 to its copy-constructor.

The **postfix decrement** and **postfix increment** operators decrement and increment, respectively, their argument by one. They are placed *after* the argument. The change is effectuated *after* the expression they are part of has been evaluated. Therefore, after

```
int a = 1;
int b = a++;
```

the value of ***a*** is 2, but the value of ***b*** is 1, because ***a*** was increased *after* the assignment had taken place. Postdecrementation and postincrementation must act on l-values. An expression like

```
(x+y)++
```

does not make sense, as `(x+y)` is not an l-value. Therefore, the *argument* of postdecrementation or postincrementation must be an l-value: but what we get as the result? In Java the result of both post- and predecrementation is never an l-value (the same applies to post- and preincrementation); in C/C++ it is more complicated: postfix operators do not produce l-values, but the corresponding prefix operators *do*. That is why

```
int k = 5;
int m = (++k)--;
```

is correct: the expression in parentheses is an l-value (being the result of *pre*incrementation), so one can now apply the postdecrementation. The result of this operation is *not* an l-value, but it does not have to be one, as it appears on the right-hand side of the assignment. After the statement has been executed, the value of ***k*** will be 5, and ***m*** will be 6. Without parentheses

```
int k = 5;
int m = ++k--; // WRONG !!!
```

the code would be in error. The precedence of postdecrementation is higher than that of preincrementation, so the expression `k-` would be executed first yielding a non-l-value, which cannot be the argument of preincrementation — see also section on l-values .

An expression like `++w` is *almost* equivalent to `w=w+1`, but not exactly. The latter form implies evaluation of the right-hand side (to get a value) and then of the left-hand side (to get a location where the result should go). If ***w*** is a more complex expression, like a function call, it will be evaluated twice. If, however, the form `++w` has been used, the ***w*** will be evaluated once only. In rare situations this can make a difference.

The **type identification operator typeid** (not mentioned in the table) allows for obtaining the type identifier of a type (statically, at compile time) or of an object (dynamically, at run time). This mechanism is called RTTI — *run-time type identification*. We will have a look at

it in more detail in the chapter on RTTI , but an example will be presented below (in the file sizes.cpp).

The **cast** (**conversion**) operators (also not mentioned in the table): **const_cast**, **static_cast**, **dynamic_cast** and **reinterpret_cast**, allow for conversions of values of one type into values of other types. The names of these operator are so awkward intentionally, as casting can often be quite dangerous; its is therefore better to see clearly where it is. We will discuss the details in the special chapter .

9.2.3 Operators of precedence 14

The first three of the operator from this group, **sizeof** operators, provide the size of an object or of objects of a given type. The result is of type **size_t** (which is an alias, defined by **typedef**, of an unsigned integer type, usually **unsigned long**). As the argument, we can specify the name of a type (in parentheses) or an expresssion with a value of some type (parentheses can be then left out) or a so called package (in the last case the operator has three dots at the end: **sizeof...**). The example below also illustrates the **typeid** operator:

P56: sizes.cpp *sizeof operator*

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  typedef int ARRINT15[15];                                  ❶
6.
7.  void siz(ARRINT15 t1, ARRINT15& t2) {                      ❷
8.     cout << "G. t1 in siz: " << sizeof t1 << endl;
9.     cout << "H. t2 in siz: " << sizeof t2 << endl;
10. }
11.
12. int main() {
13.    ARRINT15 arr1;                                          ❸
14.    int      arr2[15];                                      ❹
15.    int     *arr3 = arr2;                                   ❺
16.
17.    if (typeid(arr1) == typeid(arr2))
18.       cout << "A. arr1 and arr2: same type"      << endl;
19.    else
20.       cout << "A. arr1 and arr2: different type" << endl;
21.
22.    if (typeid(arr2) == typeid(arr3))
23.       cout << "B. arr2 and arr3: same type"      << endl;
24.    else
25.       cout << "B. arr2 and arr3: different type" << endl;
26.
27.    cout << "C. ARRINT15 : " << sizeof(ARRINT15) << endl;❻
28.    cout << "D. arr1     : " << sizeof arr1      << endl;❼
29.    cout << "E. arr2     : " << sizeof(arr2)     << endl;❽
30.    cout << "F. arr3     : " << sizeof arr3      << endl;❾
31.    siz(arr2, arr2);
32. }
```

The program prints:

```
A. arr1 and arr2: same type
B. arr2 and arr3: different type
```

```
C. ARRINT15 : 60
D. arr1     : 60
E. arr2     : 60
F. arr3     : 8
G. t1 in siz: 8
H. t2 in siz: 60
```

This result illustrates some interesting features of the C/C++ language.

First we include the header `typeinfo`, to access tools for type identification (see chapter on RTTI).

On line ❶ we define by **typedef** an alias for the type *15-element array of **int** s* (see the section on typedef specifier).

As we can see from line ❻ of the program and the first line of the printout, the **sizeof** operator found the size of the type **ARRINT15** correctly. The parentheses could not be left out here, as **ARRINT15** is the name (alias, in our case) of a type.

We then define arrays ***arr1*** and ***arr2*** (lines ❸ and ❹) using both the alias **ARRINT15** and the full name of this type. Comparing their types we can convince ourselves that indeed they are the same (line 'A' of the printout).

On line ❺ we define a variable ***arr3*** of type **int*** and we assign it the value of ***arr2***. Thie assignment is legal, but we must remember that it implies a conversion: therefore, as we can see from the line 'B' of the printout, their types are different.

On lines ❻-❾ we print the sizes of the type **ARRINT15** and of objects ***arr1***, ***arr2*** and ***arr3***. Both ***arr1*** and ***arr2*** have size 60 (15 four-byte numbers). The variable ***arr3*** has size 8, as it is just a pointer, not an array.

In the last line we pass *the same* array ***arr2*** through two arguments to the function **siz**. The first parameter was declared as **ARRINT15**, so we would expect that the function “knows” that the corresponding argument is an array. However, printing (line 'G' of the printout) the size of ***t1*** inside the function we get 8 — the value of the argument was converted to a value of the pointer type.

The second argument of function **siz** was declared as as a reference. There is no conversion now! The name ***t2*** in the function is now just another name of the variable used as the argument (***arr2***). Therefore, its type is really **ARRINT15**, i.e., an array type. Consequently, printing the size of ***t2*** we get 60 (line 'H' of the printout).

Prefix decrement and **prefix increment** operators are similar to the corresponding postfix operators. There are important differences, though. The incrementation or decrementation takes place immediately, *before* the value is used for other evaluations. Therefore, after

```
int a = 1;
int b = ++a;
```

the value of ***a*** is 2, but the value of ***b*** is 2 as well, as in the second statement ***a*** was incremented before the assignment. The result of the predecrementation or preincrementation operator *is* an l-value: the expression `++++a` would therefore be perfectly legal.

Operators of logical negation (logical NOT) and bitwise negation (bitwise NOT) will be described later, together with the remaining logical and bit operators.

The unary plus operator, '`+`', is actually the identity operator, i.e., it does nothing (*no-op* operator). It exists just for convenience, to make such construct like '`k = +5`' legal. Operator of address extraction and of indirection (dereference) were already introduced in the chapter on data types .

Operators **new** and **delete** are used to allocate and deallocate memory for objects (variables) and arrays — we will cover this topic in chapter on memory management .

The last of these operators, denoted by a pair of round parentheses, is the **cast** (conversion) **operator**. It belongs to one-argument operators: acting on a p-value it returns another p-value, of a different type, which corresponds, in some sense, to the value of the argument. The target type has to be specified in the parentheses. The cast operator should be used with caution and only when we are absolutely sure that the conversion does make any sense (the compiler will allow for even exotic conversions).

For example, in the second statement of the snippet

```
double x = 7;
int k = (int)x;
```

casting (conversion) of the value of ***x*** to the corresponding value of type **int** is reasonable. We try to “squeeze” a **double** to a narrower type **int** — this implies loosing information, so, although legal, will probably trigger some warning messages of the compiler. Use of explicit cast tells the compiler (and readers of the code) that we do it intentionally — no warnings will be generated.

It is important to remember that casting acts on a p-value and always gives a p-value, never an l-value (of course, this p-value can then be assigned to a variable of an appropriate type). Variables themselves never change their type!

In C++ it is recommended to use more secure *conversion operator*, which are mentioned in our table in the precedence 16 group. We will tell more about them shortly (in the chapter on converions).

9.2.4 Operators of precedence 13

Two operators of memeber selection belong to this group: we will tell more about them in chapter on classes .

9.2.5 Arithmetic operators

The arithmetic operators are familiar to us. They have the obvious meaning, as in most

computer languages. There are sometimes minor differences, however. For example, the remainder operator ('%') can only be applied to integer arguments (in Java it works for **double** s, too). Some ambiguity may arise with negative arguments. Integer division gives always an integer: the fractional part is cut off. For both positive and negative values the result will be truncated "towards zero". It should always be true that, for $b \neq 0$,

```
a =  (a/b)*b + a%b
```

This implies that if truncation of integer division is towards zero, the following rule holds: the absolute value of `a%b` is equal to `|a|%|b|` and its sign is the same as the sign of `a` (vertical bars stand for absolute value). The following program illustrates this:

P57: mod.cpp Remainder operator

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     int    i, j;
6.
7.     i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
8.     i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
9.     i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
10.    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;
11.
12.    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
13.    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
14.    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
15.    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
16. }
```

The result is:

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

The rule would not hold if the truncations were "downwards" (as can be the case for some old compilers). It is therefore better to avoid using the remainder operator for negative arguments.

9.2.6 Relational operators and comparisons

The interpretation of **relational operators** (' <', ' <=', ' >', ' >=') and equality (nonequalty) operators (' ==', ' !=') should also be clear. The value of the expression '***a*** == ***b***' is of **bool** type and answers the question: *is the value of **a** equal to the value of **b***. Similarly, '***a*** != ***b***' answers the question *is the value of **a** different than the value of **b*** and '***a*** < ***b***' the question *is the value of **a** smaller than the value of **b***.

Both arguments can be of any scalar type, including addresses (values of pointer variables or results of the address-of operator). This is different than in Java where references could be compared for identity (' ==' or ' !='), but not with relational operators, like ' <' or ' >'. In

C++ adresses can be compared with relational operators if they correspond to elements of the same array.

All these operators yield logical result: **true** or **false**. As we already know (see section on logical type), logical values are represented by integer values according to the rule:
0 corresponds to **false** and any nonzero value corresponds to **true**. This holds for pointers as well: null value (**nullptr**) is interpreted as **false**, any other value as **true**.

9.2.7 Bitwise operators

Bitwise operators belong to groups with precedence 11 (bitwise shifts), 8, 7 and 6. Arguments must be integer; normal rules of conversion to the common type apply.

The operators are “not interested” in numerical values of arguments but rather in their bit representation. Let us recall that by convention we assign position zero to the least significant bit (the rightmost in standard notation) — see section on integer types .

Let us consider these operators in more detail. The negation is a unary operator, the others are binary.

The bitwise negation (NOT) operator ('~'), acting on a integer value yields a p-value which is of the same type but with all bits negated: every bit which was set (1) in the argument is cleared (0) in the result, and *vice versa*,

w	1	0	0	0	1	1	1	0
$\sim w$	0	1	1	1	0	0	0	1

as shown in the figure, where, for simplicity, we show only 8 bits of a variable. The negation is of course involutive: negating a value twice brings us back to the original value.

Bitwise OR operator!bitwise OR (|) ('|') is a binary infix operator: in the result, each bit is the logical sum of the two bits on the same position in the arguments: if at least one of them is set (1), then the corresponding bit will be set in the result; otherwise, if both were unset, the corresponding bit in the result will be cleared (0), as in the figure below

v	1	0	0	0	1	1	1	0
w	0	0	0	1	0	1	0	1
$v \mid w$	1	0	0	1	1	1	1	1

Bitwise OR is often used to set various options (“ORRing” options). For example, in C/C++, open files have various “modes” which correspond to integer constants: ***ios::in*** — for reading, ***ios::out*** - for writing, etc. In the following program we print the bit representation of some of these constants. As we can see, these are full powers of 2, i.e., in their bit

representation there is only one bit set with all the other bits zeroed. For ***ios::out*** only the bit on position 4 is set, for ***ios::app*** on position 0. Therefore, for example, the constant which describes files open for reading and writing simultaneously will be ***ios::in*** | ***ios::out*** and will have bits on positions 3 and 4 set (this is not guaranteed – one should always refer to these constants by their names.)

Bitwise AND ('&') is also a binary operator: this time the bits in the result are obtained as products of corresponding bits in the arguments. The bit will be set (1) if, and only if, both bits on the corresponding position in the arguments are set (see the figure):

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\&w$	0	0	0	0	0	1	0	1

Bitwise AND is often used for the so called “masking”. For example, let us consider the integer constant determining the mode of an open file. If the fourth bit (on position 3 counting from zero) is set, then the mode is ***in*** (the details will be given in chapter on i/o operations). If the constant determining the mode of file is called ***mode***, then masking (“ANDing”) it with 8 (= 2^3) will answer the question whether the mode ***in*** is set or not. The representation of 8 contains only zeros except for the position 3 (the fourth bit from the right), which is set (1). Therefore, ANDing ***mode*** with 8, we will get a number with all bit set to zero except the fourth, which will be one if the corresponding bit was set in ***mode***. The result will then be nonzero only if mode was ***in***, independently of other bits.

Bitwise XOR operator ('^') is a binary operator like AND and OR. The bit on a given position in the result of this operator is set if, and only if, the corresponding bits in the arguments are different (0 and 1 or 1 and 0).

v	1	0	0	0	1	1	1	1
w	0	0	0	1	0	1	0	1
$v\hat{}w$	1	0	0	1	1	0	1	0

The operation of “XORing” two values has an interesting property which follows from the logical table of this operation:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

We can see that XORing a bit b twice with the same mask m, one always gets the starting value. This property is often used in computer graphics.

The **bitwise shift** operators ('<<' and '>>') act on their left argument, which is an integer value viewed as a series of individual bits, and performs a shift by number of bits given as the right argument — the reaulting value has the same bit pattern as the left argument but shifted to the left ('<<') or to the right ('>>').

w	1	0	0	0	1	1	1	0
$w = w \ll 2$	0	0	1	1	1	0	0	0
$w = w \gg 3$	0	0	0	0	0	1	1	1

In the figure, the upper part corresponds to the bits of the left argument ***w***. After shifting the bits by two to the left, '`w = w << 2`', the bits of the result will look as depicted in the middle portion of the figure. The bits "flowing out" to the left are lost; new bits which enter from the right are always set to zero. If this value is now shifted by three bits to the right ('`w = w >> 3`'), we arrive at the situation depicted in the bottom part of the figure. Now the bits flowing out to the right are lost, and new bits, entering from the left, are set to zero (under some conditions). Shifting to the left is always performed as stated above.

However, there is a complication with the right shifts: it concerns new bits entering from the left. If the type of the left argument is **unsigned** (see section on integer types), bits entering from the left are always set to zero. However, if the type is **signed**, the leftmost bit (sign bit) of the original is reproduced: if it is set, all bits entering from the left will be set, if it is unset, the entering bits will be unset (0) as well. This convention implies that right shift by one bit is equivalent to integer division by 2, regardless of a sign; similarly, left shift corresponds to muliplication by powers of 2 (as long as the leftmost bit is not changed).

The right argument of the shift operators should always be positive and smaller than the size (in bits) of the left operand.

Let us consider an example of bit operations:

P58: bits.cpp Bit operations

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsChar(char k) {
5.      int bits = 8*sizeof(k);                ❶
6.      unsigned char mask = 1<<(bits-1);
7.      for (int i = 0; i < bits; i++) {
8.          cout << (k & mask ? 1 : 0);
9.          mask >>= 1;                        ❷
10.     }
11.     cout << endl;
12. }
13.
14. void bitsShort(short k) {
15.     int bits = 8*sizeof(k);
16.     unsigned short mask = 1<<(bits-1);
17.     for (int i = 0; i < bits; i++) {
18.         cout << (k & mask ? 1 : 0);
19.         mask >>= 1;
20.     }
21.     cout << endl;
22. }
```

```
23.
24. void bitsInt(int k) {
25.     int bits = 8*sizeof(k);
26.     unsigned int mask = 1<<(bits-1);
27.     for (int i = 0; i < bits; i++) {
28.         cout << (k & mask ? 1 : 0);
29.         mask >>= 1;
30.     }
31.     cout << endl;
32. }
33.
34. int main() {
35.     short s = -1; int i = 259;
36.
37.     cout << "char 'a' : "; bitsChar('a');
38.     cout << "short -1 : "; bitsShort(s);
39.     cout << "int  259 : "; bitsInt(i);
40.     cout << endl;
41.     cout << "ios::in  : "; bitsInt(ios::in);
42.     cout << "ios::out : "; bitsInt(ios::out);
43.     cout << "ios::app : "; bitsInt(ios::app);
44.     cout << "ios::in | ios::out\n           ";
45.     bitsInt(ios::in | ios::out);
46. }
```

We define here three almost identical functions, differing only with the type of the argument: it can be **char**, **short** or **int**; in chapter on function templates we will see that such repetition of virtually the same code could have been avoided. It will be sufficient to analyse one of these functions, say **bitsChar**. On line ❶ we check the size (in bits, assuming that one byte occupies 8 bits) of the argument ***k*** (here we know that the result will be 8, as `sizeof(k)` for ***k*** of type **char** yields 1). We then create a mask ***mask*** of type **unsigned char**. The length of the mask should be equal to the length of ***k***, but we prefer it to be **unsigned** to avoid problems with bits entering from the left during right shifts: we will always get zeros. The mask is initialized with '`1 << 7`' (as ***bits*** is 8). We take the value 1, which corresponds to the rightmost bit set and all the others zeroed, and shift it by seven positions to the left; what we get is a bit pattern with 1 on position 7 (the leftmost bit) and 0 on all the others (6,..., 0). The purpose of this operation was to print bits from left to right and not from right to left.

In the loop, we take conjuction (AND) of the variable ***k*** and the mask ***mask***. As the mask has only one nonzero bit, in this way we check if the corresponding bit in ***k*** is also set or not: if it is, the value of '`k&mask`' in the conditional statement on line 9 is nonzero and the character '1' is printed, otherwise the condition is false and '0' is printed. on line ❷ we shift bits in the mask, so in the next cycle of the loop the next bit will be checked. As the mask is of type **unsigned**, we can be sure that new bits, entering the mask from the left, will all be zero, so they will not affect the results of tests.

The other two functions are identical; the only difference is the type of their argument (and of the mask used).

In the main program we just print the bit representation of a few values of integer types, getting

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

One can see that, as explained in chapter on data types , the representation of -1 is composed of all bits set (1). The character 'a' corresponds, as can be easily checked, to the integer value $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, which is the ASCII code of lowercase 'a'.

Then we illustrate what we already know about constants ***ios::in***, ***ios::out*** etc. We can see that they are exact powers of 2, so in their representation there is one and only one bit which is set. When ORing such numbers, we get values with more than one bit set (the last line of the output).

Another example: here we show how to store four numbers form the range $[0, 255]$, so fitting one byte only (like red, green, blue, alpha components of a color) in one variable of 32-bits length. The **encode** function puts color components into one integer by ORing and shifting to the left, while the **decode** does the opposite and prints the individual components:

P59: bitColors.cpp Storing RGBA color in one integer

```
1. #include <iostream>
2. #include <cstdint>
3.
4. std::uint32_t encode(int r, int g, int b, int a) {
5.     std::uint32_t n = a;
6.     n = (n << 8) | b;
7.     n = (n << 8) | g;
8.     n = (n << 8) | r;
9.     return n;
10. }
11.
12. void decode(std::uint32_t n) {
13.     std::cout <<   "r = " << (n & 0xFF);
14.     n >>= 8;
15.     std::cout << ", g = " << (n & 0xFF);
16.     n >>= 8;
17.     std::cout << ", b = " << (n & 0xFF);
18.     n >>= 8;
19.     std::cout << ", a = " << (n & 0xFF) << '\n';
20. }
21.
22. int main() {
23.     decode(encode(23, 44, 129, 255));
24. }
```

The program prints

```
r = 23, g = 44, b = 129, a = 255
```

In the C++20 standard, several new bitwise operations have been added (in the `bit` header); among others, bit rotations: **rotl** and **rotr** functions. They can be applied only to **unsigned** variables and perform rotations to the left (what comes out on the left, comes in from the right) and to the right (what comes out on the right, comes in from the left). For example (this time the **showBits** function is a template):

P60: rotLR.cpp Rotacje bitów

```
1. #include <bit>      // std::rotl, std::rotr
2. #include <cstdint>
3. #include <iostream>
```

```
4.
5. template <typename T>
6. std::string showBits(T t) {
7.     size_t sz = 8*sizeof(T);
8.     std::string s(sz, ' ');
9.     for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10.         s[j] = (t & (1 << i)) ? '1' : '0';
11.     return s;
12. }
13.
14. // C++20 required
15.
16. int main() {
17.     using std::cout; using std::rotr; using std::rotl;
18.
19.     std::uint8_t n = 0b01001101;
20.     cout << "n           : " << showBits(n) << '\n';
21.     cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22.     cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23.     cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24.     cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25. }
```

prints

```
n           : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Logical operators

Arguments of logical operators '`&&`' (AND), '`||`' (OR) and '`!`' (NOT) can be of type **bool** or of any integer type. In the latter case the interpretation is as usual: 0 → **false**, non-zero → **true**. The result is of logical type: the alternative (OR - '`||`') is **true** if, and only if, at least one of the arguments is **true**; the conjuction (AND - '`&&`') is **true** if, and only if, both arguments are **true**.

Evaluation of both logical OR and logical AND operators is **short-circuited**. This means that the second argument is not evaluated at all if the value of the first operand is sufficient to determine the value of the whole expression:

- for the AND operator (logical conjuction), if the first operand evaluates to **false**, then the value of the conjuction must be **false** and the second operand will not be evaluated;
- for the OR operator (logical alternative), if the first operand evaluates to **true**, then the value of the alternative must be **true** and the second operand will not be evaluated.

For example, if ***a***, ***b*** and ***r*** are integer variables, then the assignment

```
r = a && b;
```

is equivalent to

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

and

```
r = a || b;
```

is equivalent to

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Let us consider another example:

P61: shortc.cpp Short-circuited logical operators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  bool fun(int k) {
5.      k = k - 3;
6.      cout << "Fun returns " << k << endl;
7.      return k;
8.  }
9.
10. int main() {
11.     if ( fun(1) && fun(2) && fun(3) && fun(4) )      ❶
12.         cout << "AND true " << endl;
13.     else
14.         cout << "AND false" << endl;
15.
16.     if ( fun(1) || fun(2) || fun(3) || fun(4) )      ❷
17.         cout << "OR  true " << endl;
18.     else
19.         cout << "OR  false" << endl;
20. }
```

On line ❶ we check if a conjuction of four logical values returned by the function **fun** is true. As the function call `fun(3)` returns 0 (i.e., **false**), the function will not be invoked with argument 4, as the result is already known to be **false**. We can see it from the printout:

```
Fun returns -2
Fun returns -1
Fun returns 0
AND false
Fun returns -2
OR  true
```

Similarly for the alternative from line ❷: the first function invocation returns non-zero (-2, interpreted as **true**). Therefore, no other invocation will ocurr: the result is already known to be **true**.

9.2.9 Assignment operators

The group of priority 2 consists of a variety of **assignment operators** . The simplest one is just the “normal” assignment ' ='.

The left-hand side of an assignment must be an l-value. Thus

```
double x;
x + 2 = 7; // WRONG
```

would be illegal, while

```
double x, *y = &x;
*(y + 2) = 7;
```

is legal, as the dereference yields an l-value.

When an assignment statement is executed, first the *value* of the right-hand side is evaluated (giving an r-value). Then this value is assigned to the variable (not necessarily named) which stands on the left-hand side of the assignment. Note the assymetry: the right-hand side tells *what* to evaluate, the left-hand side — *where* to store the result.

The assignment as a whole has itself its value and is an l-value. Namely, its value, type and address is equivalent to the value, type and address of the variable on the left-hand side *after* the assignment has been performed. For example,

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

prints '6 2'.

On line 7 of the following code

P62: assign.cpp The value of an assignment

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int k;
7.     while ( (k = cin.get()) != '\n' )
8.         cout << "Entered char '"     << (char)k
9.              << " with ASCII code " << k << endl;
10. }
```

we assign to ***k*** the value of a character read from the standard input, i.e., its ASCII kode — see the chapter on i/o operations . The assignment ' `(k=cin.get())` 'as a whole has the value of ***k*** after the operation has been completed; this value is then compared with the constant '\n' (which is ***NL*** — the newline character); if these two values are equal, the loop terminates. Note that the parentheses are necessary, as the precedence of comparison ' `!=`' is higher than that of the assignment, and we want the assignment to be performed first. The result of the program can be something like

```
cpp> g++ -pedantic-errors -Wall -o assign assign.cpp
cpp> ./assign
The quick...[ENTER]
Entered char 'T' with ASCII code 84
Entered char 'h' with ASCII code 104
Entered char 'e' with ASCII code 101
Entered char ' ' with ASCII code 32
Entered char 'q' with ASCII code 113
Entered char 'u' with ASCII code 117
Entered char 'i' with ASCII code 105
Entered char 'c' with ASCII code 99
```

```
Entered char 'k' with ASCII code 107
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
Entered char '.' with ASCII code 46
cpp>
```

As the result of an assignment is an l-value, we can use it in sequence:

```
int k = 7, j, m;
int n = m = j = k;
```

The assignment is right-associative, so the value of '`m=j=k`' is equivalent to the value of '`m=(j=k)`', i.e., to the value of ***m*** after assignment (7 in our case). This value will be used to initialize the newly created variable ***n***. As a side effect, the variables ***m*** and ***j*** will also have new values assigned (note that the statement would be illegal if any of the variables ***m***, ***j*** or ***k*** did not exist).

The **compound assignment** operators allow for simpler formulation of some assignments: those in which the same variable appears on the left- and rigthand side of the assignment. Instead of

```
a = a @ b;
```

where the symbol '@' stands for one of the symbols

+ - * / %

<< >> & ^ |

one can use

```
a @= b;
```

There is a subtle difference between these two forms, usually insignificant but sometimes important. In the second form (`a @= b`), the variable ***a*** is evaluated only once, while in the first form (`a = a b;`) — twice. Therefore, this second form is recommended as more effective.

Let us consider an example:

P63: comp.cpp Compound assignments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void bitsInt(int k) {
5.      unsigned int mask = 1<<31;
6.      for (int i = 0; i < 32; i++, mask >>= 1) {
7.          cout << (k & mask ? 1 : 0);
8.          if (i%8 == 7) cout << " ";
9.      }
10.     cout << endl;
11. }
12.
13. int main() {
14.     unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ❶
15.     cout << "k before: "; bitsInt(k);
16.     (k <<= 8) >>= 24;
17.     cout << "k  after: "; bitsInt(k);                  ❷
18. }
```

The function **bitsInt** is similar to that from the program `bits.cpp` — it prints the bit representation of an integer. In this version we assumed that **int** occupies four bytes. We also moved shifting of the mask to the incrementation part of the header of the **for** loop, placing there two expression statements separated by a comma (more on the comma operator in one of the next subsections). We added printing a space between eight-bit groups to improve readibility.

On line ❶ we construct a number with a particular bit representation. The expression '`255 << 24`' yields a value with all bits of the least significant byte set to 1 and then left shifted by 24 bits; in the result, the most significant byte is filled with 1's, all other bits being unset. The expression '`153 << 16`' represents the bit pattern `10011001` left shifted by 16 bits (so it occupies the third byte from the right). Similarly, '`255 << 8`' gives eight 1's in the second byte and 255 eight 1's in the least significant byte. By ORing all these values, we get a number with bit representation as shown in the first line of the output:

```
k before: 11111111 10011001 11111111 11111111
k  after: 00000000 00000000 00000000 10011001
```

Operator of compound assignment was used on line❷. The expressiom '`(k <<= 8)`' shifts all bits in ***k*** by 8 to the left (the leftmost byte flows out, the other three are moved, so `10011001` becomes the leftmost, and the rightmost byte is filled with zeros) — the resulting value replaces the old contents of the variable ***k***. The result (which is ***k*** after the shift has been completed) becomes now the left argument of the next shift — this time to the right by three bytes (24 bits). Now all three least significant bytes flow out to the right and are lost, the byte `10011001` becomes the least significant (the rightmost), and three most significant bytes are filled with zeros, as ***k*** is of an **unsigned** type. As the result, we get the number whose bit representation was contained in the third byte (counting from the right) of the original value. In a similar way we could cut out other bytes — such operations are often needed when dealing with colors whose three (or four) components are packed into one integer variable.

9.2.10 Conditional operator

Conditional operator

The **conditional operator** is the only **ternary** (acting on three arguments) **operator**. Its syntax is

```
b ? w1 : w2
```

The value of expression ***b*** is evaluated first and, if needed, converted to type **bool**. If the result is **true**, the value of ***w1*** is evaluated and becomes the value of the whole expression (the value of ***w2*** is not evaluated at all). If the result is **false**, the value of ***w2*** is evaluated and becomes the value of the whole expression, while the value of ***w1*** is not evaluated at all. If (and only if) both ***w1*** and ***w2*** are l-values, then the whole expression is also an l-value.

For example, a function returning the larger of its arguments can be implemented as:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Another example will be given in the next section.

9.2.11 Exception-throw operator

Operator of exception throwing (**throw**): we will explain it in the chapter on exceptions .

9.2.12 Comma operator

The **comma operator** is an infix binary operator: its arguments appear on its two sides

```
expr1 , expr2
```

The operator acts as follows:

- the expression on the left is evaluated and the result is ignored;
- the expression on the right is evaluated and the result becomes the value of the whole expression.

The comma operator is frequently used in the initialization and incrementation parts in headers of **for** loops; an example can be found on line 7 of `comp.cpp`.

In the following program we have another, slightly bizarre, example of the comma operator:

P64: comma.cpp Comma operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int r = 0;
6.      int k;
7.
8.      while (cin >> k, k) {                          ❶
9.         r += k > 0 ? (cout << "Positive\n" , +1)
10.                    : (cout << "Negative\n" , -1);
11.     }
12.     cout << "Number of positive - Number of negative : "
13.          << r << endl;
14. }
```

The operator is used on line ❶ in the condition of the **while** loop. The first expression, '`cin >> k`', reads a number from the keyboard; the value of this expression (which is ***cin***) is ignored. Then the value of ***k*** is evaluated giving the value of the number which has just been read in. This value becomes the value of the whole expression in parentheses. If it is zero, the loop terminates. Inside the loop ***r*** is incremented by the value of the conditional operator — see section on conditional operator — which checks the sign of ***k***. Whether it is **true** or **false**, the value will be evaluated as the result of another comma operator

```
(cout << "Positive\n" , +1)
```

which will be `+1` with a side effect consisting in printing the word "Positive". Similarly, for negative ***k***, the value of ***r*** will be decremented by 1 and as a side effect the word "Negative"will be printed. After the termination of the loop, the value of ***r*** will be equal to the difference between the number of positive and negative numbers which have been entered.

For example,

```
cpp> ./comma
9
Positive
5
Positive
-3
Negative
6
Positive
-1
Negative
0
Number of positive - Number of negative : 1
cpp>
```

9.2.13 Alternative operator names

Some operators have alternative, purely textual names:

textual	symbolic	textual	symbolic
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	| =	xor	^
xor_eq	^=		

Table 9.2: Alternative operator names

These names can be used interchangeably with those expressed by non-letter symbols.

Prev: 9.1 Precedence and associativity of **Up:** 9. Operators **Next:** 10. Standard conversions. Order of

10. Standard conversions. Order of evaluation

In this chapter we will discuss standard conversions which are performed (often silently, where one would not expect them...) Understanding this issue is necessary to understand the mechanism of invoking functions and their overloading, which will be covered in the next chapter. We will pursue the issue of conversions in more detail in chapter on conversions .

The problem of the order of evaluation of various parts of complex statements is perhaps much less fundamental, but one has to realize some facts connected with this issue to avoid some particularly nasty and hard to detect errors.

Subsections

- 10.1 Conversions
- 10.2 Order of evaluation

T.R. Werner, September 27, 2023; 22:32

10.1 Conversions

Let us consider the following snipet:

```
double x = 1.5, y;
int    k =  10;
    // ...
y = x + k;
```

What function will effectuate the addition '`x+k`'? The variable ***k*** is of type **int**, while ***x*** of type **double**. They have different bit representation — even their size is different. It is clear that in order to add them, this difference has to taken into account. What will be the type of the result? Is there a special function which adds **int** s and **double** s? What if it were **double** and **unsigned short** to be added — is there another function to carry out this operation? The same problem arises with invoking functions. After including the header `cmath` (or `math.h`), we have access to the **sin** function which has one parameter of type **double** (or **long**). Does it mean that `sin(1)` is an error, or there is a version of **sin** with parameter of type **int**? These are problems we are going to deal with now.

Suppose there is an expression with a binary operator (like addition, multiplication, ...). Before the operation as such takes place, values which are to be acted upon are **converted**in such a way that

- types of arguments are the same;
- there is a function which performs the operation for this common type;
- the precision of the result is *preferably* not worse than the precision of arguments.

The first two objectives come from the fact that functions which perform such operations are implemented only for arguments of the same type, but not for all possible types. It is also important what is the type of the result:

> **The type of the result of an arithmetic binary operation is the same as the common type of the arguments *after* standard conversions.**

How does the process of searching for the common type of arguments look like? Generally, the conversions are, if possible, **promotions**, i.e., “smaller” type is promoted to a “larger” type, so the precision is not lost. In this sense, **int** is “smaller” than **double**, as every value of type **int** can be represented, without loss of precision, as **double**, but not *vice versa*. On the other hand, not for all pairs of types such relation of inclusion exists: the set of values for the types **int** and **unsigned int** both have the same cardinality (equal to 2^{32} elements), but none of them is a subset of the other.

The rules are as follows:

1. If one of the arguments is a **long double**, then the second is converted to **long double**;

2. Otherwise: If one of them is **double**, then the second is converted to **double**;
3. Otherwise: If one of them is **float**, then the second is converted to **float**;
4. Otherwise both arguments are of integer type and are subject to integer promotion, according to the following procedure:
   - Values of types **signed char**, **unsigned char**, **signed short** and **unsigned short** are converted to type **int**, if **int** can represent all their values (which is normally true for common 32-bit architectures). If it is not the case, they are converted to type **unsigned int**, what *will* change the value of negative variables!
   - Values of enumerated types are converted to the smallest of types **int**, **unsigned int**, **long**, **unsigned long**, which is sufficient for representing all values of a given enumeration type.
   - Bit fields (see section on bit fields) are converted to values of type **int** if this is sufficient, or to **unsigned int** otherwise.
   - Values of type **bool** are converted to values 0 (**false**) and 1(**true**) of type **int**.
5. Then, if one of the argument is of type **unsigned long**, then the second is converted to **unsigned long**;
6. Otherwise: if one of the argument is of type **long** and the second of type **unsigned int** and all values of type **unsigned int** can be represented as **long** (what is normally *not* the case), then the value of type **unsigned int** is converted to **long**; otherwise both are converted to **unsigned long** (what can lead to a disaster...);
7. Otherwise: if one of the arguments is of type **long**, then the second is converted to **long**;
8. Otherwise: if one of the arguments is of type **unsigned int**, then the second is converted to **unsigned int** (what is also dangerous);
9. Otherwise: both arguments are of type **int** and no conversion is needed.

For example, the expression from the third line below

```
int i = 1, j = 2, k = 3, m;
// ...
m = (j > i) + (k > j);
```

is correct: the values of type **bool** of the expressions `(j > i)` and `(k > j)` will be converted to **int** s 0 or 1 and then the addition will be performed. In the example both values are 1 and the value of the right hand side is 2.

The very common error is connected with integer division. One has to remember that when two integer values are divided, the result is always integer — the fractional part is not “cut off”; it is not calculated at all! The fact that we assign the result to a floating point variable will not help: the fractional part simply does not exist in the result! If, for eample, the current value of a variable ***k*** of type **int** is 7, then the value of '`k/2`' is *exactly* 3. If $3\frac{1}{2}$ was what we meant, then it would be enough to add a decimal point at the literal 2: the value of '`k/2.`' is 3.5, as '2.' (with the dot) is interpreted as a literal of type **double**, so ***k*** will be converted also to this type and the result will be **double** too.

There are other standard conversions which can be performed (even without our knowledge and acceptance...):

- Any pointer to an object can be converted to type **void***. This does not apply to function pointers — section on function pointers ;
- Integer value 0 can be converted to the pointer type giving the “empty” pointer (**nullptr**). This applies to the value zero *only* — other integer values cannot be converted to pointers. The conversion from empty pointer to integer zero is illegal;
- If **T** is a type, then values of type **T*** can be converted to **const T*** and values of type

T& to **const T&**; there is no standard conversion in the opposite direction (although we can enforce it);
- Values of integer, floating point or pointer types can be converted to type **bool** (zero → **false**, non-zero → **true**);
- Integer values can be converted to integer values of another type; if the target type is **unsigned**, then the resulting value will be constructed by simply copying as many bits as the target type has, without any interpretation;
- Values of type **float** can be converted to type **double** and *vice versa!*
- Values of integer types can be converted to floating point types and *vice versa!* — in the latter case the fractional part will be ignored;
- Pointers or references to object of a publicly derived classes can be converted to pointers or references to objects of their base class.

It sounds complicated because *it is* complicated. Moreover, conversions can be dangerous, as in C++ even conversions leading to loss of information can be formally legal.

Let us look at an example:

P65: surp.cpp Standard conversions

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.      int      k   = -2;
7.      unsigned uns =  1;
8.
9.      int      x = k + uns;
10.     unsigned y = k + uns;
11.
12.     cout << "x   = " << x      << endl;
13.     cout << "y   = " << y      << endl;
14.     cout << "y+1 = " << y + 1  << endl;
15.
16.
17.     signed   char c = 255;
18.     unsigned char d = 255;
19.
20.     cout << "c+1 = " << c + 1  << endl;
21.     cout << "d+1 = " << d + 1  << endl;
22.     d = d + 1;
23.     cout << "d   = " << (int)d << endl;
24. }
```

The program prints:

```
x   = -1
y   = 4294967295
y+1 = 0
c+1 = 0
d+1 = 256
d   = 0
```

The value of ***x*** is `-1`, as expected. But, what is perhaps less expected, the value of ***y*** is `4294967295` (this is not an accidental number; its value is exactly equal to 2^{32} - 1). Moreover, incementing this value by one gives exactlu zero (line 14 of the program prints '`y + 1 = 0`'). It seems that variables ***c*** and ***d*** have the same value, and after adding unity they still should have the same value. But line 20 prints '`c + 1 = 0`', while line 21 prints '`d`

`+ 1 = 256`'. However, when we assign the value of '`d + 1`' back to ***d*** and print its integer value we get again 0 (line 23). As one can see, conversions between **signed** and **unsigned** types can be particularly dangerous — this is related with the fact that sets of values for these types are different and none of them is a subset of the other: one cannot tell which is “smaller” and which is “larger”. It is therefore better to avoid such subtle considerations and write our code in a simple and straightforward way using, if possible, only basic types.

It is worthwhile to remember that all “small” integer types are converted to **int**. It applies, in particular, to **char** s. When used as arguments of operators or passed to functions as arguments they are converted to **int** s with the value equal to their ASCII codes. For example, after the assignment

```
int k = 3 + 'a';
```

the value of ***k*** is 100, as the ASCII code of the character 'a' is 97.

This feature is used in the function below, which reads characters until non-digit is encountered and interprets the string as a positive integer number:

P66: conv.cpp Conversions character → number

```
1. #include <iostream>
2. using namespace std;
3.
4. int convert(char* str) {
5.     int w = 0, i = 0, c;
6.     while (c = str[i++], c >= '0' && c <= '9')
7.         w = 10*w + c - '0';
8.     return w;
9. }
10.
11. int main() {
12.     char tab1[] = "123a";
13.     char tab2[] = "456 1";
14.     char tab3[] = " 56";
15.
16.     cout << "tab1 -> " << convert(tab1) << endl;
17.     cout << "tab2 -> " << convert(tab2) << endl;
18.     cout << "tab3 -> " << convert(tab3) << endl;
19. }
```

The printout is

```
tab1 -> 123
tab2 -> 456
tab3 -> 0
```

Funkcja **convert** takes as the argument an array of characters (as a pointer to its first element). Then, in a **while** loop, it processes consequtive characters. In the parentheses we have here a comma expression: the left expression reads in one character, assigns it to the variable ***c*** and increments the current value of the index. The whole expression has the value of the right argument of the comma operator, i.e., the value of boolean expression which checks if the character corresponds to a digit. What is the value of `c >= '0'`? Before comparison, both arguments are converted to **int** s. The value of ***c*** will be equal to the ASCII code of the character; similarly, ***'0'*** (with apostrophes!) will be the ASCII code of the character '0' — we know it is equal to 48, but this knowledge is not necessary here. This should be emphasized here: the value corresponding to *character* '0' is *not* zero — it is

the ASCII code of this character. The literal of the character with ASCII code zero would be '\0', with apostrophes and a backslash.

The program assumes that the codes of consequtive digits, 0,...,9 are consequtive integers, as they are in ASCII (this is, strictly speaking, is *not* guaranteed by the standard!). Therefore, the condition `c >= '0' && c <= '9'` checks if the code of a character is simultaneously greater or equal than the code of '0' and less or equal than the code of '9', i.e., if the character is a digit. If not, the loop will terminate.

The expression appearing in the next line, `c-'0'` has the *numerical* value equal to the number represented by character ***c***. If, e.g., ***c*** is the character '4', then `'4'-'0'` is the difference of ASCII codes of '4' and '0', which is `52 - 48 = 4` (numerically). Therefore, if a character is a digit, the value of ***w*** is multiplied by 10 (what corresponds to shifting its digits one position to the left) and the number which has just been read in is added. When the loop terminates, the variable ***w*** corresponds to the numerical value represented by digits from the input C-string. That is why ***tab2*** will be interpreted as 456; processing will termianate when the space separating the digits 6 and 1 is encountered. For ***tab3*** we will get zero, as the first character seen by the function is the space (non-digit).

Prev: 10. Standard conversions. Order of **Up:** 10. Standard conversions. Order of **Next:** 10.2 Order of evaluation

10.2 Order of evaluation

Expressions are often built of subexpressions whose values must be evaluated before the value of the whole expression can be evaluated. For example, it can be a function invocation:

```
int k = fun1(x) + fun2(y);
```

In order to perform the addition, both its arguments have to be evaluated, i.e., functions **fun1** and **fun2** have to be called. But in what order? In many other languages this order is specified by their standard; this is not the case for C/C++. In C/C++ the compiler is free to choose an order which is more effective or simpler to implement. Consequently, one should never write a code whose result could depend on the order of evaluation of subexpressions. Otherwise, we can get strange and quite unexpected results! Let us consider the following simple program:

P67: ordeval.cpp Order of evaluation

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int zzz;      // global, initialized with 0
5.
6.  int fun1() {
7.      return zzz += 1;
8.  }
9.
10. int fun100() {
11.     return zzz += 100;
12. }
13.
14. int main() {
15.     cout << fun1() << " " << fun100() << endl;  ❶
16. }
```

One could expect that on line ❶ **fun1** will be invoked first: this will increment ***zzz*** by one and return 1. Then the call to **fun100** will add 100 to ***zzz*** and return the resulting 101. Therefore, one could expect that the program prints 1 and then 101. And this is indeed the case for some compilers. However, if we use e.g., **gcc**, we will get

```
101 100
```

This means that the function **fun100** was invoked first, its result (100) stored (pushed on some kind of an internal stack), and then **fun1** was called and its result (101) printed. Afterwards, the stored result of **fun100** invocation was printed.

For the similar reasons one should avoid decrementing or incrementing variables in the same statement. For example,

```
arr[i] = ++i;
```

is unclear: is the index on the left hand side already incremented or not? (It should be, but such constructs can lead to confusion).

Prev: 10.1 Conversions **Up:** 10. Standard conversions. Order of **Next:** 11. Functions

T.R. Werner, September 27, 2023; 22:32

11. Functions

We have been using functions from the beginning of our course — in this chapter we will describe them in more detail. We will deal with **global functions** (not members of classes). Such functions are sometimes termed **free functions**

Functions, like objects, may be pointed to by pointers, the so called **function pointers**. They will be described in this chapter — on a few examples we will show how to define and use them.

Subsections

- 11.1 Introduction
- 11.2 Declarations and definitions of functions
- 11.3 Function call
- 11.4 Default arguments
- 11.5 Variable-length argument lists
- 11.6 Reference arguments
- 11.7 Functions returning references
- 11.8 Recursive functions
- 11.9 Static functions
- 11.10 Inlined functions
- 11.11 Function overloading
- 11.12 Function pointers
- 11.13 Lambda functions
- 11.14 Function templates

11.1 Introduction

A function is essentially a prescription which tells how to obtain an effect given some data as an input. From the point of view of a program, a function is a set of statements (strictly speaking one compound statement) which uses data unspecified in the definition of the function, but which will be provided when the function is used (called, invoked). Data which must be provided when invoking a function is called the function's **arguments**. In the definition of a function, this data (values) is unspecified and enters the definition as its **parameters**. When a function is invoked, current values of arguments are assigned to parameters, so the code of the function can be executed. Functions can (but do not have to) return a *single* value as their result — we can assume that an invocation of a function in a program will be replaced by the returned value during execution.

Functions allow us to code a set of identical statements (differing only in values of input data) once; when defined it can be used any number of times with the same or different sets of input data.

Contrary to Java and some other languages, functions in C++ do not have to be members of any classes (in C they cannot, as in C there are no classes at all). Such a function, not defined as a member of any class, is called a **global function** or a **free function**.

There is one feature related to functions in C/C++ that can surprise Java programmers. The order in which definitions (or at least declaration) of functions appear in the *text* of a program *does* matter. When the compiler encounters a usage of a function, its definition (or at least declaration) must have already been processed. As we know from introductory chapter , the `#include` directive includes the contents of a file into the file being processed. It is therefore possible (and quite usual) to put declarations (or sometimes also definitions) into a header file and to include this header in all source files that use functions declared there.

11.2 Declarations and definitions of functions

As we have said, when the compiler encounters a usage of a function, this function must have already been declared/defined. This is because the compiler must check

- which function of this name is meant here (there can be many functions with the same name);
- whether the invocation is correct, i.e., whether the number and type of arguments corresponds to the number and type of parameters, if the type of the returned value is acceptable, etc. This is needed, for example, in order to decide whether conversions of arguments or of the return value is necessary.

If something is wrong, compilation is terminated, what is better than producing absurd executable (as it was possible in traditional C, where declarations before usage were recommended but not required).

Why definition *or* declaration and what *declaration* is, in the first place. Why not just definition?

First, let us imagine the following situation: we define two functions, **fun1** and **fun2**. The function **fun1** calls the function **fun2** and *vice versa* — function **fun2** calls **fun1**:

```
1.      void fun1(int k) {
2.          // ...
3.          fun2(k)
4.          // ...
5.      }
6.
7.      void fun2(int m) {
8.          // ...
9.          fun1(m)
10.         // ...
11.     }
```

In what order these two functions should be defined? If we define them as shown above, the compilation will be interrupted on line 3, as no information is available about **fun2**. Reversing the order will not help, as then **fun1** will be unknown inside the definition of **fun2**.

As the matter of fact, in the situation above, the compiler does not need any definition on line 3. All what it is required is to check if the usage of **fun2** is correct, i.e., if the number of arguments and their types are correct etc. In order to be able to perform this task, all what the compiler needs is the **prototype** of **fun2**. This is just what declarations are designed to provide. The prototype has the form of the header of a function being declared but without the body (definition itself) — the header is terminated by a semicolon right after the list of parameters: For example

```
string&   fun1(char* c1, char* c2, bool b);
void      fun2(int k, double d[]);
AClass*   fun3(AClass* k1, AClass* k2);
```

Declarations have no body, so they are not definitions. However, they provide all

information about the name of a function, number and type of parameters and the function's return type. Hence, in our example, it will suffice to add a declaration of **fun2** *before* the definition of **fun1** and it will be compiled smoothly:

```
1.      void fun2(int);
2.
3.      void fun1(int k) {
4.          // ...
5.          fun2(k)
6.          // ...
7.      }
8.
9.      void fun2(int m) {
10.         // ...
11.         fun1(m)
12.         // ...
13.     }
```

Note that in the declaration in the first line we did not mention any name for the parameter of the function **fun2**, although we usually have to specify it in the definition. In the declaration what matters is the number and *types* of parameters, not their names. Names may be specified (and in fact should be: well chosen names can often play rôle of a useful comment), but they do not have to be specified; they are completely ignored by the compiler in declarations, so even when specified they may be different than those given afterwards in the definition.

Three declaration given above can be rewritten without names of parameters:

```
string&   fun1(char*, char*, bool);
void      fun2(int, double[]);
AClass*   fun3(AClass*, AClass*);
```

Every declared function must be eventually defined (once only). Otherwise the program will be compiled but not linked. Note, however, that there will be no error if a declared function is never actually used — we can declare functions that we only plan to implement as long as we do not try to use them before defining them. Definition does not have to appear in the same file; it will suffice if it is defined in any file (compilation unit) that the linker will find when producing the executable. In all other compilation units in which a function is used, the function's declaration is sufficient.

As our short programs are contained in one file, we will deferr the details to section on modules .

Functions can be declared many times, even in the same file. However, they should be defined once only (ODR – "one definition rule"); this does not apply to inlined functions, to be discussed shortly.

Of course, all declarations of the same function must agree, i.e., they must define the same prototype which will be eventually used in the function's definition.

The **header** of a function specifies its prtototype: in its simplest form it looks like this:

```
Type Name ListOfParams
```

where ***Type*** specifies the return type: type of a single value returned by the function (in front of ***Type*** some modifiers acn appear; we will discuss them later). The ***Name*** stands for the name of the function, and ***ListOfParams*** for the list of the function's formal parameters.

Return type.

It must be the name of a valid type: built-in (like **int**, **char**, ...), user-defined (like **Person**, **Employee**, ...) or derived (**int***, **Person&** etc.). If the function does not return any value, its return type must be specified as **void**. The function with return type **Type** is itself termed as being of type **Type**.
Array or function type cannot be the return type of any function. This is not so restrictive, as *pointers* to arrays or functions *can* be returned by functions. In C++ functions may return references.

Name.

An **identifier**, or simply *name*, is a sequence of letters, digits, and underscores (space characters in names are *not* allowed!). The first character cannot be a digit. An identifiers must not have the same spelling as a reserved word (like **while** or **private**). Identifiers are case sensitive.

List of parameters.

This is, enclosed in round parentheses, a list of comma separated declarations of single parameters of the function. Each declaration has the form '`type name`', where **type** is the name of a type, and ***name*** is the name of the parameter (names are not necessary in declarations). "Compound" declarations, '`type name1, name2`', are not allowed — it would be unclear if ***name2*** is the second parameter of the type **type** or the name of the type of the next, anonymous (what is legal), parameter. Names of all parameters must be different.
We know that names may be omitted in declarations; they may remain unspecified also in a definition if the function does not use the corresponding parameter. This often is the case in the process of creating a function, when definition is not yet final, but we want to have its header already in its full form.
A list of parameters may be empty; the parentheses, however, cannot be omitted (to emphasize the fact that the list is empty, one can put in parentheses the keyword **void**).
One should keep in mind that adding **const** to type name yields different type: **int*** and **const int*** are different types. If a parameter is, e.g., **const int***, the compiler will not allow to change the variable pointed to by the pointer which was passed to the function (what is passed is a *copy* of an address, but it can point to a variable from the calling function).

The part of the function's header consisting of its name and the list of parameters (only types, parameter names are irrelevant) is called its **signature**. Note that the return type is often considered not to belong to the signature. For example, if the prototype of a function is

```
double fun(double x, char* nap);
```

then its signature is

```
fun(double, char*)
```

The **definition** of a function cosists of a header and a body containing the definition proper. The body follows the header and is enclosed in braces (what formally makes it a compound statement). If a parameter is to be used in the body (definition) of a function, this parameter must be given a name. Except for inline functions, any function of a program must have exactly one definition (this is called *one definition rule*). Inline functions will be covered in section on inlined functions).

The body of a definition can be viewed as one compound statement. Declarations of parameters belong to its scope. All variables (generally: *names*) defined in the function's body are local; this applies to parameters as well. They can be referred to only inside the function. Variables specified as parameters will be created after invocation of the function and initialized by the values of arguments used in this particular call. After the execution of the function is completed, all local variables (with the exception of variables declared as **static**) are removed.

This means, in particular, that names defined in different funcions are completely independent and denote completely distinct entities. Therefore, different funcions may define entities with the same name without any conflicts. The only problem which can arise is with names declared in the global scope. Globally declared names are hidden by names declared locally; the global names are still accessible, but the operator of global scope resolution (double colon) must be used (see section on scopes and visibility of variables).

Definitions of functions cannot be nested (as they can in other languages, e.g., Pascal or Fortran 90/95). However, under the new C++11 standard, inside a function, one *can* define the so called lambda functions (more about lambdas later).

Definitions (and declarations) of functions can have slightly different syntax in the new C++11 standard. Namely:

```
auto f_name( parameters ) -> return_type { body }
```

Instead of specifying the return type *before* the name of the function, we put there the keyword **auto**, and the return type is specified *after* the list of parameters and the “arrow” sign (' ->'). This is called **trailing return type**. We do not even have to specify this type explicitly; we can use here a **decltype** expression (see. chapter on data types). It is then possible to use names of parameters in **decltype** expression, as in this place we already are in the scope of the function. Let us illustrate it with an example:

P68: fundefnew.cpp Function definition with trailing return type

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  auto D(int a, double b) -> decltype(a*b);   ❶
5.
6.  double A(int a, double b) {                 ❷
7.      return a*b;
8.  }
9.
10. auto B(int a, double b) -> double {         ❸
11.     return a*b;
12. }
13.
14. auto C(int a, double b) -> decltype(a*b) {  ❹
15.     return a*b;
16. }
17.
18. double D(int a, double b) {                 ❺
19.     return a*b;
20. }
21.
22. int main() {
23.     cout << A(4,2.5) << " " << B(4,2.5) << " "
24.          << C(4,2.5) << " " << D(4,2.5) << endl;
25. }
```

We define here four functions (**A**, **B**, **C**, **D**), which are identical — they all just return the product of their arguments. The definition of function **A** (❷) is „normal”. On line ❸ the new syntax has been used (with **auto** before the function's name and return type after the parameter list and the arrow.) In the definition of **C** (❹), instead of specifying the type explicitly, we “asked” the compiler to determine by itself the type of the product of arguments (it will be **double**, of course). As we can see, the new syntax can also be used to declare a function (❶) — the corresponding definition (❺) can be expressed using the the new or the traditional syntax (then, of course, the return type has to agree with that deduced by the compiler from the declaration.)

The form used here to define the function **C** (❹) will turn out to be very useful when defining function templates.

Let us also emphasize that declarations and definitions of functions are *not* executable statements (as they are, e.g., in Python). Their order is therefore irrelevant, as long as we adhere to the rule that at least one declaration must lexically precede all statements in which the function is used.

Prev: 11.1 Introduction **Up:** 11. Functions **Next:** 11.3 Function call

11.3 Function call

Functions which have been at least declared (or fully defined) can be used in the program, i.e., **called** (**invoked**). To invoke (to call) a function, its name has to be specified followed by a list of comma separated arguments enclosed in round parentheses. Of course, the number and types of arguments must be consistent with the declaration. Even for parameterless functions, the parentheses (empty) are still required. Without a list of parameters, function's name will be interpreted as function pointer, not its invocation (see the section on this subject).

Arguments do not have to be specified as names of any variables. Any expression with the value of an appropriate type can play rôle of an argument. The type does not have to be exactly the same as the type of the corresponding parameter: if there is a standard conversion which can be used to convert the value of the argument to the value of the type of the parameter, it will be implicitly performed (see section on standard conversions).

As we know, standard conversion can lead to loss of information (e.g., when **double** is converted to **int**). If such a conversion is needed, the compiler will usually emit some kind of warnings, but compilation can succeed.

If a function is not of type **void**, its execution has always to end with the **return** statement with an expression having the value of the declared return type of the function (or a type that can be converted to this type by means of a standard conversion).

In functions which are of type **void**, **return** statements can also be used. No expression to be returned is then allowed. When the **return** statement is encountered, the flow of control exits the function and returns to the caller. For such functions, the **return** statement is tacitly implied just before the brace closing the body of the function.

We can imagine that the process of function invocation proceeds as follows (the details depend on platform and compiler used):

- The values of expressions used as arguments are evaluated. They are converted to the type of the corresponding parameters, if necessary. This conversion must be a valid standard conversion, otherwise the program is in error.
- The values obtained are then pushed onto the program's stack. It is important to remember that only the *value* is pushed (copied) onto the stack: if we use a name of a variable as an argument, then a copy of its value will be copied, so the function will never “know” what variable was used as the argument and, in particular, what was the name of this variable. As a consequence, the variable used as an argument is usually not accessible by the function and hence cannot be modified by it. The value which is pushed on stack must be exactly of the type declared as the type of the corresponding parameter — if necessary, an appropriate standard conversion will be performed automatically.
- The flow of control enters the function: values of arguments are used to initiate *local* variables corresponding to function's parameters. All other non-static variables defined inside the body of the functions are also created on the stack.

On the other hand, a **static** variable defined in the function is created and initiated once only — when the flow of control enters the function for the first time. Afterwards, inside the function, *the same variable* will be visible, with its value exactly as it was when the flow of control left the function after the previous entry. Such a **static** variable will exist until the end of the program — it differs from global variables, however, because it is visible only in the scope of the function, from the place it was declared to the end of the narrowest surrounding block (i.e., to the end of the body of the function, unless it was declared inside a block nested inside the function).

- The flow of control leaves the function and returns to the caller when the end of the body of this function or a **return** statement has been encountered. The part of the stack containing all local variables, including the variables corresponding to the parameters, is at this moment rewound (in other words, disappears). The result of the function, if it exists, is returned to the caller — we can imagine that this is done through the stack as well, although other mechanisms are normally used in practice. The value returned must be exactly of the type declared as the type of the function: if necessary, appropriate conversions will be performed. As with arguments, copies of values are returned, not names.

The following program illustrates conversions of argument and the return value:

P69: function.cpp Conversions of arguments and return values

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int d2i(double);
5.  double i2d(int);
6.
7.  int main() {
8.      double x = 3.5;
9.
10.     cout << "d2i(x) = " << d2i(x) << endl
11.          << "i2d(x) = " << i2d(x) << endl;
12. }
13.
14. int d2i(double x) { return 3*x; }
15. double i2d(int k) { return 3*k; }
```

Both functions, **d2i** and **i2d**, return the value of their argument multiplied by 3. As both are called with the argument 3.5, in both cases the result should be 10.5. Compilation gives some warnings but succeeds and the program runs smoothly:

```
cpp> g++ -o function function.cpp
function.cpp: In function `int main()':
function.cpp:11: warning: passing `double' for converting 1
                 of `double i2d(int)'
function.cpp: In function `int d2i(double)':
function.cpp:14: warning: converting to `int' from `double'
cpp> ./function
d2i(x) = 10
i2d(x) = 9
cpp>
```

Neither **d2i** nor **i2d** gave the correct result! Why?

The parameter of **d2i** is of type **double**. Therefore, the value of ***x***, which is also of type **double**, has been passed as **double**. Inside the function this value has been multiplied by 3; the result is 10.5. However, the function **d2i** is of type **int** (i.e., it has to return an **int**).

The result is therefore converted to this type, giving exactly 10.

The function **i2d** has one parameter of type **int**. Therefore, the value of the argument ***x***, which is 3.5, will be converted to this type before passing it to the function: the function will get the integer value 3. This will be multiplied by 3 giving 9. The return type of the function is **double**, so the value 9 will be converted to **double** — the type of the result will be **double**, but its value will remain 9 (i.e., 9.00000).

Note that in this program, we first declared both functions, without defining them. This was enough for the compiler when it encountered their invocations in **main**. Eventually, of course, definitions must have been given.

Prev: 11.2 Declarations and definitions of **Up:** 11. Functions **Next:** 11.4 Default arguments

11.4 Default arguments

In C/C++ it is possible to define default values for all or some arguments of a function. If such default values are specified for a function, one does not have to specify the corresponding argument when invoking this function: if argument is not specified, the default value will be used.

Default value for argument corresponding to a given parameter of a function must be specified once only. If specified, it will hold for all subsequent declarations and the definition *without* repeating them there. Default values are specified in the header of a function (declaration or definition) by adding an equal sign ('=') followed by default value at the end of a parameter declaration. It is only allowed to specify default values for the rightmost parameters on the list: if a parameter has a default value, all parameters to the right must have them too. Let us consider an example. The function below will have one obligatory argument (of type **int**) and two parameters with default values provided:

```
void fun(int i, int b = 0, double x = 3.14);
```

As this is a declaration, not a definition, names of parameters could have been omitted:

```
void fun(int, int = 0, double = 3.14);
```

Of course, somewhere in the program a definition will have to be provided: as default values have already been specified in the declaration, they must not be repeated in the definition:

```
void fun(int i, int b, double x) {
    // ...
}
```

After declaring the function, we can call it in different ways; e.g.,

```
fun(3, 4, 7.5);
fun(3, 4);
fun(3);
```

In the first line all three arguments have been specified, so no default value will be used. In the second line we specified two arguments: the value of the first will be used for initialisation of the first parameter (***i***), the value of the second — for the second parameter (***b***; the default value 0 will be ignored here). The third argument has not been specified, so the third parameter (***x***) will be initialised with its default value (which is 3.14). In the third line, only one argument was provided (for the first parameter) — for the two other default values will be used.

Therefore, the three invocations above are equivalent to:

```
fun(3, 4, 7.5);
fun(3, 4, 3.14);
fun(3, 0, 3.14);
```

Note that it would be impossible to specify the third argument without specifying the second:

consecutive arguments are always associated with consecutive parameters, and only if the list of parameters has not been exhausted, the remaining will assume their default values. It is also impossible to invoke a function without specifying arguments for parameters that have not been assigned default values. In our case, for example, the function **fun** cannot be called without arguments, as no default value has been assigned to the first argument.

A default value does not have to be specified as a literal value of an appropriate type. It can be an expression containing global variables or some function calls. If this is the case, the default value will be evaluated each time the function with default arguments is called. The evaluation will be performed in the scope to which the declaration where the default values were assigned belongs. For example, in the program below, in the scope to which the declaration of function **disc** (❶) belongs, the *global* variable ***PI*** (defined just above) is visible.

P70: defval.cpp Default arguments

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  double PI = 3;
5.
6.  double disc(double, double = PI);            ❶
7.
8.  int main() {
9.      double r = 1;
10.
11.     cout << "1. PI = " << PI << " Area = "
12.                       << disc(r) << endl;   ❷
13.     double PI = 3.14;
14.     cout << "2. PI = " << PI << " Area = "
15.                       << disc(r) << endl;   ❸
16.     ::PI = 3.14;                            ❹
17.     cout << "3. PI = " << PI << " Area = "
18.                       << disc(r) << endl;
19. }
20.
21. double disc(double r, double pi) {
22.     return pi*r*r;
23. }
```

Therefore, the current value of this variable (3) will be used to evaluate the default value for the second argument of the function invocation on line ❷. Next, we define a *local* variable ***PI*** with a different value (3.14) and invoke the function again (❸). As we can see from the second line of the output,

```
1. PI = 3 Area = 3
2. PI = 3.14 Area = 3
3. PI = 3.14 Area = 3.14
```

the default value used was still 3, as it was evaluated in the global scope, where the *global* ***PI*** is visible, with the value still equal to 3. However, on line ❹, we change the value of the global ***PI***. Now the new value will be used to evaluate the default value of the second argument, as is clearly seen from the last line of the output.

Parameters which were declared as obligatory can change their status and be redeclared with default values assigned to them. One should only remeber that every parameter can be assigned a default value *once only*. When parameters (one or more) are assigned default values, they have to be the rightmost parameters of all those which have not yet been

assigned any default value.

Suppose we have declared, in a header file **defval1h.h** , a function with four parameters, the last of which has a default value (equal to 255):

```
void color(int r, int g, int b, int alpha = 255);
```

Let us now consider a program:

P71: defval1.cpp Redeclaring parameters with default values

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  #include "defval1h.h"
5.
6.  void color(int, int , int = 0, int);        ❶
7.
8.  int main() {
9.      color(100,150,250,199);
10.     color(100,150,250);
11.     color(100,150);
12. }
13.
14. void color(int red, int green, int blue, int alpha) {
15.     cout << "Alpha = " << alpha << " (R,G,B) = (" << red
16.          << "," << green << "," << blue << ")" << endl;
17. }
```

First, we include the header **defval1h.h**. In the global scope we now have the declaration of function **color** with one (the fourth) default argument. The last *obligatory* argument is the argument number 3. We can then change its status by redeclaring (❶) the function **color**. We add the default value for the third argument. Note that we *must not* repeat the default value for the fourth argument, as it has already been specified (in the included header). It would be even invalid to assign the same value of 255 to the fourth argument. After the redeclaration, we can invoke the function with four, three or two arguments, and get the result:

```
Alpha = 199 (R,G,B) = (100,150,250)
Alpha = 255 (R,G,B) = (100,150,250)
Alpha = 255 (R,G,B) = (100,150,0)
```

Of course, it is better to avoid tricks with redeclaring functions and changing the status of their arguments from obligatory to having default values, as this can easily lead to a total confusion.

Default arguments are often used in constructors of classes.

11.5 Variable-length argument lists

There are functions which do not have a natural, well defined number of parameters. For example, we could imagine a function which returns the maximum value of all of its two, three, four... arguments. Or a function which prints its one, two, three... arguments. The mechanism providing tools which allow to achieve this in C/C++ is called **variable-length argument list**. In C++ it is usually better to use the mechanism of overloading, or tools provided by the new standard, like tuples or initialization lists.

Functions of this type are declared with an ellipsis (' . . .') in place of parameters whose number and type remains unspecified. Before the ellipsis (but not after), other, "normal" parameters are specified in the usual way (in a declaration, as we know, names can be omitted). For example

```
int fun(char* ...);
```

declares a function which will be called with one obligatory argument (of type **char***) and any number of other arguments. The same declaration can be written with a comma after the last obligatory parameter:

```
int fun(char*, ...);
```

First of all, in order to use the mechanism of variable-length argument list, one has to include the header `cstdarg`. Then, in the function, one proceeds according to the following scheme:

1. Define a variable of type **va_list**, which is defined in the included header file (the name of this type comes from *variable-arguments list*). Traditionally, this variable is named ***ap*** (from *argument pointer*).
2. Function **va_start** should now be called *once* at the beginning of the body of the function (this is usually not a function but rather a preprocessor macro; this is, however, irrelevant for us as users). As the first argument we pass the just created variable (***ap***) of type **va_list**, and as the second one the name of the *last* obligatory parameter; example below this call will be `va_start(ap,type)`.
3. The values of arguments can now be read off, one by one, by function (macro) **va_arg**: its first argument is ***ap***, and the second one is the *name* of the *type* of the argument to be read from the stack. Hence we have to know this type in advance! Usually, information on the number and types of arguments is somehow coded in the first, obligatory argument. Each invocation of the function **va_arg** returns the value of the next argument and advances the internal argument pointer (***ap***) to the next argument.
4. When all arguments have been read, function (macro) **va_end** must be called (once). It performs necessary cleanup operations; neglecting to call it will leave the program stack in an undefined state, most probably causing a crash. The function is called with argument pointer (***ap***) as its only argument.

Let us consider an example:

P72: vararg.cpp Variable-length argument list

```
1.  #include <iostream>
2.  #include <cstdarg>
3.  using namespace std;
4.
5.  void types(const char typ[] ...);
6.
7.  int main() {
8.      types("SxS", "John", 0, "Mary");
9.      types("issD", 17, "John", "Mary", 1.);
10.     types("iDdsiI", 17, 19.5, 1.5, "OK", -1, 8);
11. }
12.
13. void types(const char typ[] ...) {
14.     int     i = 0, integ;
15.     char    c, *strin;
16.     double  doubl;
17.
18.     va_list ap;
19.
20.     va_start(ap,typ);
21.
22.     while ( (c = typ[i++]) != '\0') {            ❶
23.         switch (c) {
24.             case 'i':
25.             case 'I':
26.                 integ = va_arg(ap,int);
27.                 cout << "An int  : " << integ << endl;
28.                 break;
29.             case 'd':
30.             case 'D':
31.                 doubl = va_arg(ap,double);
32.                 cout << "A double: " << doubl << endl;
33.                 break;
34.             case 's':
35.             case 'S':
36.                 strin = va_arg(ap,char*);
37.                 cout << "A string: " << strin << endl;
38.                 break;
39.             default:
40.                 cout << "Invalid typecode!!!" << endl;
41.                 goto END;
42.         }
43.     }
44.     END:
45.     cout << endl;
46.
47.     va_end(ap);                                  ❷
48. }
```

The first, obligatory argument of the function **types** is a C-string, i.e., a pointer to a character array terminated with ASCII 0 character (' \0'). Characters of this C-string specify types of arguments which will follow:
'd' or 'D' — type **double**,
'i' or 'I' — type **int**,
's' or 'S' — type **char***, i.e., a C-string.
In the function's body, after the first two steps from the scheme we have just described, we have a **while** loop which reads the arguments passed to the function. First (❶), we examine the next character of the string ***type*** (if it is equal to ' \0', the loop terminates). This character stands for a typecode which determines the type of the next argument to be read: we use a **switch** statement to read this argument (using the macro **va_arg**) and assign its value to a variable of the appropriate type. If the character does not correspond to any valid typecode, the execution of the function should be terminated; the program prints an error message and exits both the **switch** statement and the loop (using **goto**). Still, **va_end** must be called even in this case (❷) to perform a cleanup — the function then

exits cleanly and the program continues. We can see this in case of the first invocation of the function **types**, when the string determining types of arguments contains an invalid typecode (' `x`' in this case):

The printout of this program is:

```
A string: John
Invalid typecode!!!

An int  : 17
A string: John
A string: Mary
A double: 1

An int  : 17
A double: 19.5
A double: 1.5
A string: OK
An int  : -1
An int  : 8
```

To simplify the use of functions with variable arguments, all “small” integer arguments are converted to **int** before pushing them onto the stack; similarly **float** s are promoted to **double** s.

One has to remember that in case of such functions, strict type checking is not possible. Suppose, for example, that 1 (without a dot) was used as the last argument in the second invocation of **types**. That would mean that the argument is of type **int** (which is a four-byte value). However, the corresponding typecode is 'D' (**double**), so the **va_arg** macro would read 8 bytes from the stack and interpret them as a **double**. Such an error could easily escape our attention undetected causing the program to crash, or, even worse, to give wrong results.

Functions with variable-length argument list are therefore dangerous and should better be avoided.

In new C++11 standard there are other and better ways to pass undefined in advance number of arguments to functions — we will come to this later.

11.6 Reference arguments

References (see section on references) can be used as both arguments and return values of functions.

Let us first consider reference arguments. Suppose a function is declared as (note the ampersand):

```
void fun(int& k) {
    k = k + 2;
}
```

This means that an argument of type **int** will be passed *by reference*, i.e., that the name ***k*** in the function will be *another name* (an alias) of an integer variable used as the argument in function's invocation. In other words, no copy of argument is made and pushed onto the stack: the function will have access to the *original* variable which appeared as the argument. We can use the above function like this:

```
int m = 1;
fun(m);
cout << "m = " << m << endl;
```

The variable ***m*** was used as the argument, so this variable (and *not* its copy) will be accessible in the function under the name ***k***. The function increments ***k*** by 2, but ***k*** is in this invocation just another name of ***m***, so after the flow of control returns to the caller, the value of ***m*** is modified: in the third line, 3 will be printed.

Note that the form of function invocation looks exactly the same as if ***m*** was passed by value. But if it were a call by value, ***m*** could not be changed, because the function would “see” and modify only its copy. The fact that looking at an invocation we are not able to tell if this is a call by value or by reference is very confusing. It can lead to erroneous interpretation of a code. To avoid such errors, one has to look carefully at declarations (or definitions) of function to see if there is an ampersand there or not. As we will see, despite of this danger, reference arguments are extremely convenient (and sometimes cannot be avoided).

If a parameter declared as a reference will be another name of a variable used as an argument, this variable has to exist! No local copy is created. But what will happen if one passes, as an argument, an expression which has a value, but is not a l-value? The parameter will be another name of what? Or one can use an l-value, but of different type than the type of the corresponding reference parameter. Will a name of a **double** be another name of an **int**? That would lead to chaos.

In both cases the compiler will detect an error. There is, however, an exception to this rule. If a reference parameter is of a **const** type, both cases mentioned above *can* be valid. A temporary variable of the appropriate type will be created and initialized with the value of the argument. In the function, the name of the reference parameter will be “another name” of this temporary variable, which will be destroyed after the function's termination. Note that

this will not cause any chaos, as the parameter was declared as **const** anyway, so the user knows in advance that the variable passed as an argument will not be modified.

Reference parameters are very convenient when passing arguments of a substantial size. Of course, variables of built-in types are usually “small”, but objects of user-defined types can be large. Using references can save copying such objects, creating local variables etc. Therefore, it is very often much more efficient. There are situations, when copying is just impossible (for eample, for objects representing streams). Reference arguments are then a natural choice.

On the other hand, when passing arguments by reference, one has to remenber that the function will has access to the original and will be able to modify it, what is not necessarily our intention. This can be avoided by declaring types of reference parameters as **const**. The argument itself does not have to be **const**, as the conversion from **Type** to **const Type&** is a trivial standard conversion (see section on conversions).

Reference arguments are very useful when modification of argument is just what we do want. For example, a function could calculate two or more values we want to know: normal mechanism of returning values by the **return** statement can give us one such value, but what with the rest of them? This can be easily solved by reference arguments which are modified by the function, as these new values are visible in the calling program. Of course, we could also use pointer arguments to achieve this goal (in pure C it is the only way). We pass the *address* of an existing variable and the function, knowing the address, has access to the variable pointed to by this address. The address is sent by value (so the function accesses its copy), but the address itself is the address of the original variable.

The program below illustrates various ways of passing arguments to functions:

P73: vrpe.cpp Passing arguments by value, reference and pointer

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void fun(int, int&, int*);
5.
6.  int main() {
7.      int a = 1, b = 2, c = 3;
8.
9.      cout << "Before: a = " << a << " b = " << b
10.                            << " c = " << c << endl;
11.     fun(a, b, &c);
12.
13.     cout << "After : a = " << a << " b = " << b
14.                            << " c = " << c << endl;
15. }
16.
17. void fun(int x, int& y, int* z) {
18.     x = 2*x;
19.     y = 3*y;
20.    *z = 4*(*z);
21. }
```

The first argument is passed by value. Therefore, the variable ***x*** in the function will be a *local* variable initialized with the value of ***a***. The function modifies ***x***, but this will not influence the value of ***a*** from the **main** function. The second argument, ***b***, is passed by reference (the second parameter was declared as **int&**). This means, that during this

particular invocation of the function **fun**, the variable ***y*** will be just another name of ***b*** from **main**. Any modification of ***y*** inside the function will be then equivalent to changing the value of ***b*** from **main**. The third argument is passed by pointer — the type of the parameter is **int***. As the function expects the address of an **int**, what we pass is not ***c*** as such, but rather its address, i.e., ***&c***. In the function, ***z*** is a local variable of pointer type, but initialized with the address of ***c***. Therefore, ***z*** from the function points to ***c*** from the **main**; consequently, ***z** denotes the **int** variable pointed to by ***z***, i.e., ***c***. Any modification of ***z** is a modification of ***c***, what can be seen from the output:

```
Before: a = 1 b = 2 c = 3
After : a = 1 b = 6 c = 12
```

References to arrays need some special consideration. If one declares reference to an array as the type of a parameter, will the size of this array be known in the function? It will be, because declaring an array type we *must* specify its size. Let us look at an example:

P74: arrref.cpp Reference to an array

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void funarr(double[]);                    ❶
5.  void arrref(double (&)[6]);               ❷
6.
7.  int main() {
8.      double arr[] = {1,2,3,4,5,6};
9.      cout << "sizeof double : " << sizeof(double)  << endl;
10.     cout << "sizeof double*: " << sizeof(double*) << endl;
11.     cout << "sizeof arr in main: " << sizeof(arr) << endl;
12.     funarr(arr);
13.     arrref(arr);                          ❸
14. }
15.
16. void funarr(double t[]) {
17.     cout << "sizeof t w funarr : " << sizeof(t) << endl;
18. }
19.
20. void arrref(double (&t)[6]) {             ❹
21.     cout << "sizeof t w arrref : " << sizeof(t) << endl;
22. }
```

The parameter of **funarr** declared on line ❶ is of type **double[]**, i.e., is equivalent to the pointer type **double***. Inside the function the local variable ***t*** is of type **double***, and its size is therefore 8 (or 4). The size of an array pointed to by this pointer is not known in the function — in order to use it we would have to pass it as a separate argument. But the type of the parameter of **arrref** (declared on line ❷) is the *reference* to an array. This is "true" reference to an array — not converted to pointer or something like this. Therefore, the size must be specified: the type is '`double (&)[6]`', i.e., *reference to six-element array of **double** s*. The parentheses around '`&`' are necessary here: without them the type would be *six-element array of references to **double** s* — something like this does not exist!

There are no arrays of references.

Similarly, on line ❹, '`double (&t)[6]`' means ***t** is the reference to six-element array of **double** s*, or *identifier **t** is another name of a six-element array of **double** s which was used in function's invocation as the corresponding argument*. The size has to be specified,

five-element array is of different type than six-element array. Therefore, the function **arrref** “knows” the size of the array ***t***:

```
sizeof double : 8
sizeof double*: 8
sizeof arr in main: 48
sizeof t w funarr : 8
sizeof t w arrref : 48
```

This, of course, does not mean that we have solved the problem of passing sizes of arrays to functions, as it is implemented, for example, in Java. Our function **arrref** will work *only* for six-element arrays. Try to change the size of ***arr*** by adding one element. The function **funarr** will not notice it, but the invocation from line ❸ will not even compile!

Prev: 11.5 Variable-length argument lists **Up:** 11. Functions **Next:** 11.7 Functions returning references

T.R. Werner, September 27, 2023; 22:32

11.7 Functions returning references

Functions can return references. The following declaration

```
int& fun(int);
```

declares a function which returns a reference to an **int**, or, as we say, returns an **int** by reference. It means that an expression like `fun(3)` is another name of an **int** which appeared in the **return** statement terminating the execution of the function.

Functions cannot return references (or pointers) to their local variables: such variables do not exist after the return. However, functions can return a local variable by value: after the return the variable does not exist, but the copy of its value, after a conversion, if necessary, is passed to the caller through the stack, processor's register or in another way — this is an implementation detail of no interest to the user.

The fact that invocation of a reference-type function is another name of an existing variable means that it is an l-value. It can, for example, appear on the left-hand side of an assignment. Such construct looks somewhat bizarre but can be perfectly valid — look at the invocation of the function **funmax** in the following program:

P75: varepoe.cpp Various types of arguments and return values

```
#include <iostream>
#include <cmath>
using namespace std;

double powers(double&,double*);
int*   square(int*);
int&   funmax(int[],int);

int main() {

      // reference and pointer arguments
    double u = 4, v;
    double cube = powers(u, &v);                    ❶
    cout << "Cube: " << cube    << "; square: "
         << u << "; sq. root: " << v  << endl;

      // even this is OK
    int i = 4;
    cout << "20? : " << ++*square(&i)+3 << endl;    ❷

      // returning reference
    int tab[] = {1,4,6,2};
    cout << "Array before: ";
    for ( i = 0; i < 4; i++ ) cout << tab[i] << " ";
    cout << endl;

    funmax(tab,4) = 0;                              ❸

```

```
29.     cout << "Array after : ";
30.     for ( i = 0; i < 4; i++ ) cout << tab[i] << " ";
31.     cout << endl;
32. }
33.
34. double powers(double& u, double* v) {                    ❹
35.     double x = u;
36.     u *= u;
37.     *v = sqrt(x);
38.     return u*x;
39. }
40.
41. int* square(int* p) {                                    ❺
42.     *p *= *p;
43.     return p;
44. }
45.
46. int& funmax(int* tab, int ile) {                         ❻
47.     int i, ind = 0;
48.     for ( i = 1; i < ile; i++ )
49.         if ( tab[i] > tab[ind] ) ind = i;
50.     return tab[ind];
51. }
```

The function **powers** takes one pointer argument and one reference argument. We pass, by reference, a positive number as the first argument; the function calculates the square, the cube and the square root of the number passed (❹). The square is assigned back to variable ***u***; as this is a reference parameter, this value will overwrite the original value (which was 4) and this new value will be visible in the **main** function after the return.

We pass (by value) a copy of the *address* of the variable ***v*** as the second argument (❶). The function puts the value of the square root of the first argument to the variable pointed to by *pointer* ***v*** — this will be the **double** variable ***v*** from the **main** program. Finally, the function returns the cube of the first argument.

After invoking the **powers** function the cube of the original argument ***u*** is in ***cube***, its square is the new value of ***u***, and the square root is in ***v***; one can see this from the printout:

```
Cube: 64; square: 16; sq. root: 2
20? : 20
Array before: 1 4 6 2
Array after : 1 4 0 2
```

Quite a tricky construct is used on line ❷. The function **square** (❺) calculates the square of the argument passed by pointer (we pass the address of variable ***i***). The result is written into the variable pointed to by the pointer (i.e., onto ***i*** in our case) and the same pointer is returned by the function (note that this is not a pointer to a local variable: this is the address of an existing variable which came from the **main**). Therefore, the value of `square(&i)` is the address of ***i***, and the value of ***i*** is now 16 (= 4×4), as the function squared the argument and put the result back onto this variable. We now dereference this pointer, and `*square(&i)` is the same as ***i*** with the value 16. This value is now incremented and then 3 is added to the result, so we finally get 20 as the value of `++*square(&i)+3`. Such strange looking expression appear quite often in real programs (although perhaps they should not).

Let us now have a look at the function **funmax** (❻). We pass, in a standard way, an array and its size (❸). The function looks for the maximum element of the array and returns this element *by reference*. That the return is by reference is not seen from the **return** statement, but can be checked by inspecting declaration and/or definition. Therefore, the expression `funmax(tab,4)` can be viewed as another name of a single **int** which is the

largest element of the array (i.e., another name of ***tab[2]***). As this expression stands on the left hand side of an assignment, the element found by the function is assigned the value of 0 — this can be verified by looking at the output.

Prev: 11.6 Reference arguments **Up:** 11. Functions **Next:** 11.8 Recursive functions

11.8 Recursive functions

Recursive function is a function which calls itself, either directly, or indirectly (e.g., **f** is recursive if **f** calls **g** and **g** calls **f**). Of course, the programmer is responsible for designing every recursive function in such a way that the chain of self-calls eventually stops. Usually there is a condition which is checked before a self-call; if this condition is not satisfied, the function exits without calling itself.

A classic (and useful) example of a recursive function is the function calculating the greatest common divisor of two natural numbers using the well known algorithm described by Euclid in Proposition VII.2 of his *Elements* (this is a special case of a more general — and much earlier — algorithm attributed to Theaetetus). Thanks to recursion (**recurrence**), the algorithm can be coded in an amazingly compact way — the function **gcd** in the following program contains essentially only one line:

P76: gcd.cpp Euclid's algorithm as an example or recurrence

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int gcd(int a, int b) {
5.      return b == 0 ? a : gcd(b, a % b);
6.  }
7.
8.  int main() {
9.      int x, y;
10.
11.     x = 732591; y = 1272585;
12.     cout << "gcd(" << x << "," << y << ")\t= "
13.          << gcd(x,y) << endl;
14.
15.     x = 732591; y = 1270;
16.     cout << "gcd(" << x << "," << y << ")\t= "
17.          << gcd(x,y) << endl;
18. }
```

The only statement of the **gcd** function is a **return** statement. However, as long as ***b*** is non-zero, the expression after a colon will be evaluated and the function will be called again (with different arguments). From mathematical consideration we know that eventually, after a finite number of steps, ***b*** will become 0. Then the condition '`b==0`' will return **true** and **gcd** will not be called any more:

```
NWD(732591,1272585)      = 129
NWD(732591,1270)         = 1
```

Self-invocation of a function is basically a normal function call: all local variable are created in each invocation separately, the mechanism of passing arguments is the same as for non-recursive functions. The function **gcd**, when invoked, creates local variables ***a*** and ***b*** and then, evaluating the expression after the colon, calls the same function **gcd** and waits for the result to return. This next “incarnation” of the function creates on the stack its own

variables ***a*** and ***b***, calls **gcd** again and waits for the result to return, etc. When ***b*** becomes zero, the function returns the value of ***a***, which is then returned by the previous incarnation, which is then returned by the previous incarnation, etc.

It is not unusual that algorithms coded as recursive functions are very compact and easy to write and understand. However, recursion can be dangerous, especially if a functions calls itself more than once. If recursive calls are not quickly "tamed", number of calls and/or size of the stack required can explode exponentially. A naïve implementation of a function which calculates Fibonacci numbers can serve as an example. Let us recall that Fibonacci numbers are defined by the following (recursive!) formulae:
$F_n = n$ for $n < 2$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

The very definition of Fibonacci numbers asks for a recursive function, like the one in the following program:

P77: fibo.cpp Fibonacci numbers: bad example of recurrence

```
1.  #include <iostream>
2.  #include <iomanip>
3.  using namespace std;
4.
5.  int counter;
6.
7.  int fib(int n) {
8.      counter++;
9.      return (n < 2) ? n : fib(n-1) + fib(n-2);     ❶
10. }
11.
12. int main() {
13.     cout << "\n  i    Fib(i)    # of calls\n"
14.             "---------------------------" << endl;
15.     for (int i = 10; i <= 43; i += 3) {
16.         counter = 0;
17.         int w = fib(i);
18.         cout << setw(3)  << i << setw(12) << w    ❷
19.              << setw(12) << counter << endl;
20.     }
21. }
```

The function **fib** contains (❶) *two* invocations of itself. We introduced a *global* variable ***counter*** which is incremented every time the flow of control enters the function **fib** and is zeroed before calculations for the next integer. In this way we are able to print not only consecutive Fibonacci numbers but also the information on how many calls it takes to calculate these numbers. The execution of the program takes quite a few seconds; most of the time is spent to evaluate the last value of F_{43}. As seen from the printout, in order to calculate F_{43} almost one and a half billion function calls were needed!

```
  i    Fib(i)    # of calls
---------------------------
 10        55          177
 13       233          753
 16       987         3193
 19      4181        13529
 22     17711        57313
 25     75025       242785
 28    317811      1028457
 31   1346269      4356617
 34   5702887     18454929
 37  24157817     78176337
```

```
40    102334155    331160281
43    433494437   1402817465
```

Of course, the same function can be implemented in a non-recursive (“iterative”) way: then it takes a fraction of a second to execute. [The manipulator **setw** on line ❷ was used just to format the output; it will be described in section on i/o operations .]

Prev: 11.7 Functions returning references **Up:** 11. Functions **Next:** 11.9 Static functions

T.R. Werner, September 27, 2023; 22:32

11.9 Static functions

Functions can be declared with modifier **static**. Its meaning is different for global functions (declared in global scope; outside of classes) and for member functions of classes. This latter case will be described later (see section on classes); here we will focus our attention on static functions declared in the global scope.

Without the **static** modifier, a function declared globally becomes “truly” global: it will be accessible in every compilation unit where it is declared, no matter where it was defined (of course, it has to be eventually defined in one of compilation units, so the definition can be found by the linker). However, if it is **static**, it belongs to the scope of a compilation unit in appears in: it will be not “exported” (this is similar to static global variables — see section on storage classes). This mechanism makes it possible to use the same name for different functions in different compilation units: there will be no name clash if they are **static**. However, the recommended way to achieve similar goals is to use **namespace** s which we will describe in section on namespaces .

11.10 Inlined functions

Both global functions and member functions of classes can be **inlined** by the compiler. Inlining means that the code of a function is inserted into the executable in every place where the function is invoked — there will be *no* actual function call: everything what the function does is put directly into the code. Therefore, in the resulting executable, this function will not appear as a separate function.

Suppose we have a very simple function which just multiplies its argument by 2:

```
int fun(int m) {
    return 2*m;
}
```

Note that every invocation of the function is a quite complicated process: arguments and return address have to be pushed on the stack, flow of control must jump to another part of the code and then back to where it was before the call, etc. The code would be perhaps a little larger, but the execution faster if this multiplication was put directly into the program in every place it is needed, without invoking any function. On the other hand, suppose that we realize that the function needs a modification. We would have to look for all places where this multiplication appears, be careful about possible name clashes, context, etc. In order to avoid such complications, we can leave this convenient function as it is, but tell the compiler to inline it. We do it by adding the specifier **inline** to the declaration/definition of a function, e.g.,

```
inline int fun(int m) {
    return 2*m;
}
```

By adding the specifier **inline**, we are telling the compiler not to compile this function, but to put its full code directly where it is used, as many times as needed. If we now want multiplication by 2 to be replaced by multiplication by 3, we change the definition of our function and the compiler will take care of placing this change everywhere where it is relevant. To be able to do it, the compiler must know the *definition* of the function.

The definition of an inlined function must be known in every place of the program when this function is invoked.

Definition, not only declaration. Compiler has to put the function's body into the code, so the declaration is not enough. Normally, functions can be declared many times but defined once only. For inline function the situation is different: they have to be *defined*, not only declared, in each compilation unit they are used. Moreover, the definition must appear lexically before function is used. The most convenient way to ensure that this will be the case is to define the inlined function in a header file and then include this header into every compilation unit which will use this function.

It is very important to remember that specifying a function as **inline** does not guarantee that the function will actually be inlined. This is often too difficult for the compiler — in such

cases the compiler gives up and does not inline the function: it will be compiled in the normal way. Whether a function will or will not be inlined depends on the compiler, so we should never assume that our function will be actually inlined. Usually a function will not be inlined if it is recursive or if its flow of control is complicated (many **if** s, especially inside loops). Therefore, to qualify for inlining, a function should be small and simple but invoked many times (e.g., inside deeply nested loops). Generally, functions should be written as not inlined and only then, after careful analysis (using profilers), some of them, crucial for program's efficiency, should be inlined.

Prev: 11.9 Static functions **Up:** 11. Functions **Next:** 11.11 Function overloading

11.11 Function overloading

In C++ (but not in C), two or more functions in the same scope can have the same name. This is called function **overloading** Of course, the compiler, when it encounters a function invocation, must decide which of the overloaded versions is meant — for this to be possible, overloaded functions have to differ in such a way that the compiler, looking at the invocation, is able to deduce the proper version.

A necessary (but not sufficient) condition that the overloaded functions have to meet is that their **signatures** differ.

> **Signature is defined by the function's name and number and type of its parameters, *not* taking into account parameters with default values assigned. The type of the function (type of its return value) is usually not considered a part of its signature.**

For example

```
int    fun(double x, int k =0);
double fun(double z);
```

declares two different functions, but with the same signature, namely '`fun(double)`'. Invocation '`fun(1.5)`' would be a valid invocation of both these functions. Therefore, two such functions cannot be visible in the same scope — this overloading would be illegal.

However,

```
double fun(int);
double fun(unsigned);
```

is valid. The siganture is different and, e.g., the invocation `fun(15)` matches the first version, because it does not need any conversions (literal '15' is of type **int**), while invoking the second with `fun(15)` would require a conversion.

Different signatures are not sufficient for the overloaded functions to differ sufficiently. For example, two functions

```
void fun(int i);
void fun(int& i);
```

have different signatures (`fun(int)` and `fun(int&)`), but the invocation `fun(k)`, with ***k*** of type **int**, would match, without any conversions, both of them. This overloading would be then invalid. For a similar reason

```
void fun(int tab[]);
void fun(int * p);
```

or

```
void fun(tab[3][3]);
void fun(tab[5][3]);
```

would be invalid as well, as the first dimension of an array is irrelevant to its type. But

```
void fun(tab[3][3]);
void fun(tab[3][5]);
```

is correct, because multidimensional arrays differing in a dimension other than the first are of different types.

As we know, argument of type **T** can be used in invocation of a function with the corresponding parameter of type **T**, **const T** or **volatile T** — overloaded functions cannot differ only in type of such a parameter. Therefore

```
int fun(int k);
int fun(const int k);
```

would be invalid. Note that **const int** does not make much sense as a parameter type, because the call is by value, so the function gets only a copy of the value: the original cannot be changed anyway, even without **const**.

The types **T***, **volatile T*** and **const T*** (and, analogously, **T&**, **volatile T&** and **const T&**) *are* sufficiently different: looking at the invocation, the compiler can see if the argument is **const** or not. The overloading

```
int fun(int& k);
int fun(const int& k);
```

is therefore valid.

It can happen that two or more overloaded functions match an invocation after some standard conversions of arguments to the types of parameters. How to resolve which one should be called?

The process of searching for the best much among a few “candidates” proceeds in several steps, in which the quality (degree) of match is checked:

Exact match. All arguments have exactly the types declared as the types of corresponding parameters.

Match after trivial conversions. In order to obtain the full match, only trivial (minor) coversions are necessary. Trivial conversions are (`T` is a type, `fun` is a function — function pointers will be described in the next section):

T	→	T&
T&	→	T
T[]	→	T*
fun()	→	(*fun)()
T	→	const T
T	→	volatile T
T*	→	const T*
T*	→	volatile T*

Table 11.1: Trivial conversions

Match after standard promotions. In order to obtain the full match, only standard promotions are necessary. This can be an integer promotion (to **int**) or floating point promotion (to **double**) — see section on standard conversions (e.g., **char** → **int**).

Match after "another" standard conversion. In order to obtain the full match, standard promotions, other than promotions, are needed (e.g., **int** → **double** or *vice versa*).

Match after a user-defined conversion. In order to obtain the full match, a user-defined conversion is needed — we will explain how to define such conversions in section on conversions .

Match to a function with variable-length argument list. As the last resort, matching a call to a function with variable-length argument list is tried (see section on variable-length argument list).

The search for the best match is a complicated process, often not so intuitive. Let us consider the program:

P78: match.cpp Matching a call to overloaded functions

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  string fun1(   int) { return "\'int\'\n";    }
6.  string fun1(  char) { return "\'char\'\n";   }
7.  string fun1(double) { return "\'double\'\n"; }
8.
9.  string fun2( short) { return "\'short\'\n";  }
10. string fun2(double) { return "\'double\'\n"; }
11.
12. int main() {
13.     int    kin =   0;
14.     char   kch = '\0';
15.     float  kfl =   0;
16.     double kdo =   0;
17.
18.     cout << "fun1(   int) -> " << fun1(kin);
19.     cout << "fun1(  char) -> " << fun1(kch);
20.     cout << "fun1( float) -> " << fun1(kfl);  ❶
21.     cout << "fun1(double) -> " << fun1(kdo);
22.
23.     cout << "fun2( float) -> " << fun2(kfl);  ❷
24.   //cout << "fun2(  char) -> " << fun2(kch);
25.   //cout << "fun2(   int) -> " << fun2(kin);
26. }
```

We define three functions with the same name **fun1**. All the functions return a string allowing us to identify which of these function has actually been invoked. The functions are called in **main** with arguments of various types. The ouput reads:

```
fun1(   int) -> 'int'
fun1(  char) -> 'char'
fun1( float) -> 'double'
fun1(double) -> 'double'
fun2( float) -> 'double'
```

As we can see, if there is a function with the parameter's type exactly matching the type of argument, this function will be selected. In case of the argument of type **float** (line ❶), the function with **double** parameter was selected, because this match needs only one standard promotion **float** → **double** (**float** → **int** is also a standard conversion, but not a standard promotion).

The function **fun2** is more interesting. There are two overloaded versions: with parameter

of type **short** and **double**. There is no problem with calling the function with **float** argument (❷) — conversion **float** → **double** is a standard promotion. However, the invocation with argument of type **char** is illegal!r One could expect, that integer promotion **char** → **short** is much more “natural” than conversion **char** → **double**. However, the *standard* integer promotion leads directly to **int**, the promotion **char** → **short** is not a standard promotion but falls into category “another” standard conversion, exactly as the conversion **char** → **double**. Therefore, both functions turn out to be equally good (or equally wrong) — such a call will be detected as an error. Similarly, the last line is also wrong. It seems that the promotion **int** → **double** would be better than the narrowing conversion **int** → **short**. However, they are equally wrong. The former is a promotion, but not a standard integer promotion, so it falls into category “another” standard conversion, exactly as the latter.

Of course, such non obvious overloadings can lead to many hard to detect errors and generally should be avoided.

Prev: 11.10 Inlined functions **Up:** 11. Functions **Next:** 11.12 Function pointers

11.12 Function pointers

Function pointers play a very important rôle in both C and C++. As we remember, pointers are variables whose values are addresses of other variables: numbers, strings, arrays etc. Defining a pointer variable, one has to specify the type of variables this pointer can point to. This is necessary, e.g., to make the pointer arithmetic possible (see chapter on data types).

A variable is a piece of data stored in the computer memory. As such, it has an address. A function is a piece of binary code and, of course, must be stored somewhere in memory as well. It justifies the notion of function's address and, consequently, the notion of a pointer to a function. However, there is a difference between “normal” and function pointers. If a pointer points to a variable of a “regular” type, it is known what is the size of the object which is pointed to. Of course, every function is different and the size of its code is usually unknown (and can depend not only on platform or compiler, but even on compilation options used). Therefore,

Pointer arithmetic does not apply to function pointers.

Generally, pointers point to objects of a well defined type: information on type is necessary not only to know the size, but to know how this particular sequence of bits is to be interpreted and what operations it can be subject to. In case of functions pointed to by function pointers, the compiler must know what the type of the function is; otherwise it would not be able to check if operations performed on this function (e.g., its invocation) are valid. Therefore, declaring a function pointer, one has to specify the type of functions that this pointer will point to (i.e., the number and type of parameters and of returned value).

How one can declare/define a function pointer? Let us show this on a few examples (it would be useful to recall the rules of “reading” definitions of types from section on data types).

What is the type of ***fun*** after the following definition:

```
int (*fun)(int);
```

We proceed according to the rules from the section just mentioned : VARIABLE ***fun*** IS — closing parenthesis to the right, so we look to the left — POINTER TO — we leave parentheses and we look to the right — FUNCTION WITH ONE PARAMETER OF TYPE **int** RETURNING — we look to the left — VALUE OF TYPE **int**. Now ***fun*** is a function pointer which can hold, as its value, the address of a function: not of any function though, but only of a function which takes one argument of type **int** and returns an **int**. Exactly as after

```
int *k;
```

the pointer variable ***k*** exists but has no sensible address assigned as its value, our pointer ***fun*** exists but does not point to any function. Note also, that parentheses around `*fun` could not have been omitted:

```
int *fun(int);
```

would read: VARIABLE ***fun*** IS — opening round parenthesis to the right — A FUNCTION WITH ONE ONE PARAMETER OF TYPE **int** RETURNING — we look to the left — A POINTER TO — further to the left — A VARIABLE OF TYPE **int**. That would be a *declaration* of a function **fun**.

Another example:

```
double (*fun[3])(double);
```

reads: VARIABLE ***fun*** IS — opening bracket to the right — A THREE-ELEMENT ARRAY OF — we look to the left — POINTERS TO — we leave the parentheses and look to the right — FUNCTIONS WITH ONE PARAMETER OF TYPE **double** RETURNING — we look to the left — VALUES OF TYPE **double**. So now ***fun*** is an array of function pointers; each of them can point to a function of type **double** → **double**.

If we have an integer variable ***k*** and a pointer ***p*** of type **int***, then ***p*** can be assigned a value of the address of ***k*** with

```
p = &k;
```

Analogously: when we have a function, say **fun**, of type, say, **int** → **int** and a function pointer variable ***pfun*** which is of the corresponding type, i.e., can point to functions of type **int** → **int**, then the assignment will have the usual form:

```
pfun = &fun;
```

However, the same assignment can also be written as

```
pfun = fun;
```

because conversion from ***fun*** to ***&fun*** is for function pointers a trivial conversion (this is *not* the case for other, "normal" types of pointers!). One has only to remember *not* to add parentheses after the name of function: with parentheses it would be a function call, e.g.,

```
pfun = fun();
```

would mean *invoke **fun** without arguments and assign the returned value to **pfun***.

Let us consider an example:

P79: funpoint.cpp Function pointers

```
1.  #include <iostream>
2.  #include <cmath>
3.  using namespace std;
4.
5.  const double PI = 4*atan(1.);
6.
7.  double ours(double);
8.
9.  int main() {
10.     double (*f)(double);                          ❶
11.
12.     f = sin;                                      ❷
13.     cout << "sin(PI/2) =  " << (*f)(PI/2) << endl;  ❸
14.
15.     f = &cos;                                     ❹
16.     cout << "  cos(PI) = "  << f(PI)      << endl;  ❺
17.
```

```
18.     f = ours;                                   ❻
19.     cout << "  ours(3) =  " << f(3)       << endl;  ❼
20. }
21.
22. double ours(double x) {
23.     return x*x;
24. }
```

We include the header `cmath` to have access to mathematical functions like **sin** and **cos**. On line ❶ we define a function pointer ***f***, which can to point to functions taking one **double** and returning **double**. This pointer is then assigned a value on line ❷: this will be the address of the function **sin** (as we already know, we can leave out the '&' operator if we wish so). On line ❹ the value of the pointer is modified — now it points to function **cos**. Finally, we modify it once more On line ❻: now it will point to our own function **ours** which is of the correct type (**double** → **double**). As we can see, when assigning to a function pointer, we can use on the right hand side the name of a function with or without the address operator ' `&`'.

On lines ❸, ❺ and ❼ the function currently pointed to by the pointer ***f*** is invoked — each time it is a different function, as we can see from the output:

```
sin(PI/2) =  1
  cos(PI) = -1
  ours(3) =  9
```

The variable ***f*** is a pointer to a function. Therefore, formally, we should first dereference it (using ' `*`'), and then use the result as a name of a particular function. This is what we do on line ❸ (the partentheses around `*f` are necessary, as function call operator, i.e., parentheses, has higher precedence than the dereference operator '*'). However, as we can see on lines ❺ and ❼, in case of function pointers the dereference operator is not necessary: the compiler will perform the corresponding conversion itself (this is, again, *not* true for “normal” pointers).

Function pointers (or rather their values, i.e., addresses of functions) may be used as arguments and return values of other functions. Declaration of parameter type looks like declaration of a function pointer (in declarations, but not definitions, the name of a parameterr can be left out). Thus

```
double fun( double (*f)(double), double a, double b) {
    return f(a) + f(b);
}
```

would be the definition of function **fun** which takes three arguments: one is a pointer to a function of the type **double** → **double**, and the two remaining are **double** s. The function could be then used like this:

```
#include <cmath>
const double PI = 4*atan(1.);
// ...
double result = fun(sin, 0, PI/2);
```

The result should be equal to

$$\sin(0) + \sin(\pi/2) = 0 + 1 = 1$$

As usually, for declarations names of parameters may be left out, so the declaration of the function above could have been written as

```
double fun(double(*)(double),double,double);
```

what definitely looks somewhat confusing.

Specification of type is often quite complicated for function pointers. One can make life a bit easier by assigning a name (alias) to such types; this can be done by using **typedef** specifier (see section on this specifier). In our case, we could give a name (e.g., **FUNDtoD**) to the type *pointer to function taking one **double** and returning **double***:

```
typedef double (*FUNDtoD)(double);
```

and then use this name in, e.g., declaration of our function **fun**:

```
double fun( FUNDtoD, double, double);
```

or in declarations/definitions of function pointers:

```
#include <cmath>
// ...
FUNDtoD f = sin;
// ...
f = atan;
// ...
```

Functions taking functions as arguments are nothing exotic. Suppose, for example, that we want a function which finds roots (zeros) of another function. We would not like to write separate root-finding function for every function whose roots we are interested in. We would rather code a general root-finding algorithm once and then use it for whatever function we are dealing with. A simple implementation of such a function follows:

P80: roots.cpp Function as parameter of function

```
1.  #include <iostream>
2.  #include <cmath>
3.  #include <cassert>
4.  using namespace std;
5.
6.  const double PI = 4*atan(1.);
7.
8.  typedef double (*FD2D)(double);            ❶
9.
10. double root(FD2D, double, double);         ❷
11.
12. double polyn(double x) {return 3 - x*(1 + x*(27 - x*9));}
13.
14. int main() {
15.     double r;
16.
17.     cout.precision(15);                    ❸
18.
19.     r = root(sin, 3, 4);                   ❹
20.     cout << "sin      " <<  r    << endl
21.          << "exactly: " <<  PI   << endl << endl;
22.
23.     r = root(cos, -2, -1.5);               ❺
24.     cout << "cos:     " <<  r    << endl
25.          << "exactly: " << -PI/2 << endl << endl;
26.
27.     r = root(polyn, 0, 1);                 ❻
28.     cout << "polyn:   " <<  r    << endl
29.          << "exactly: " <<  1./3 << endl;
30.
31.     r = root([] (double x) -> double {     ❼
32.                 return 3-x*(1+x*(27-x*9));
33.             }, 0, 1);
34.     cout << "lambda:  " <<  r    << endl
35.          << "exactly: " <<  1./3 << endl;
36. }
37.
```

```
38. double root(FD2D fun, double a, double b) {
39.     /* Finding root of function using bisection.
40.        fun(a) and fun(b)  must be of opposite sign */
41.     static const double EPS = 1e-15;
42.     double f, s, h = b-a, f1 = fun(a), f2 = fun(b);
43.
44.     if (f1 == 0) return a;
45.     if (f2 == 0) return b;
46.     assert(f1*f2 < 0);                    ❽
47.
48.     do {
49.         if ((f = fun((s=(a+b)/2))) == 0) break;
50.         if (f*f1 < 0) {f2 = f;b = s;}
51.         else          {f1 = f;a = s;}
52.     } while ((h /= 2) > EPS);
53.
54.     return (a+b)/2;
55. }
```

The first parameter of the function **root** (❷) is declared as a (pointer to) function of type **double** → **double**. For later convenience, we have given this type a name (alias), **FD2D**, using **typedef** (❶). The two remaining parameters of our function are both of type **double** — these will be abscissae of two points such that the root which is searched for is guarateed to be located in-between them. For the algorithm to work, the function must assume values of opposite sign in these two points. Whether this is actually the case, is checked (❽) by the macro **assert** from the header `cassert`. The macro takes a logical argument — a condition which should be met: if it is **false** (the condition is *not* met), the program will be interrupted with a comprehensible message sent to standard error stream (`stderr`). All **assert** statements can be disabled by defining the macro ***NDEBUG*** (e.g., `#define NDEBUG` in the source file or `-DNDEBUG` as a compilation option).

If the signs of function's values for arguments ***a*** and ***b*** are opposite and the function is continuous, the algorithm should, virtually always, work properly and give a solution, i.e., a root of the function in a given range.

In the main program we invoke **root** four times. In the first call (❹) we pass the function **sin**, and the range $[3, 4]$. As we know, the **sin** function has one root in this range, (viz., $x = \pi$); we know it, but the function does not, so it has to calculate it.

Then (❺), we look for the root of **cos** function in the range $[-2, -1.5]$ — that, as we know, should be $-\pi/2$. On line ❻, we search for a root of our own function, **polyn**, which has been defined before the **main**. One can easily check, that this function is a straightforward implementation of the following cubic polynomial

$$W(x) = 9x^3 - 27x^2 - x + 3$$

with roots at $x_{1,\,2} = \pm 1/3$ and $x_3 = 3$. Inside the segment $[0, 1]$ there is only one root, viz., $x = 1/3$.

Finally, on line ❼, we look for the root of the function equivalent to function **polyn** but this time defined as a lambda function (more on such functions in section on lambda functions).

After finding roots, we print them together with their exact values, which in this case are known to us. The call '`cout.precision(15)`' on line ❸ is necessary to print the results with greater precision than the default 6 significant digits (more about formatting output — see section on i/o operations).

As one can see form the printout

```
sin       3.14159265358979
exactly:  3.14159265358979

cos:      -1.5707963267949
exactly:  -1.5707963267949

polyn:    0.333333333333333
exactly:  0.333333333333333
lambda:   0.333333333333333
exactly:  0.333333333333333
```

the roots have been calculated exactly up to 15 significant digits.

It is also possible to create arrays of functions (function pointers), exactly as it is possible to have arrays of normal pointers. In the program below, we first define aliases for types which will be needed; therefore, after

```
typedef double (*TABFUN[])(double);
```

the name **ARRFUN** is the name of type *array of pointers which point to functions of type* ***double*** → ***double*** (❶, alias **ARRNAM** is defined in the new syntax from C++11). The name **ARRNAM** denotes the type *array of pointers to* ***char***, i.e., array of C-strings (which will contain names of functions).

P81: arrayfun.cpp Arrays of functions

```
1.  #include <iostream>
2.  #include <cmath>
3.  using namespace std;
4.
5.  typedef double (*ARRFUN[])(double);          ❶
6.  // typedef const char *ARRNAM[];
7.  using ARRNAM = const char* [];  // C++11
8.
9.  void funprnt(ARRFUN,ARRNAM,double);
10.
11. int main() {
12.     const double PIover4 = atan(1.);
13.
14.     ARRFUN arrfun = { sin,  cos,  tan };   ❷
15.     ARRNAM arrnam = {"sin","cos","tan"};
16.
17.     cout << "sizeof(ARRFUN) = " << sizeof(arrfun) << endl
18.          << "sizeof(ARRNAM) = " << sizeof(arrnam) << "\n\n";
19.
20.     for (int i = 0; i < 3; i++) {          ❸
21.         cout << "arrfun[" <<   i   << "](pi/4) = "
22.              << arrfun[i](PIover4) << " ("
23.              << arrnam[i]          << ")\n";
24.     }
25.
26.     funprnt(arrfun,arrnam,PIover4);
27. }
28.
29. void funprnt(ARRFUN f, ARRNAM t, double x) {
30.     cout << "\n";
31.     for (int i = 0; i < 3; i++) {
32.         cout << "funprnt: " << t[i] << " "
33.              << "value    " << f[i](x) << endl;
34.     }
35. }
```

We define an array of function pointers on line ❷ and initialize it with addresses of three

existing functions. We have not used the address operator '&', but it was used implicitly, so the array is not an array of functions (there is no such thing!) but an array of pointers — one can show it by printing its size, which is 24 (three pointers eight bytes each).

```
sizeof(ARRFUN) = 24
sizeof(ARRNAM) = 24

arrfun[0](pi/4) = 0.707107 (sin)
arrfun[1](pi/4) = 0.707107 (cos)
arrfun[2](pi/4) = 1 (tan)

funprnt: sin value    0.707107
funprnt: cos value    0.707107
funprnt: tan value    1
```

Actually, the standard does not require the size of function pointer to be the same as the size of normal pointers — function pointers may be implemented completely differently than other pointers. Therefore, it does not make any sense to cast such pointers to any other pointer type.

We invoke functions from the table in the loop (❸). Elements of the array, like ***arrfun[i]***, are function pointers, so they are automatically dereferenced when used with parentheses, i.e., when we invoke the functions that they point to. Notation `arrfun[i](PIover4)` denotes a call of function pointed to by the pointer which is the $(i+1)$-th element of the array ***arrfun*** — the value of ***Plover4*** (which is equal to $\pi/4$) is used as an argument.

The function **funprnt** takes an array of functions (function pointers), an array of C-strings and a **double**. In the body of the function, the pointers, which are element of the array, named ***f*** here, are called again (after implied dereferencing to functions).

Function pointer can also be returned by functions. Let us look at an example:

P82: funretur.cpp Returning function pointers

```cpp
#include <iostream>
#include <cmath>
using namespace std;

typedef double (*FUNDtoD)(double);                    ❶
typedef FUNDtoD ARRFUN[];

FUNDtoD funmax(ARRFUN,double);                        ❷

double fun0(double x) { return log(x); }
double fun1(double x) { return x*x;    }
double fun2(double x) { return exp(x); }
double fun3(double x) { return sin(x); }
double fun4(double x) { return cos(x); }

int main() {
    ARRFUN arrfun = { fun0, fun1, fun2, fun3, fun4 };  ❸

    FUNDtoD fun = funmax(arrfun,1);                   ❹

    int i;
    for (i = 0; i < 5; ++i)
        if (fun == arrfun[i]) break;

    cout.precision(14);
    cout << "Largest value at x=1 assumed by function no "
         << i << ".\nThe value is " << fun(1) << endl; ❺
```

```
}

FUNDtoD funmax(ARRFUN f, double x) {
    double m = f[0](x),z;
    int    k = 0;

    for (int i = 1; i < 5; i++) {
        if ( (z = f[i](x)) > m) {
            m = z;
            k = i;
        }
    }
    return f[k];
}
```

First (❶) we define an alias **FUNDtoD** for type *pointer to function of type **double** → **double*** and then, using it, another alias **ARRFUN** for type *array of pointers to functions of type **double** → **double***. Thanks to these aliases, the declaration of function **funmax** (i❷) is quite comprehensible: this is a function taking as the first argument an array of function pointers and returning a pointer to function. Without aliases defined by **typedef**, the declaration would be much more complicated and could be hard to understand. The task which the function **funmax** performs is to find and return that function among those passed in the array which assumes the largest value at abscissa passed as the second argument.

We define five simple function and then, in **main**, we initialize the array ***arrfun*** (of type **ARRFUN**) of function pointers with their addresses (❸). Next, we pass the array to function **funmax** (❹) and we store the result (which is a pointer of type **FUNDtoD**) in variable ***fun***.

Now in a loop, we find the index of the function in the array which is equal to the returned value, as we know that the function must have returned one of the pointers from the array. This allows us to print this index and call the function found by **funmax** (❺). We call the function for the same argument (equal to 1) to check, if the answer is correct. In our case the returned function should be (and indeed is) the **exp(x)** function (i.e., e^x), which for argument 1 assumes the value equal to the base of natural logarithm (approximately 2.71828). The printout shows that this is the case:

```
Largest value at x=1 assumed by function no 2.
The value is 2.718281828459
```

In C++ one can define **function objects** which are, in a sense, a generalization of a function — we will discuss them in section on function objects .

11.13 Lambda functions

Staring from version C++11 it is possible to define the so called **lambda functions**. These are anonymous functions which can be defined locally; the syntax is as follows:

```
[ capture ] ( parameters ) -> return_type { body }
```

A list of parameters is provided in parentheses, as for normal function. After the “arrow” we specify the return type. In some (even most) situations this part may be omitted and the compiler will deduce the proper return type itself — this will be **decltype** of the returned expression (we discussed **decltype** in section on data types). If there is no **return** statement in the body of the function, then **void** will be deduced.

We have square brackets at the beginning of the definition; they can be left empty what means that the function will only have access to data passed explicitly by arguments. However, we can put into these brackets comma separated symbols:

equal sign ('=')

the function will have access to *copies* of the values of all *local* variables from the current scope, except those which are mentioned explicitly (see below);

ampersand ('&')

the function will have access to *references* to all local variables from the current scope (i.e., to their originals, not copies) — again: except those which are mentioned explicitly;

`var`

where ***var*** is the name of a local variable: the function will have access to a *copy* of the values of ***var***;

`&var`

where ***var*** is the name of a local variable: the function will have access to the *reference* to variable ***var*** (i.e., its original).

For example, `[&,a]` means that the lambda function will have access to all local variables by reference, but variable ***a*** will be accessible by its current value. Similarly, `[=,&a,&b]` means that the lambda will contain copies of all local variables, but will see `originals' of ***a*** and ***b***.

The type of lambda function is not specified by the standard; it usually is different for any two lambdas. What is important, however, is the fact that these type are convertible to types **function<Type(Types)>** (from the header **functional**), where **Type** is the return type, and **Types** are comma separated types of parameters. Plain function pointers are converted to these types automatically. In many (but not all) cases, one can use the keyword **auto** to avoid having to specify the type explicitly.

Let us consider an example:

P83: lambdas.cpp Lambda functions

```
1.  #include <iostream>
2.  #include <functional>
3.  using std::cout; using std::endl;
4.
5.  double square(double x) {
6.      return x*x;
7.  }
8.
9.  void invoke(std::function<double(double)> f, double arg) {
10.     double res = f(arg);
11.     cout << "invoke(" << arg << ")=" << res << endl;
12. }
13.
14. int main() {
15.       // auxiliary lambda function
16.     auto print =
17.         [](double p1, double p2, double p3,
18.            double arg, double val) -> void
19.         {
20.             cout << " a=" << p1 << " b=" << p2
21.                  << " c=" << p3 << " x=" << arg
22.                  << " res=" << val << endl;
23.         };
24.
25.       // lambda function a*x*x+b*x+c
26.     int a = 1, b = 1, c = 1;
27.       // all local variables captured by value
28.     auto pol1 =
29.         [=](double x) -> double
30.         {
31.             double res = c+x*(b+x*a);
32.             print(a,b,c,x,res);
33.             return res;
34.         };
35.     cout << "pol1=" << pol1(2) << endl;
36.     a = b = c = 2;
37.     cout << "pol1=" << pol1(2) << endl << endl;
38.
39.       // all local variables captured by reference
40.     auto pol2 =
41.         [&](double x) -> double
42.         {
43.             double res = c+x*(b+x*a);
44.             print(a,b,c,x,res);
45.             return res;
46.         };
47.     cout << "pol2=" << pol2(2) << endl;
48.     a = b = c = 1;
49.     cout << "pol2=" << pol2(2) << endl << endl;
50.
51.       // a and c by reference, b and print by value
52.     auto pol3 =                                    ❶
53.         [&a,b,&c,print](double x) -> double
54.         {
55.             double res = c+x*(b+x*a);
56.             print(a,b,c,x,res);
57.             return res;
58.         };
59.     cout << "pol3=" << pol3(2) << endl;            ❷
60.     a = b = c = 2;                                 ❸
61.     cout << "pol3=" << pol3(2) << endl << endl;    ❹
62.
63.       // type specified explicitly
64.     std::function<double(double)> f = pol3;
65.     invoke(f,2);
66.       // converting 'plain' function pointer
67.     invoke(square,2);
68.     f = square;
69.     invoke(f,2);
70.       // lambda as an argument
```

```
71.     invoke([](double x) {return x*x*x;}, 3);
72.
73.       // void->void only needs brackets and a body
74.     [] {
75.         cout << "Done" << endl;
76.     }(); // construct a function and invoke it immediately
77. }
```

The **invoke** function takes a lambda function (or a function pointer, or a functor) and invokes it for a given argument. At the beginning of the **main** function (therefore, *inside* a function), we define an auxiliary lambda function **print** with empty capture (so all information will be passed by arguments). Let us notice that ***print*** is itself a local variable here. This lambda function is then used several times in the program. Afterwards we define, using the keyword **auto**, three simple functions (**pol1**, **pol2**, **pol3** — all are implementations of the same polynomial of the second degree $ax^2 + bx + c$) using different captures: some local variables will be accessible by values (their values are copied at the time of defining the lambda function), while others will be accessed by reference (so the function will see their modifications). For example, on line ❶ we define a lambda which captures current values of ***b*** (which is 1) and **print** and references to ***a*** and ***c*** (which also have values equal to 1). Invoking the lambda with $x = 2$ (line ❷), we get 7. Then we change the values of ***a***, ***b*** and ***c*** — they are now all equal to 2 (line ❸). However, value of ***b*** seen by the function is not modified, because the function sees a copy of the value as it was when the lambda was defined (i.e., 1). On the other hand, variables ***a*** and ***c*** are seen by reference, so the function will see the modifications and invocation from line ❹ will yield 12.

At the end of the program, we demonstrate conversions of plain function pointers and passing lambda function and function pointers to other functions (in this case, the function **invoke**). One can see that function pointers are implicitly converted to type **std::function<double(double)>**.

It is important to analyze the program and understand its printout:

```
 a=1 b=1 c=1 x=2 res=7
pol1=7
 a=1 b=1 c=1 x=2 res=7
pol1=7

 a=2 b=2 c=2 x=2 res=14
pol2=14
 a=1 b=1 c=1 x=2 res=7
pol2=7

 a=1 b=1 c=1 x=2 res=7
pol3=7
 a=2 b=1 c=2 x=2 res=12
pol3=12

 a=2 b=1 c=2 x=2 res=12
invoke(2)=4
invoke(3)=27
Done
```

In newer versions of the standard, parameters of lambdas may be declared with the **auto** keyword. Then the compiler can create several versions of the corresponding function inferring appropriate return type and types of parameters based on types of arguments used when calling the lambda. For example, the program:

P84: lambdagen.cpp Deducing types in lambdas

```
1. #include <iostream>
2.
3. int main() {
4.     using namespace std::literals;
5.
6.     auto pr = [] (auto e) {
7.         std::cout << "Result is " << e << '\n';
8.     };
9.     auto f = [] (auto e1, auto e2) {
10.         return e1 < e2 ? e1 : e2;
11.     };
12.     auto ri = f(3, 1);
13.     pr(ri);
14.     auto rs = f("Cindy"s , "Alice"s);
15.     pr(rs);
16. }
```

prints

```
Result is 1
Result is Alice
```

As we can see, one lambda „handles" various return types and types of arguments (this is possible due to overloading the **operator()** method in the class representing the lambda). Note that literals `"Alice"` and `"Cindy"` have the letter 's' at the end. This means that they should be treated as literals of objects of type **string**, and not as C-strings — then they would be of type **const char*** (such syntax is possible after including the namespace `std::literals`).

Values passed to the capture clause are normally treated as constants. However, using the keyword **mutable** after the list of parameters, we can make them mutable. Then the function represented by a lambda can modify them and these changes will be retained between subsequent invocations. In this way, for example, we can build lambdas representing generators: parameterless function for which invocations return subsequent values of a sequence. In the program below, lambda ***fibo*** (❶) will generate values of the Fibonacci sequence

$$F_0 = 0,\ F_1 = 1,\ F_2 = 1,\ F_3 = 2, \ldots,\ F_n = F_{n-2} + F_{n-1}, \ldots$$

and lambda ***triangle*** subsequent triangular numbers

$$t_0 = 0,\ t_1 = 1,\ t_2 = 3,\ t_3 = 6, \ldots,\ t_n = t_{n-1} + n, \ldots$$

P85: lambdamutable.cpp Lambdas with **mutable** option enabled

```
1. #include <iomanip>     // setw
2. #include <iostream>
3.
4. int main() {
5.     using std::cout; using std::endl; using std::setw;
6.
7.     auto fibo = [fp=-1, fn=1] () mutable  {      ❶
8.             int d = fp; fp = fn; return fn += d;
9.         };
10.
11.     auto triangle = [t=0, i=0] () mutable {      ❷
12.             return t += i++;
```

```
        };

    for (size_t i = 0; i <= 10; ++i)
        cout << setw(2) << i << ":" << setw(3) << fibo()
             << setw(3) << triangle() << endl;
}
```

Note that values ***fp*** and ***fn*** (and similarly ***t*** and ***i***) are *not* local variables from the surrounding scope: they are defined and initialized directly in the capture part and their type is deduced by the compiler automatically, based on the initializers. The program prints

```
 0:  0  0
 1:  1  1
 2:  1  3
 3:  2  6
 4:  3 10
 5:  5 15
 6:  8 21
 7: 13 28
 8: 21 36
 9: 34 45
10: 55 55
```

Note also that if the **mutable** option appears, the parentheses are needed even where there are no parameters. The return type, however, may be omitted if it can be deduced by the compiler.

Prev: 11.12 Function pointers **Up:** 11. Functions **Next:** 11.14 Function templates

11.14 Function templates

Very often we need many functions which have the same functionality but differ in type of arguments and/or of returned values. For example, we can think about a function which finds the maximum element of an array. We will have to write several versions of such a function — separately for arrays of **int** s, of **double** s, or of objects of class **Person** (assuming that persons can be compared). Such redundancy cannot be avoided, because one has to specify a type of every parameter, and if it is an array of **Person** s, one cannot pass an array of **int** s as the argument. And if it turns out that we need to do the same thing for **Animal** s, we will have to write yet another version of the function and recompile the module it is defined in.

And it is exactly such situations when function templates come to the rescue. We can create a **template** parametrized with *types* of variables (parameters and/or returned values) and, using this template, the compiler will produce and compile as many different versions of the function as it is needed — it will do so even for types that do not exist when the template itself is defined!

Let us consider an example which shows the syntax of function templates:

P86: tmplt.cpp Function template

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  template <class T1, typename T2>                          ❶
6.  int howmany(const T1* arr, T2 mn, T2 mx, int size) {      ❷
7.      int count = 0;
8.      for (int i = 0; i < size; ++i)
9.          if (arr[i] > mn && arr[i] < mx) ++count;
10.
11.     // test
12.     cout << "T1=" << typeid(T1).name() << " "             ❸
13.          << "T2=" << typeid(T2).name() << " ";
14.
15.     return count;
16. }
17.
18. int main() {
19.     double mnd = 0, mxd = 10;
20.     int    mni = 0, mxi = 10;
21.     double tabd[] = {-2, -1, 2, 5, 7, 11};
22.     int    tabi[] = {-2, -1, 2, 5, 7, 11};
23.
24.     int ii = howmany(tabi,mni,mxi,6);                     ❹
25.     cout << "res=" << ii << endl;
26.
27.     int id = howmany(tabi,mnd,mxd,6);                     ❺
28.     cout << "res=" << id << endl;
29.
30.     int di = howmany(tabd,mni,mxi,6);                     ❻
```

```
31.     cout << "res=" << di << endl;
32.
33.     int dd = howmany(tabd,mnd,mxd,6);                ❼
34.     cout << "res=" << dd << endl << endl;
35.
36.     int xx = howmany<double,double>(tabd,mni,mxi,6);  ❽
37.     cout << "res=" << xx << endl;
38. }
```

The keyword **template** (❶) informs the compiler that what follows will be the definition of a template of a function (or a class) and not a definition of a concrete function. After that, in angle brackets, there must appear a comma separated list of formal parameters, each preceded with keyword **class** or **typename** – in this context these two keywords are synonyms, although the latter is preferred. Formal parameters of templates can have arbitrary names, often capital letter T is used – in our case we used **T1** and **T2**. Now the definition od the template follows. We use names **T1** and **T2** where names of *types* are expected. We can also use names of derived types like **T1&** or **T2***.

Looking at the code, we can see that the template defines something like a function **howmany** (❷), which takes a pointer to an array of elements of type **T1** and returns the number of elements of the array which are larger than ***mn*** and smaller than ***mx***. Those variables in turn are of type **T2**. [The statement ❸ is not necessary here; it has been added in order to print, at run time, the names (or at least some kind of codes) of types associated with **T1** and **T2**. More about **typeid** operator in chapter on RTTI .] Of course, the compiler cannot compile the template as if it were a function, because there are no types named **T1** or **T2**. However, it will remember that there is a function template named **howmany**.

How can we use the template just defined? On line ❹ we try to call a function **howmany** passing, as arguments, an array of **int** s and values of variables of type **int** as ***mn*** and ***mx***. The compiler will notice that there is no such function, but there is a template named **howmany**. It will then try to find types which, when substituted for **T1** and **T2** in the template, will give the correct signature of the function **howmany**. In our case, replacing **T1** with **int** and **T2** also with **int**, we will get the signature of **howmany** exactly matching the invocation from line ❹. The replacement will then be done, and the resulting function compiled (and then executed at run time). This is what we call **concretization** of a template. Thus the function obtained as the result of concretization will have the form

```
int howmany(int* arr, int mn, int mx, int size) {
    //
    // ...
    //
}
```

Somewhere later in the program, there may be another invocation of **howmany** with arguments of exactly the same types. No concretization will then be needed, because by then the appropriate function will already exist.

Let us now look at ❺. This invocation of **howmany** does *not* exactly match the already existing version of the function, because the second and third arguments are now of type **double**. Therefore, the process of concretization will now take place again. Now the exact match can be obtained by substitutions **T1** → **int** and **T2** → **double**. Hence, another overloaded version of **howmany** will now be generated, this time of the form

```
int howmany(int* arr, double mn, double mx, int size) {
    //
    // ...
    //
```

```
}
```

Invocations ❻ and ❼ are similar: in the first case exact match will correspond to replacements **T1** → **double** and **T2** → **int**, while in the second to **T1** → **double** oraz **T2** → **double**.

Note also the syntax used on line ❽. Here we explicitly specified (in angle brackets after the name of the function) that **double** should be substituted for the first formal parameter of the template (**T1**) and also **double** for the second (**T1**). In this case a version with such signature already exists, and it will be used here, despite the fact that the second and third arguments of the invoked function are now of type **int** (nothing wrong will happen, because conversion **int** → **double** is a standard one and can be performed without loosing information).

The result of the program

```
T1=i T2=i res=3
T1=i T2=d res=3
T1=d T2=i res=3
T1=d T2=d res=3

T1=d T2=d res=3
```

shows that indeed four overloaded versions of **howmany** have been generated. We do not have to worry about the fact that type **int** has been named just **i**, and **double** is **d** — these are just internal names (codes) of types used by the compiler (and can be different for another compiler).

The keyword **class** in the list of formal parameters of the template could suggest that **T1** can only correspond to a type defined by a class. This is, however, not true – equally well this can be a built-in type (as it was in our example). Therefore, the synonym **typename** might be less confusing in this context.

Let us consider another example:

P87: tmpl.cpp Function templates

```
1.  #include <iostream>
2.  #include <typeinfo>
3.  using namespace std;
4.
5.  template <typename T>                          ❶
6.  T larger(T k1, T k2) {
7.      cout << "T=" << typeid(T).name() << " ";   ❷
8.      return k1 < k2 ? k2 : k1;
9.  }
10.
11. double larger(double k1, double k2) {          ❸
12.     cout << "Spec. double ";
13.     return k1 < k2 ? k2 : k1;
14. }
15.
16. template<>                                     ❹
17. short larger<short>(short k1, short k2) {
18.     cout << "Spec. short ";
19.     return k1 < k2 ? k2 : k1;
20. }
21.
22. template<>
```

```
23. long larger<long>(long k1, long k2) = delete;  ❺
24.
25. int main() {
26.     short s1 = 4, s2 = 5;
27.
28.     cout << larger(1.5,2.5)     << endl;        ❻
29.     cout << larger(111,222)     << endl;        ❼
30.     cout << larger('a','d')     << endl;        ❽
31.     cout << larger<int>(s1,s2) << endl;        ❾
32.     cout << larger(30L,50L)     << endl;        ❿
33. }
```

We define (❶) a function template **larger**. The template depends on one type parameter **T**. Functions generated by the template will take two arguments of the same type, and return by value a result of type **T** — it will be the value of the larger of the two arguments. As before, we added a line (❷) printing information on the actual type associated with **T** (using the **typeid** operator from the header `typeinfo` — see chapter on RTTI).

Note that there is also a function (*not* a template) with the same name **larger** (❸). Its signature matches exactly the template **larger** with the substitution **T** → **double**.

In a similar way we provide a specific version of **larger** for the type **short** (❹). This time we explicitly indicated that we are providing specific concretization of a template for a given type — note empty angle brackets after the keyword **template** and name of a concrete type (**short** in this case) after the function's name. This form is preferred, because the compiler will be able to check whether what we are doing actually is a syntactically correct concretization of the template **larger**.

Finally, line ❺ declares the concretization of our template for type **long** as nonexistent(**=delete** in place of the body — this is a feature added by the C++11 standard).

The program does not compile:

```
tmpl.cpp: In function 'int main()':
tmpl.cpp:32:27: error: use of deleted function
        'T larger(T, T) [with T = long int]'
```

The reason is, of course, the last line (❿). Deleted functions *are* taken into consideration when looking for the best candidate; here it will be concretization of the template with **T** → **long** which gives a perfect match. It is *after* this candidate has been found and selected, when the compiler will "realize" that this function is deleted — no other candidate will then be looked for and the compilation will fail.

After commenting out the last line, compilation succedes and running the program gives:

```
Spec. double 2.5
T=i 222
T=c d
T=i 5
```

Let us have a closer look at some general aspects of function templates illustrated by the program:

- On line ❻ **larger** is invoked with two arguments of type **double**. This would exactly correspond to a function generated from the template with **T** → **double**. However, such a function has been explicitly defined in the program (❸) and, as we can see from the output, if an explicit form exists, it will be chosen by the compiler.
- Then, on line ❼, **larger** is called with two **int** s: this *exactly* correspond to function

generated from the template after substitution **T** → **int**. It also corresponds to the signature of the explicitly defined version with parameters of type **double**, after applying the standard conversion **int** → **double** to the arguments. However, as we can see from the output, the compiler preferred in this case to produce a new version of **larger** which *exactly* matches the type of arguments, while the existing version with **double** s would require conversions of arguments.

- We then call (❽) **larger** with two arguments of type **char**. Again, this invocation could have been "served" by already existing version with **int** s, but the compiler will not use it because it can produce from the template a version which gives a perfect match.
- On line ❾ we explicitly request concretization with **T** → **int** (which in this case has already been created earlier). Although a better match could be produced with concretization **T** → **short**, the compiler will not do it, because we specified explicitly which concretization we want to use.

Names of types appearing in the printout (**i**, **c**) are internal names of the types **int** and **char** — the names themselves are not so important, what is important is the fact that different types have different names, so one can compare types of objects.

In the body of the template we compare ***k1*** and ***k2*** using the '<' operator. This does not pose any problem for numeric types of these variables. However, one might be in trouble if **T** corresponded to a user defined type which does not support comparison — using such type for concretization would provoke compilation error (although, as we will see, it *is* possible to define comparisons with '<' operator even for our own types).

Let us consider a more realistic example. The following program defines a function (more precisely, a template) **minmaxmed**, which takes an array of object and calculates minimum and maximum elements of the array, together with the median (such a value that one half of elements is larger than this value and the other half smaller). Again, the type of elements must be such that they can be compared. The minimum and maximum elements are returned through reference arguments and the median as the value returned by the function. We also define additional function templates: **printarr** (for printing elements of an array), **inssort** (for sorting an array using the insertion sort algorithm), and finally **test**, which invokes the other functions:

P88: sorttmplt.cpp Function templates

```
1.  #include <iostream>
2.  #include <cstring>    // memcpy
3.  using namespace std;
4.
5.  template<typename T>
6.  void printarr(ostream&,const T[],int);
7.
8.  template<typename T>
9.  void inssort(T[],int);
10.
11. template<typename T>
12. double minmaxmed(const T[],int,T&,T&);
13.
14. template<typename T>
15. void test(T[],int);
16.
17. int main() {
18.     cout << "\n===array int===" << endl;
19.     int arri[] = {9,7,2,6,6,2,7,9,2,9,5,2};
```

```
20.     test(arri,sizeof(arri)/sizeof(int));
21.
22.     cout << "\n===array double===" << endl;
23.     double arrd[] = {9.5,2.5,6,7.5,9,2,5,2.5};
24.     test(arrd,sizeof(arrd)/sizeof(double));
25.
26.     cout << "\n===array unsigned===" << endl;
27.     unsigned arru[] = {23,32,12,76,21,45,20,67};
28.     test(arru,sizeof(arru)/sizeof(unsigned));
29. }
30.
31. template<typename T>
32. void test(T arr[],int size) {
33.     T min, max;
34.
35.     double median = minmaxmed(arr,size,min,max);
36.
37.     cout << "min = " << min << ", max = " << max
38.          << ", median = "   <<  median   << endl;
39.
40.     cout << "Original array: ";
41.     printarr(cout, arr, size);
42.
43.     inssort(arr, size);
44.
45.     cout << "Sorted   array: ";
46.     printarr(cout, arr, size);
47. }
48.
49. template<typename T>
50. void printarr(ostream& str, const T t[], int size) {
51.     str << "[ ";
52.     for (int i = 0; i < size; ++i) str << t[i] << " ";
53.     str << "]" << endl;
54. }
55.
56. template<typename T>
57. void inssort(T a[], int size) {
58.     int i, indmin = 0;            // sentinel
59.     for (i = 1; i < size; ++i)
60.         if (a[i] < a[indmin]) indmin = i;
61.     if (indmin != 0) {
62.         T p = a[0];
63.         a[0] = a[indmin];
64.         a[indmin] = p;
65.     }
66.
67.     for (i = 2; i < size; ++i) {  // sorting
68.         int j = i;
69.         T v = a[i];
70.         while (v < a[j-1]) {
71.             a[j] = a[j-1];
72.             j--;
73.         }
74.         if (i != j ) a[j] = v;
75.     }
76. }
77.
78. template<typename T>
79. double minmaxmed(const T t[], int size, T& min, T& max) {
80.     T* arr = new T[size];
81.     memcpy(arr,t,size*sizeof(T));
82.
83.     inssort(arr, size);
84.
85.     min           = arr[0];
86.     max           = arr[size-1];
87.     double median = size%2 == 0 ?
88.                        0.5*(arr[size/2] + arr[size/2-1])
89.                      : arr[size/2];
90.
91.     delete [] arr;
92.     return median;
93. }
```

In order to find the minimum, maximum and the median, the function **minmaxmed** creates a copy of the array passed through argument (so the original will not be modified), sorts it, so it can easily find the answer, and finally removes the work copy of the array. As we can see from the printout,

```
===array int===
min = 2, max = 9, median = 6
Original array: [ 9 7 2 6 6 2 7 9 2 9 5 2 ]
Sorted   array: [ 2 2 2 2 5 6 6 7 7 9 9 9 ]

===array double===
min = 2, max = 9.5, median = 5.5
Original array: [ 9.5 2.5 6 7.5 9 2 5 2.5 ]
Sorted   array: [ 2 2.5 2.5 5 6 7.5 9 9.5 ]

===array unsigned===
min = 12, max = 76, median = 27.5
Original array: [ 23 32 12 76 21 45 20 67 ]
Sorted   array: [ 12 20 21 23 32 45 67 76 ]
```

we were able to get an answer for arrays of various types. Note, however, that we have used (although we did not have to) the function **memcpy** for copying arrays — this will work only for types for which “bit to bit” copying gives correct results. Notice also that we can invoke functions generated from templates in other such functions: for example, template function **test** calls **minmaxmed**, **inssort** and **printarr**, which are also defined in terms of templates.

In this program we first declared templates at the beginning and the definitions come at the end: in both cases, of course, we have to use the keyword **template**.

Let us illustrate one more syntactic “trick” that can be useful when defining templates of functions. Starting from C++14 version of the standard, we can make the compiler fully responsible for determining the return type of a function. We just declare the return type as **decltype(auto)** — compiler will then infer the correct type by analyzing the body of the function and looking at all **return** statements. Of course, in all of them, a value of exactly the same type must be returned. Let us see a very simple example

P89: autoret.cpp Inferring the return type

```
1. #include <iostream>
2. #include <typeinfo>
3. using namespace std;
4.
5.   // return type may be deduced when T and U are known
6. template<typename T, typename U>
7. decltype(auto) mul(T x, U y) {
8.     return x*y;
9. }
10.
11. int main() {
12.     auto r1 = mul(2.0,7); // double*int -> double
13.     std::cout << r1 << " :: " << typeid(r1).name() << "\n";
14.     auto r2 = mul(2,7L);  //   int*long -> long
15.     std::cout << r2 << " :: " << typeid(r2).name() << "\n";
16. }
```

From the output

```
14 :: d
14 :: l
```

we can see that

- in the first invocation, types of arguments are **T** → **double** and **U** → **int**; deduced type of the returned value is **double**;
- in the second invocation, types of arguments are **T** → **int** and **U** → **long**; deduced type of the returned value is **long**.

Prev: 11.13 Lambda functions **Up:** 11. Functions **Next:** 12. Dynamic memory management

12. Dynamic memory management

In C/C++ the programmer can control the memory, its allocation and deallocation, himself. This is a strong point of these languages, as the possiblity of dynamic memory management allows the programmer to write better, more effective code. On the other hand, there is a weak point: this issue can be fairly difficult; unfortunately, errors due to improper memory management belong to the most common and hardest to detect. Undoubtedly, memory management makes the program more difficult to write and to maintain — it also requires more skills from the programmer.

In many languages (Java, Lisp, Smalltalk, Python), memory management is taken care of by special modules, called “garbage collectors”. Working in background, this modules reclaim segments of memory on the heap occupied by objects which are not accessible from the program and as such they cannot serve any useful purpose any more. While very convenient, this mechanism limits programmer's control over his/her program.

Subsections

- 12.1 Introduction
- 12.2 Memory allocation — *new* operator
- 12.3 Deallocation of memory — operator *delete*
- 12.4 Dynamic multidimensional arrays
- 12.5 Memory management in C
- 12.6 Functions operating on memory
- 12.7 Placement *new* operator

12.1 Introduction

Data cretaed by the running program can be stored in several parts of the memory allocated by the operating system to the process. Generally, this issue is very implementation dependent, but in almost all architectures two main parts of memory for user's data are **stack** and **heap** (also called **free memory**).

Local variables, defined in functions (but not allocated by the **new** operator) are located on the stack. This part of the memory changes dynamically: on each entry to a function, new local variables belonging to this function's namespace are created, to be released immediately when the function exits (we say thet the stack is then **rewound**). In fact, this rule applies to any *block*, not necessarily a function. Generally, the programmer cannot (or at least, should not) manipulate with the stack “by hand”.

The second part of the memory is called the **heap**. This is the part where the programmer can freely allocate space for his/her variables (objects), then read or modify them. They will exist until the end of the program's execution unless they are released explicitly, “by hand”, by the program (what is possible and necessary in C/C++; in Java it's virtually impossible). This causes a real danger of exhausting the memory before normal termination of the program — in such cases the program usually crashes in a more or less civilized way.

Prev: 12. Dynamic memory management **Up:** 12. Dynamic memory management **Next:** 12.2 Memory allocation — new operator

12.2 Memory allocation — *new* operator

The programmer can allocate new segment of memory for his/her data using **new** operator (in C there is a **malloc** function performing a similar task — we will tell more about later).

The **new** operator requires that the type of data for which memory is to be allocated be specified. It is also possible to allocate memory for arrays; in this case not only the type of elements but also their number has to be specified (what is logical — the program has to know the size of memory requested).

In its simplest form, the **new** operator may be used to allocate memory for a single piece of data of a specified type. The syntax can be:

```
int* pi = new int;
```

The **new** operator looks for a segment of free memory sufficiently large to hold an **int**, returns the *address* of this segment and marks this piece of memory as being occupied. It also initializes the newly created variable (in the case of an **int** initialization in fact does nothing, but for other types it is a nontrivial operation involving invocation of a constructor). The type of the returned value is **int*** (not **int**!), so it can be assigned to a variable of this type. Note that there are *two* variables involved here: one *local* of pointer type (named ***pi*** in the above example) and one anonymous variable of type **int** on the heap. The variable on the heap does not have any name: the only way we can refer to it is by a pointer pointing to this variable. In particular, if we loose the pointer, e.g., when it is removed, being a local variable, when exiting the block it was defined in, we will not be able to refer to the variable on the heap any more. But the variable on the heap itself will *not* be removed — the segment of memory it is located in will remain marked as occupied until the end of the program's execution! This segment of memory will not be accessible for allocating other variables while being completely useless. This is what we call **memory leakage**.

Removing a pointer does *not* free the memory pointed to by this pointer!

Of course, instead of **int**, any other type could have been used in the above example, also a user-defined type (i.e., a class). If it were a class named **AClass** (with a public default constuctor), we could write

```
AClass* pk = new AClass;
```

and if it would have a non-default constructor requiring arguments, e.g. two integers,

```
AClass* pk = new AClass(12,8);
```

The same syntax can be used to allocate a variable of a primitive type, e.g.,

```
int* pi = new int(18);
```

will allocate an **int** on the heap and initialize it with value 18, as if **int** were the name of

a class with a one-argument constructor. This feature of the language is intentional: its creator, Bjarne Stroustrup, wanted to make the syntax used when dealing with built-in and user-defined types identical, so they can be treated on the same footing. In the same way we can create on the heap a *constant*:

```
const int* a_const = new const int(1);
```

This would not have been possible without an initializer in parentheses, since constants must be initialized while they are being created; after that it would be too late.

Variables allocated on the heap are anonymous but, if it is convenient, can be given an alias using references. In the following program, the variable ***rd*** will be a reference to an anonymous variable on the heap pointed to by pointer ***pd***:

P90: ref.cpp Reference to a variable on the heap

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     double *pd = new double(4.5),
6.            &rd = *pd;
7.
8.     cout << "*pd = " << *pd << endl;
9.     cout << " rd = " <<  rd << endl;
10.    *pd = 1.5;
11.    cout << "*pd = " << *pd << endl;
12.    cout << " rd = " <<  rd << endl;
13.    delete pd;
14. }
15.
```

Variable ***pd*** is a pointer: it points to a newly crated anonymous variable on the heap. Then we define a reference ***rd*** which becomes an alias denoting this variable. From now on we can reference the variable by both `*pd` or just ***rd***: both names refer to the same physical variable:

```
*pd = 4.5
 rd = 4.5
*pd = 1.5
 rd = 1.5
```

One can allocate memory for more variables of a given type, i.e., for an array. The syntax allowing to perform this task is as follows:

```
int* pi = new int[dim];
```

After specifying a type (**int** in our case), one has to specify, in square brackets, the number of elements, i.e., the size of the array which is to be allocated; formally, the type of this expression should be **size_t**, which is a **typedef** alias of an unisigned integer type (usually **unsigned long**). What is allocated is a “legal” array — all elements will be allocated in a contiguous segment of memory. Such form of the **new** operator returns the address of the first element of the array. Therefore ***pi*** can be treated as the name of an array: normal pointer arithmetic applies and one can use the notation `pi[n]` to access the $(n + 1)$-th element of the array. What is important, the size (number of elements), denoted by ***dim*** above, does not have to be known at compile time: it can be any expression which has a positive integer value (input from keyboard, read from a file, calculated by the program,

etc.). That was not the case for static arrays, where the size must have been a constant. For that reason, arrays allocated with **new** operator are called **dynamic arrays**. The whole process of allocating memory in this way is called **dynamic memory allocation**.

If, for example, the current value of ***dim*** is 40, then 160 bytes of memory will be allocated (4×40) and the address of this memory segment will be returned as a value of type **int***, which we assigned to the variable ***pi***.
Memory allocated in this way *is* initialized, although for primitive types this amounts to "do nothing" (situation is different for class types — then, elements of the array are created using the default constructor). As we noted, ***pi*** can be used as the name of an array, so we could assign sensible values to elements of our array like this:

```
for (int i = 0; i < dim; ++i) pi[i] = 2*i;
```

As we have seen, the name of a type in **new** expression may be followed by round or square parentheses. These have very different meaning and it is important not to mix up these two forms. Round parentheses are used when allocating *one* object of a given type; what goes into parentheses is an initializer (or arguments for a constructor in case of class types). Square parentheses (brackets) denote the size of the array to be allocated (number of its elements).

Note also, that it is possible to initialize dynamic arrays (as it was possible for static arrays) using the "brace-list" initializers:

```
int* p = new int[5]{1, 2, 3, 4, 5};
```

Multidimensional arrays can also be allocated dynamically by **new** operator. However, only *the first* dimension may be dynamical, all the rest must be static, i.e., known at compile time — exactly as for static arrays. Therefore, one could call such arrays "semidynamic". Let us look at an example:

P91: semidyn.cpp "Semidynamic" multidimensional arrays

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      const int DIM = 3;
6.      cout << "Enter the first dimension: ";
7.      int size;
8.      cin >> size;
9.      int (*t)[DIM] = new int[size][DIM];        ❶
10.
11.     for (int i = 0; i < size; ++i)
12.         for (int j = 0; j < DIM; ++j)
13.             t[i][j] = 10*i + j;
14.
15.     int* p = reinterpret_cast<int*>(t);        ❷
16.
17.     for (int i = 0; i < DIM*size; ++i)
18.         cout << p[i] << " ";
19.     cout << endl;
20.
21.     cout << "t[0]         : " << t[0] << endl;  ❸
22.     cout << "t[1]         : " << t[1] << endl;
23.     cout << "sizeof(t[0]): " << sizeof(t[0]) << endl; ❹
24. }
```

We allocate memory for a matrix of dimensions ***size*** × ***DIM*** (line ❶), where ***size*** is not known in advance — it will be read from the keyboard at runtime. However, the second dimension (and the same would apply if there were more dimensions) is specified as a constant (in our case it is ***DIM***=3). Note, the the type of ***t*** is *pointer to three-element array of **int** s*, i.e., what is pointed to is *not* an **int** but a three-element array of **int** s. Consequently, ***t[0]*** is such an array and its size is 12 bytes (3×4). It is the first row of the matrix. Similarily, ***t[1]*** is also such an array — it corresponds to the second row of the matrix and should start in memory 12 bytes away from the beginning of the first row. Note that printing ***t[0]*** we get an address — ***t[0]*** is of array type, so will be converted to the pointer pointing to the beginning of the segment of memory where the array starts. Therefore, printing ***t[0]*** and ***t[1]*** (❸) we get addresses and we can convince ourselves that everything is as expected (0x15da01c-0x15da010=0xC in the hexadecimal, or 12 in the decimal notation; these particular addresses could have been different, but the difference should be like the one we got):

```
Enter the first dimension: 5
0 1 2 10 11 12 20 21 22 30 31 32 40 41 42
t[0]        : 0x15da010
t[1]        : 0x15da01c
sizeof(t[0]): 12
```

On line ❷ we create a pointer variable of type **int*** and we initialize it with the address of the matrix ***t***; we will explain later why the **reinterpret_cast** operator was needed — note that types of ***t*** and ***p*** are different (traditional for of type casting – '`(int*)`' – could also have been used here). Treating ***p*** as a one-dimensional array of **int** s, we can print its values — as we can see these are exactly the values assigned in the loop to the elements of the matrix. They are indeed located in memory row-by-row (this information can be extremely significant for the efficiency of operations on matrices; in Fortran they would have been located column-by-column).

Memory allocation can fail (e.g., if there is no more memory available). In such cases, the ***NULL*** value is returned in traditional C, while in C++ an exception is generated (of type **bad_alloc** from the header **new**). One can catch this exception and handle it somehow so the program does not crash. We will tell more about exceptions in a separate chapter , but the following example should be quite clear for those who already know Java or Python:

P92: alloc.cpp Memory allocation failure

```
1.  #include <iostream>
2.  #include <new>
3.  #include <iomanip>
4.  using namespace std;
5.
6.  int main() {
7.      const size_t mega = 1024*1024, step = 200*mega;
8.
9.      for (size_t size = step; ;size += step) {
10.         try {
11.             char* buf = new char[size];
12.             delete [] buf;
13.         }
14.         catch(bad_alloc) {
15.             cout << "FAILED: " << setw(4)
16.                  << size/mega  << " MB" << endl;
17.             return 1;
18.         }
19.         cout << "    OK: " << setw(4)
20.              << size/mega  << " MB" << endl;
21.     }
```

```
}
```

In an infinite loop, we allocate and immediately release larger and larger segment of memory (we will tell more about **delete** operator in the next section). At some moment, there is no available memory of the size requested, so an exception is thrown. We handle it in **catch** block: we print a message and terminate the program by calling **return** in the **main** function. The program generated the following output:

```
    OK:  200 MB
    OK:  400 MB
    OK:  600 MB
    OK:  800 MB
    OK: 1000 MB
    OK: 1200 MB
    OK: 1400 MB
    OK: 1600 MB
    OK: 1800 MB
    OK: 2000 MB
    OK: 2200 MB
    OK: 2400 MB
    OK: 2600 MB
    OK: 2800 MB
FAILED: 3000 MB
```

That does not mean that the computer actually had 3GB of memory: the virtual memory (swap file) was added and occupied memory subtracted.

Allocating memory for a single object and for an array is implemented in a different way. Therefore, allocating a single **int**

```
int* pi = new int;
```

is *not* equivalent to allocating a one-element array

```
int* pi = new int[1];
```

The difference is very important when it comes to releasing the allocated memory (see the next section).

Prev: 12.1 Introduction **Up:** 12. Dynamic memory management **Next:** 12.3 Deallocation of memory — operator

12.3 Deallocation of memory — operator *delete*

Memory allocated on the heap with **new** operator must be deallocated "manually", otherwise it will be marked as occupied until the program's termination, even if data which is stored there is not needed any more (or simply inaccessible). Memory can be deallocated (reclaimed) with the help of **delete** operator. Suppose we have allocated a single (non-array) object using **new** — the operator returned an address, which we assigned to a pointer variable ***pi*** of the appropriate pointer type. Exactly this address (more generally, an expression whose value is this address) must be used as argument of **delete** operator; for example:

```
delete pi;
```

will deallocate the memory allocated by the **new**. Very important: the variable ***pi*** will remain intact; it will *not* be "removed" or modified in any other way — only the memory it points to will be "removed" (released). Obviously, in place of ***pi***, one could have used any expression as long as its value is the same as the value of ***pi***.

It is illegal to use delete for an address which has not been returned by new.

Immediately after deallocation, the reclaimed segment of memory becomes available and may be used by the system for any purpose. Therefore,

invoking delete twice for the same address is illegal; such errors are not detected by the compiler and lead to unpredictable behavior of the program at runtime.

The syntax for deallocating memory for arrays (allocated with the **new** operator with a size given in brackets) is a little different. The keyword **delete** is then followed by a pair of *empty* brackets.

```
delete [] pi;
```

Now ***pi*** must be an expression whose value is the address returned by the corresponding **new** operator which allocated an *array*, and not a single object. Note that one does not specify any size in brackets — this size was specified by **new** expression and is remembered.

If memory was allocated for a single object, it must be released with delete; if this was an array (even a one-element array!), it must be released with delete[].

When applied to **nullptr**, or simply zero, both forms of **delete** — with and without brackets — do nothing: such operation is completely harmless. Some programmers assign the value

nullptr to pointers which point to a memory which has just been released — in this way they avoid using this pointer (now invalid) with another **delete**:

```
1.      int* pi = new int[40];
2.      // ...
3.      delete [] pi;
4.      pi = 0;
5.      //
6.      delete [] pi; // harmless now
```

Let us consider an example. We want a function, named **minmaxmed** below, which, given an array, finds its minimum and maximum elements as well as its median (such a value that half of the elements of the array are smaller and half are larger than this value). There are faster methods to do this job, but the simplest way is to order (sort) the array first — then the first and the last elements will be minimum and maximum, respectively, and the value of the middle element will be the median (or the mean value of two middle elements, if the array has even number of elements).

There is a problem with this method however: the input array would be modified, and this can be undesirable. Therefore, we will create a new array of the same size and type as the input array, copy elements from the original array to the newly created one, perform out task using the copy and remove it afterwards. This is exactly what the function **minmaxmed** does:

P93: median.cpp Dynamic allocation of arrays

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void   printarr(ostream&,const int[],size_t);
5.  void   inssort(int[],size_t);
6.  double minmaxmed(const int[],size_t,int&,int&);
7.
8.  int main() {
9.      int arr[] = {7,2,6,4,7,5}, min, max;                ❶
10.     size_t size = sizeof(arr)/sizeof(arr[0]);
11.
12.     double median = minmaxmed(arr,size,min,max);
13.
14.     cout << "min = " << min << ", max = " << max
15.          << ", median = "   <<  median    << endl;
16.
17.     cout << "Original array: ";
18.     printarr(cout, arr, size);                          ❷
19.
20.     inssort(arr, size);                                 ❸
21.
22.     cout << "Sorted   array: ";
23.     printarr(cout, arr, size);                          ❹
24. }
25.
26. void printarr(ostream& str, const int t[], size_t size) {
27.     str << "[ ";
28.     for (size_t i = 0; i < size; ++i) str << t[i] << " ";
29.     str << "]" << endl;
30. }
31.
32. void inssort(int a[], size_t siz) {
33.     size_t indmin = 0;
34.     for (size_t i = 1; i < siz; ++i)
35.         if (a[i] < a[indmin]) indmin = i;
36.     if (indmin != 0) {
37.         int p = a[0];
38.         a[0] = a[indmin];
39.         a[indmin] = p;
```

```
40.     }
41.     for (size_t i = 2; i < siz; ++i) {
42.         size_t j = i;
43.         int v = a[i];
44.         while (v < a[j-1]) { a[j] = a[j-1]; j--; }
45.         if (i != j ) a[j] = v;
46.     }
47. }
48.
49. double minmaxmed(const int t[], size_t size,
50.                  int& min, int& max) {
51.     int* arr = new int[size];                          ❺
52.
53.     // would be better to use memcpy...
54.     for (size_t i = 0; i < size; ++i) arr[i] = t[i];   ❻
55.
56.     inssort(arr, size);                                ❼
57.
58.     min = arr[0];
59.     max = arr[size-1];
60.
61.     double median = size%2 == 0 ?
62.                 0.5*(arr[size/2] + arr[size/2-1])
63.               : arr[size/2];
64.     delete [] arr;                                     ❽
65.     return median;
66. }
```

The function **minmaxmed** gets an array (pointer) ***t*** — the corresponding parameter was declared with modifier **const** to avoid accidental modifications of the original array. Then, on line ❺, we allocate a new array of the same size as ***t*** and we copy to it all elements of the ariginal array (❻); this copy is now sorted (❼). After that it is easy to find the results; we put them into ***min*** and ***max*** which were passed by reference, so the results will be accessible in the calling function (**main** in our case). The value of the median is then returned by value.

Before exiting the function we have to release the working copy of the original array (❽). The pointer variable ***arr*** is local in the function; when the function exits, it will be destroyed and there will be no way to access the array which it points to — in particular it will be impossible to reclaim the memory occupied by the array. Such an array would be created each time the function **minmaxmed** is called and memory leakage would cumulate.

The sorting function **inssort** that we used (❸ and ❼) implements the so called *insertion sort* (with sentinel). It is generally not the fastest way of sorting an array, but can be extremely efficient for arrays which are already almost sorted — the case we encounter surprisingly often in practice.

In the main program, we create an array (❶) and we pass it to the function **minmaxmed**. We print the results and also the orignal array (❷) to see that it has not been modified; Then we sort the original table (❸) and print it (❹) to see better that the results are correct:

```
min = 2, max = 7, median = 5.5
Original array: [ 7 2 6 4 7 5 ]
Sorted   array: [ 2 4 5 6 7 7 ]
```

Note that the printing function **printarr** can print the contents of an array to any stream of type **ostream** passed to it by reference. We pass the object ***cout***, as we do not know any other streams yet, but equally well we could have used a stream associated with a disk file (see the chapter on input/output operations).

T.R. Werner, September 27, 2023; 22:32

12.4 Dynamic multidimensional arrays

We now know how to create dynamically a one-dimensional array, or an array with more dimensions, but only one, the first, fully dynamical (i.e., such that its size can be determined at runtime). Quite frequently, however, we need truly multidimensional dynamic arrays with all its dimensions not known in advance. We will show one of the possible methods of allocating and using such arrays.

Suppose we want a two-dimensional array (matrix) of **double** s with sizes 2×3. The syntax should allow for referring the elements of the matrix in the usual way; e.g., ***m[1][2]*** should stand for the element in the second row and the third column (as counting is zero based) of the array (matrix) ***m***.

What does ***m[1][2]*** mean? We know that this is just an abbreviation for '`*(m[1] + 2)`' (see the section on pointer arithmetic). Therefore, ***m[1]*** must be a pointer of type **double***. But this means that ***m*** should be an array of pointers of this type — values of the elements of this array should correspond to addresses of beginnings of rows of the matrix. The expression `m[1]` is equivalent to '`*(m + 1)`'; as its value is of type **double***, the ***m*** itself must be a pointer to a pointer and have type **double****.

The situation is illustarted in the following figure:

We have six elements of the matrix in the upper row. These are numbers of type **double** with values indicated inside the boxes — here they are 1.5, 2.5 etc. Names of the variables corresponding to these values are given on the right-hand side of the boxes. The numbers below are *addresses* of the variables — they are eight bytes apart one from the next (as **double** s occupy eight bytes each). Note that the addresses are expressed in the hexadecimal notation, so, e.g., $38_{16} + 8_{16} = 40_{16}$ (what corresponds to $56 + 8 = 64$ in the decimal notation). The upper row represents our matrix in the computer's memory: all elements occupy a contiguous region in memory, row by row.

The middle row in the figure represents a one-dimensional two-element array of *pointers* of type **double***. Elements occupy 4 (or 8 on 64-bit machine) bytes each and their values correspond to addresses of the beginnings of rows of the matrix.

The single variable ***m*** in the bottom row in the figure is a pointer pointing to the first element of the array of pointers of type **double*** (the middle row). Its type is therefore “pointer to

a pointer", i.e., **double****.

Let us see how to implement all this in a program:

P94: matrix2dim.cpp Dynamic two-dimensional array

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  double** allocMatrix2D(size_t,size_t);
5.  void     deleteMatrix2D(double**&);
6.
7.  int main() {
8.
9.      size_t dim1 = 2, dim2 = 3;  // NOT constants!
10.
11.       // allocating
12.     double** matrix2d = allocMatrix2D(dim1,dim2);      ❶
13.
14.       // filling out the matrix
15.     for (size_t i = 0; i < dim1; ++i)                  ❷
16.         for (size_t j = 0; j < dim2; ++j)
17.             matrix2d[i][j] = 3*i+j+1.5;
18.
19.       // printing
20.     for (size_t i = 0; i < dim1; ++i) {
21.         for (size_t j = 0; j < dim2; ++j)
22.             cout << matrix2d[i][j] << " ";
23.         cout << endl;
24.     }
25.
26.       // deallocating
27.     deleteMatrix2D(matrix2d);                          ❸
28. }
29.
30. double** allocMatrix2D(size_t dim1, size_t dim2) {     ❹
31.     double** matrix2d = new double*[dim1];
32.     double*  dumm     = new double[dim1*dim2];
33.
34.     for (size_t i = 0; i < dim1; ++i)
35.         matrix2d[i] = dumm + i*dim2;
36.
37.     return matrix2d;
38. }
39.
40. void deleteMatrix2D(double**& matrix2d) {              ❺
41.     delete [] matrix2d[0];
42.     delete [] matrix2d;
43.     matrix2d = 0;
44. }
```

Matrices (two-dimensional arrays) are created by the function **allocMatrix2D** (❹). Variables ***dim1*** and ***dim2*** are sizes of the matrix: ***dim1*** is the number of rows and ***dim2*** — of columns.

First, we create a pointer ***matrix2d*** of type **double**** and we assign to it the address of the allocated array of pointers of type **double***. Number of elements of this array is equal to the number of rows of the matrix. The variable ***matrix2d*** corresponds to ***m*** in the figure, and the array it points to is represented by the middle row.

Next, we create the matrix itself. It is in fact an array of **double** s. Number of elements is the product of sizes ***dim1*** and ***dim2***. The address which is the value of ***dumm*** is the address of the first element of the allocated matrix (0x30 in the figure).

Now we have to fill out the array of pointers from the middle row of the figure. They have to point to the first elements of rows; each of them has the length (in bytes) equal to the length of one **double** multiplied by the number of elements in one row, i.e., ***dim2*** — number of columns of the matrix.

The function returns the value of the variable ***matrix2d*** which can then be used according to usual matrix syntax.

In the main program we allocate a matrix (❶) and fill it out with some arbitrary values (❷); we then print it to check if it really behaves as a two-dimensional matrix:

```
1.5 2.5 3.5
4.5 5.5 6.5
```

After using the matrix, we have to reclaim the memory it occupies (❸). To this end, we call the function **deleteMatrix2d** (❺). The function invokes **delete** twice, as the operator **new** was used twice to create the arrays. First we deallocate the array of numbers — its address is contained in ***matrix2d[0]***, since it points to the beginning of the first row. Next we deallocate the array of pointers pointed to by ***matrix2d***. The order is important: otherwise, after deallocating the array of pointers, we would loose the address of the array of numbers. To be on the safe side, we also put zero into the variable ***matrix2d*** — it was passed by reference, so this have effect in the calling function as well.

In the similar way we can build dynamical arrays (matrices) of even more dimensions. The program below illustartes this for two-, three- and four-dimensional matrices (this time their elements are of integer type):

P95: matrices.cpp Multidimensional dynamic arrays

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int**   allocMatrix2D(int,int);
5.  void    deleteMatrix2D(int**&);
6.
7.  int***  allocMatrix3D(int,int,int);
8.  void    deleteMatrix3D(int***&);
9.
10. int**** allocMatrix4D(int,int,int,int);
11. void    deleteMatrix4D(int****&);
12.
13. int main() {
14.    int dim1 = 7, dim2 = 9, dim3 = 12, dim4 = 5;
15.
16.     // 2-dimensional matrices ///////////////////////
17.
18.     // allocating
19.     int** matrix2d = allocMatrix2D(dim1,dim2);
20.
21.     // test
22.     for ( int i = 0; i < dim1; i++ )
23.         for ( int j = 0; j < dim2; j++ )
24.             matrix2d[i][j] = i+j+2;
25.
26.     cout << "2-dimensional matrix" << endl
27.          << " Middle element: "
28.          << matrix2d[dim1/2][dim2/2]
29.          << endl << "     should be : "
30.          << dim1/2 + dim2/2 +2 << endl;
31.     cout << " Last element  : "
32.          << matrix2d[dim1-1][dim2-1]
33.          << endl << "     should be : "
34.          << dim1 + dim2 << endl << endl;
```

```
35.
36.     // deleting
37.     deleteMatrix2D(matrix2d);
38.
39.     // 3-dimensional matrices ///////////////////////
40.
41.     // allocating
42.     int*** matrix3d = allocMatrix3D(dim1,dim2,dim3);
43.
44.     // test
45.     for ( int i = 0; i < dim1; i++ )
46.         for ( int j = 0; j < dim2; j++ )
47.             for ( int k = 0; k < dim3; k++ )
48.                 matrix3d[i][j][k] = i+j+k+3;
49.
50.     cout << "3-dimensional matrix" << endl
51.          << " Middle element: "
52.          << matrix3d[dim1/2][dim2/2][dim3/2]
53.          << endl << "     should be : "
54.          << dim1/2 + dim2/2 + dim3/2 + 3 << endl;
55.     cout << " Last element  : "
56.          << matrix3d[dim1-1][dim2-1][dim3-1]
57.          << endl << "     should be : "
58.          << dim1 + dim2 + dim3 << endl << endl;
59.
60.     // deleting
61.     deleteMatrix3D(matrix3d);
62.
63.     // 4-dimensional matrices ///////////////////////
64.
65.     // allocating
66.     int**** matrix4d = allocMatrix4D(dim1,dim2,dim3,dim4);
67.
68.     // test
69.     for ( int i = 0; i < dim1; i++ )
70.         for ( int j = 0; j < dim2; j++ )
71.             for ( int k = 0; k < dim3; k++ )
72.                 for ( int m = 0; m < dim4; m++ )
73.                     matrix4d[i][j][k][m] = i+j+k+m+4;
74.
75.     cout << "4-dimensional matrix" << endl
76.          << " Middle element: "
77.          << matrix4d[dim1/2][dim2/2][dim3/2][dim4/2]
78.          << endl << "     should be : "
79.          << dim1/2 + dim2/2 + dim3/2 + dim4/2 + 4 << endl;
80.     cout << " Last element  : "
81.          << matrix4d[dim1-1][dim2-1][dim3-1][dim4-1]
82.          << endl << "     should be : "
83.          << dim1 + dim2 + dim3 + dim4 << endl << endl;
84.
85.     // deleting
86.     deleteMatrix4D(matrix4d);
87. }
88.
89. int** allocMatrix2D(int dim1, int dim2) {
90.     int** matrix2d = new int*[dim1];
91.     int*  dumm     = new int[dim1*dim2];
92.     for ( int i = 0; i < dim1; i++ )
93.         matrix2d[i] = dumm + i*dim2;
94.     return matrix2d;
95. }
96.
97. void deleteMatrix2D(int**& matrix2d) {
98.     delete [] matrix2d[0];
99.     delete [] matrix2d;
100.    matrix2d = 0;
101. }
102.
103. int*** allocMatrix3D(int dim1, int dim2, int dim3) {
104.    int*** matrix3d = new int**[dim1];
105.    int**  dumm     = new int*[dim1*dim2];
106.    int*   d        = new int[dim1*dim2*dim3];
107.    for ( int i = 0; i < dim1; i++ ) {
108.        matrix3d[i] = dumm + i*dim2;
109.        for ( int j = 0; j < dim2; j++ )
110.            dumm[i*dim2+j] = d + (i*dim2+j)*dim3;
```

```
    }
    return matrix3d;
}

void deleteMatrix3D(int***& matrix3d) {
    delete [] matrix3d[0][0];
    delete [] matrix3d[0];
    delete [] matrix3d;
    matrix3d = 0;
}

int**** allocMatrix4D(int dim1,int dim2,int dim3,int dim4) {
    int**** matrix4d = new int***[dim1];
    int***  dumm     = new int**[dim1*dim2];
    int**   dum      = new int*[dim1*dim2*dim3];
    int*    d        = new int[dim1*dim2*dim3*dim4];
    for ( int i = 0; i < dim1; i++ ) {
        matrix4d[i] = dumm + i*dim2;
        for ( int j = 0; j < dim2; j++ ) {
            dumm[i*dim2+j] = dum + (i*dim2+j)*dim3;
            for ( int k = 0; k < dim3; k++ )
                dum[(i*dim2+j)*dim3+k] =
                    d + ((i*dim2+j)*dim3+k)*dim4;
        }
    }
    return matrix4d;
}

void deleteMatrix4D(int****& matrix4d) {
    delete [] matrix4d[0][0][0];
    delete [] matrix4d[0][0];
    delete [] matrix4d[0];
    delete [] matrix4d;
    matrix4d = 0;
}
```

The matrices are allocated and deallocated in the same way as for two-dimensional case, although the code becomes discouragingly complicated as the number of dimensions grows... The output of the program:

```
 Middle element: 9
     should be : 9
 Last element  : 16
     should be : 16

3-dimensional matrix
 Middle element: 16
     should be : 16
 Last element  : 28
     should be : 28

4-dimensional matrix
 Middle element: 19
     should be : 19
 Last element  : 33
     should be : 33
```

Prev: 12.3 Deallocation of memory — operator **Up:** 12. Dynamic memory management
Next: 12.5 Memory management in C

12.5 Memory management in C

The **new** and **delete** operators are specific to C++. In C, the memory can be managed by several *functions* (not operators) — they will be accessible after including the header `cstdlib` (or `stdlib.h`). These functions are often used even in programs which are written in C++ rather than in pure C,r because sometimes they can be more effective, although **new** and **delete** are easier in programming and safer.

The function **malloc** (from *'memory allocation'*) allocates a region of the free memeory (the heap). Its prototype is

```
void* malloc(size_t size);
```

where the type **size_t** is an alias (**typedef**'ed) of an unsigned integer type (e.g., **unsigned long**). The function allocates ***size*** bytes on the heap and returns the address of the allocated segment as a raw pointer of type **void***. There is no difference between allocating memory for a single object and for an array — in both cases ***size*** must be the total number of *bytes* requested and has to be specified in round parentheses, never in brackets (as **malloc** is a function requiring an argument). Returned value is of type **void*** so it does not convey any information on the type of an object which is to be stored in the allocated region of memory. Therefore, one has to explicitly cast it to the required type. For example:

```
1.    int* k = (int*) malloc(sizeof(int));
2.    *k = 5;
3.    // ...
4.    int size;
5.    cin >> size;
6.    int* m = (int*) malloc(size*sizeof(int));
7.    for (int i = 0; i < size; ++i)
8.        m[i] = 2*i;
```

Note that in the first line we allocate memory for a single **int** — still we have to specify number of bytes it will take. In the sixth line we allocate an array of size ***size*** with elements of type **int**, therefore the number of bytes needed is `size*sizeof(int)`.

In case of failure the function returns the empty pointer (**nullptr**).

The memory allocated by **malloc** is *not* initialized in any way. If we want to allocate a region and fill it with zeroes, we can use another function, **calloc**:

```
void* calloc(size_t count, size_t size);
```

where ***count*** denotes the number of objects which are to be stored in the allocated memory and ***size*** is the size (in bytes) of one such object. For example, in order to allocate memory for an array of **int** s of size ***dim*** and fill it with zeroes, we could write

```
int dim;
cin >> dim;
int* tab = (int*) calloc(dim,sizeof(int));
```

Note that still there is no information on the type of elements — pointer of type **void*** is returned.

There is also a function **realloc** which changes the size of a segment of memory already allocated before:

```
void* realloc(void* ptr, size_t size);
```

The value of the first argument must be the address returned by a call to **malloc**, **calloc** or **realloc**. The value of ***size*** is a new size requested (in bytes). If it is larger than the size used when allocating memory pointed to by ***ptr***, then a new region will be allocated, the contents of the old block of memory will be copied to the new one and the old block will be freed. The returned value is always the address of the new block — it can be equal to the address of the old block, if the size requested is smaller than the current size of the block pointed to by ***ptr***. In case of failure **nullptr** is returned and the old block of memory remains intact.

Memory allocated can be freed by the function **free**:

```
void free(void* ptr);
```

The value of ***ptr*** must be the address returned earlier by a call to **malloc**, **calloc** or **realloc**.

Memory allocated by the new (new[]) operator must ber released by delete (delete[]). If memory has been allocated by malloc, calloc or realloc, it must be freed by function free.

It is illegal to mix C forms of allocating with C++ forms of deallocating memory, and *vice versa.*

Prev: 12.4 Dynamic multidimensional arrays **Up:** 12. Dynamic memory management **Next:** 12.6 Functions operating on memory

12.6 Functions operating on memory

There are several useful functions operating on memory in the standard library. They are accessible after including the header `cstring` (or `string.h`). Some of them are:

void* memcpy(void* target, const void* source, size_t len) — copies ***len*** bytes from location pointed to by ***source*** to the memory region which starts at address ***target***; returns ***target***. The source segment of memory cannot overlap with the target one. For example, to copy an array of integers ***tab*** of the size ***size*** to a newly allocated array pointed to by ***t***, we can write

```
int* t = new int[size];
memcpy(t,tab,size*sizeof(int));
```

which can be much faster than copying the array element by element in a loop.
Note that copying is from right to left: the first argument defines the target and the second points to the source!

void* memmove(void* target, const void* source, size_t len) — as **memcpy**, but the target and source memory regions can overlap; slower than **memcpy**.

void* memchr(const void* str, int b, size_t len) — starting from the address pointed to by ***str*** looks for the first byte equal to the least significant byte of ***b*** and returns the address of the byte found; returns **nullptr** if such a byte has not been found among the first ***len*** bytes of ***str***.

int memcmp(const void* p, const void* q, size_t len) — compares the first ***len*** bytes of two memory regions which start at locations pointed to by ***p*** and ***q***; returns a negative integer if the first string is lexicographically less than the second, 0 if they are equal and a positive value if the first string is lexicographically greater than the second.

void* memset(void* p, int ch, size_t len) — copies the least significant byte of ***ch*** into each of ***len*** bytes beginning at ***p***. Returns ***p***.

The following program illustrates the functions just described:

P96: rotateE.cpp Functions operating on memory

```
1.  #include <iostream>
2.  #include <cstring>   // memcpy, memmove
3.  using namespace std;
4.
5.  template <typename T>
6.  T* rotate_left(T arr[], size_t size, size_t shift) {
7.
8.      if ((shift %= size) == 0) return arr;          ❶
9.
10.     T* aux = new T[shift];
```

```
11.
12.     memcpy(aux,arr,shift*sizeof(T));                ❷
13.     memmove(arr,arr+shift,(size-shift)*sizeof(T));
14.     memcpy(arr+size-shift,aux,shift*sizeof(T));
15.
16.     delete [] aux;
17.     return arr;
18. }
19.
20. template <typename T>
21. void writeArr(const char* mes, const T arr[], size_t size) {
22.     cout << mes << ": " << "[ ";
23.     for (size_t i = 0; i < size; ++i)
24.         cout << arr[i] << " ";
25.     cout << "]" << endl;
26. }
27.
28. int main() {
29.     char arrc[] = {'a','b','c','d','e','f'};
30.     writeArr("   char array",arrc,6);
31.     rotate_left(arrc,6,8);
32.     writeArr(" rotated by 8",arrc,6);
33.     rotate_left(arrc,6,1);
34.     writeArr("and then by 1",arrc,6);
35.
36.     cout << endl;
37.
38.     int arri[] = {1,2,3,4,5,6,7,8,9};
39.     writeArr("    int array",arri,9);
40.     rotate_left(arri,9,7);
41.     writeArr(" rotated by 7",arri,9);
42. }
```

The function **rotate_left** shifts elements of an array of size ***size*** by ***shift*** positions to the left in such a way that elements flowing out on the left side appear on the right side (rotation). In order to avoid a situation when ***shift*** > ***size***, on line ❶ we take the remainder of division of ***shift*** by ***size***. On line ❷, we copy ***shift*** elements from the beginning to an auxiliary array, then we move the remaining elements to the left using the function **memmove**, and finally we insert (copy) the elements from the auxiliary array at the end of our array (not forgetting about releasing the auxiliary array). The output of this program reads:

```
   char array: [ a b c d e f ]
 rotated by 8: [ c d e f a b ]
and then by 1: [ d e f a b c ]

    int array: [ 1 2 3 4 5 6 7 8 9 ]
 rotated by 7: [ 8 9 1 2 3 4 5 6 7 ]
```

Prev: 12.5 Memory management in C **Up:** 12. Dynamic memory management **Next:** 12.7 Placement new operator

12.7 Placement *new* operator

There is a special form of the **new** operator which allocates memory inside a segment already allocated before. The syntax is as follows:

```
new (address) Type;
new (address) Type[dim];
```

The operator is accessible after including the header file **new**. The expression in round parentheses must be an expression whose value is an address inside or at the beginning of a memory region allocated on the heap before. In this way we can allocate memory for a single object or for an array at the address specified. Note that no allocation is actually needed, as this region of memory has already been allocated; the operation in therefore very fast. To reclaim this segment of memory, one should use **delete** operator with the original address used in "normal" **new**.

In the example below we allocate (❶) an array ***arr*** of characters (which means bytes) of the size sufficient for three ***sz***-element arrays of different types (**string**, **double** and **int**). Inside the allocated region of memory, we then "allocate" three separate arrays, calculating (see, e.g., ❷) where in memory they should start, so they fit in the available memory:

P97: pnew.cpp Placement *new* operator

```
#include <iostream>
#include <string>

int main() {
    using std::string; using std::cout; using std::endl;
    int sz = 3;

    char* arr = new char[sz*(sizeof(string) +          ❶
                             sizeof(double)+sizeof(int))];

    string* nam = new (arr) string[sz]{"Sue", "Kim", "Joe"};
    double* wei = new (arr+sz*sizeof(string))          ❷
                            double[sz]{55.5, 61.2, 81.5};
    int* hei = new (arr+sz*(sizeof(string)+sizeof(double)))
                            int[sz]{170, 165, 183};
    for (int i = 0; i < sz; ++i)
        cout << nam[i] << " " << wei[i] << " "
             << hei[i] << endl;
    delete [] arr;                                     ❸
}
```

The printout

```
Sue 55.5 170
Kim 61.2 165
Joe 81.5 183
```

shows that indeed all arrays have been allocated and initialized. We remember, of course,

to deallocate memory they occupy (❸) using the address obtained by the “real” **new**!

This kind of memory allocation is also used when one wants to reuse, for new data, a region of memory which has already been allocated but contains data which are not needed anymore — it is much faster than deallocating and then allocating memory anew.

Prev: 12.6 Functions operating on memory **Up:** 12. Dynamic memory management **Next:** 13. C-structures and unions

T.R. Werner, September 27, 2023; 22:32

13. C-structures and unions

The C++ language is an object-oriented language. Therefore, it is possible to define our own types of data, together with operations that can be performed on objects of this type. We do it by defining **classes** (as in other object oriented languages). For historical reasons, classes in C++ come it two flavors: classes and structures.

However, compound types also exist in traditional C: these are structures and unions. In C++ the implementation of structures has been extended, so structures and classes basically do not differ very much. Still, it is worthwile to know the implementation of structures in C, as they are often used in C++ programs; in fact many types used by the standard library are pure C-like structures so they can be used in both C and C++ programs. Structures that do not use any extensions of C++ will be called C-structures.

Subsections

- 13.1 C-structures
- 13.2 Templates of structures
- 13.3 Unions

Prev: 12.7 Placement new operator **Up:** Programming in C++ **Next:** 13.1 C-structures

13.1 C-structures

Structures define new types of variables. Unlike the primitive types, like **int** s or **double** s, one variable of a structure type can contain many pieces of information. Unlike it is in arrays, these pieces of information do not have to be all of the same type.

C-structure is a collection of named members which can be of different types, also other structure types.

C-structure can be defined with the following syntax:

```
struct AName {
    Type1 memb1;
    Type2 memb2;
    ...
};
```

The names ***Type1*** and ***Type2*** are names of types (other than the name of the structure type being defined, **AName** here). The names ***memb1*** and ***memb2*** are arbitrary identifiers of **fields** of this structure. Each variable of this structural type will contain **members** of the types and names corresponding to the structure's fields. So far it is only the definition of a type; there are no variables of this type.

A semicolon after the closing brace is necessary!

The name of the type just defined is **AName** in C++ and **struct AName** in C. In C the keyword **struct** is a part of the type's name; in C++ it can (but does not have to) be omitted.

Having defined a type we can create objects (variables) of this type. The syntax is the same as for primitive types (like **double** or **int**):

```
struct AName a, b;   /* C and C++ */
AName  c, d;         // C++ only
```

We can also define variables of this type directly after its definition, before the semicolon:

```
struct AName {
    Type1 memb1;
    Type2 memb2;
} a, b, c, d;
```

defines a structural type and four variables of this type. It is also possible to define variables of an anonymous structure:

```
struct {
    Type1 memb1;
    Type2 memb2;
} a, b;
```

Here we constructed two objects named ***a*** and ***b***. Each of them contains two members

corresponding to the two fields of the structure. We can access these members exactly in the same way as we do it for members of other, named, structure types (see below). However, we will not be able to create any other variable of this type, as it does not have a name by which we could refer it.

Each object (variable) of a C-structure has as many members as there are fields in its definition — they have corresponding names and types. In order to access a member of an object it is therefore not enough to specify the name of a member, We have to identify which object is meant, as members with the same name but belonging to different objects are completely independent. One can do it using a member selection operator — one of those listed on positions 4 and 5 in the table of operators, depending on the way we refer to the object: by the name of the object itself or by the name of a pointer which points to this object. If a name ***a*** is the name of an *object*, then to access its memebr named ***memb*** we use the “dot” operator:

```
a.memb
```

The same rule would apply if ***a*** would be the name of a reference to an object (recall, however, that there are no references in C). Suppose now that an object is referred to by the name of a *pointer* ***pa*** which currently points to this object. Then to access its member one uses the “arrow” operator:

```
pa->memb
```

where the arrow is a digraph composed of a hyphen and a 'greater than' characters.

The same rule applies not only to C-structures but to structures and classes in general.

Note that the form `pa->memb` is essentially a shorthand notation for `(*pa).memb`, as `*pa` denotes the object pointed to by ***pa***. Hence, if ***a*** is the identifier of an object of a structure which has a field named ***x*** of type **double**, and ***pa*** is a pointer which points to this object, then the following statements are equivalent:

```
1.    a.x     = 3.14;
2.    (&a)->x = 3.14;
3.    pa->x   = 3.14;
4.    (*pa).x = 3.14;
```

The parentheses on lines 2 and 4 are required, as the member-selection operators (dot and arrow) have higher precedence than the dereference and address operators (' `*`' and ' `&`').

Summarizing,

the symbol `a.b` denotes a member *b* of an object *a*, while `pa->b` denotes a member *b* of the object *pointed to* by a pointer *pa*.

We can declare/define an object of a C-structure and initialize it in the same statement. The syntax is similar to that of initializing arrays:

```
AName ob = {expr_1, expr_2};
```

where ***expr_1*** and ***expr_2*** are expressions the values of which are to be used to initialize members of the object being created. They must be given in the order of the corresponding fields in the structure's definition — as in the case of arrays, if the number of initializers is less than the number of fields, the remaining members will be initialized with zero of an appropriate type. The keyword **struct** is necessary in C; in C++ it can be omitted.

The program below defines a structure **Car** (❶) which describes, in a very simplified way, cars. It has two fields: ***speed*** and ***year*** of type **double** and **int**, respectively. Two global variables, ***skoda*** and ***fiat***, of this type are defined immediately in the same statement. The latter is also initialized by the values in braces — they are given in the order corresponding to the order of fields in the definition of the structure. The variable ***skoda*** is created, but not initialized — its members remain undefined.

P98: cstruct.cpp C-struktury

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. struct Car {                                  ❶
 5.     double speed;
 6.     int    year;
 7. } skoda, fiat = {100, 1998};
 8.
 9. void pr(const char*,const Car*);
10.
11. int main() {
12.     Car toyota, *myCar = &toyota, *vw;        ❷
13.
14.     cout << "Size of \'Car\' objects is "     ❸
15.          << sizeof(Car) << " bytes\n";
16.     skoda.speed =  120;                       ❹
17.     skoda.year  = 1995;
18.
19.     toyota.year  = 2012;                      ❺
20.     myCar->speed =  180;
21.
22.     vw = new Car{175,2003};                   ❻
23.     vw->speed =  175;
24.     vw->year  = 2003;
25.
26.     pr("Skoda ",&skoda);
27.     pr("Fiat  ",&fiat);
28.     pr("Toyota",&toyota);
29.     pr("myCar ", myCar);
30.     pr("VW    ", vw);
31.
32.     delete vw;
33. }
34.
35. void pr(const char *name, const Car *car) {
36.     cout << name << ": speed " << car->speed
37.          << ", year " << car->year << endl;
38. }
```

Inside **main** function we define another object of type **Car** named ***toyota***. We also define two pointers: ***myCar***, initialized with the address of ***toyota***, and ***vw*** which we leave uninitialized (line ❷).

We print the size of objects of type **Car** on line ❸. This size could be 12 bytes (8 bytes for the member ***speed*** of type **double** and 4 bytes for the member ***year*** of type **int**). The exact size can be different on some platforms, which, e.g., prefer sizes which are multiples of 8; each object contains then some unused bytes — the so called **padding**: this is the case for the author's system, as can be seen from the printout:

```
Size of 'Car' objects is 16 bytes
Skoda : speed 120, year 1995
Fiat  : speed 100, year 1998
Toyota: speed 180, year 2012
myCar : speed 180, year 2012
```

```
VW    : speed 175, year 2003
```

We asign sensible values to the members of the object ***skoda*** on lines ❹ and the next. As ***skoda*** is the name of an *object*, and not of a pointer, we use the “dot” notation.

The object ***toyota*** is assigned values on lines ❺ and the next. On line ❺ we refer to this object by its name, so the “dot” notation is used again. However, in the next line we refer to the *same object* (see line ❷) but using the pointer ***myCar***. Therefore we use the notation with an “arrow” here.

Another object of type **Car** is created on line ❻. This time we allocate it on the heap (using **new**) and store the address returned in pointer ***vw***. The object created in the free memory is initialized by means of a list in curly braces; this is allowed in the new standard C++11 (but note, that using equal sign before opening brace would be an error).

The function **pr** prints the information on cars passed to it by pointer of type **const Car***. Therefore, when we invoke it, we have to pass addresses; we can use pointer variables (whose value are addresses) or, if we want to use the name of an object, we have to extract its address with the operator ' `&`'.

C-structures are heavily used in the libraries. Let us consider an example. Including the header `sys/timeb.h` we make accessible a structure type **timeb** and functions operating on variables of this type, in particular the function **ftime** (actually, the header `sys/timeb.h` is not standard, but it is present in most implementations of C/C++, including Linux **gcc** and the Microsoft's compilers). The function **ftime** takes the *address* of an object of type **timeb**, which is a structure type defined as:

```
struct timeb {
    time_t          time;
    unsigned short  millitm;
    short           timezone;
    short           dstflag;
};
```

Given the address of an object, the functions fills it out with system data about current time. The member ***time*** will be the number of seconds since 1st of January 1970 (beginning of the Unix epoch). Its type, **time_t**, is equivalent (by means of a **typedef**) to a signed integer type. This is usually **long**, so for 32-bit machines, where `sizeof(long)` is 4, the maximum value of ***time*** is 2^{31} -1 = $2147483647 \approx 2.1 \cdot 10^9$ and will be reached in the year 2038 (one year is approximately equal to $10^7 \pi$ seconds).

The member ***millitm*** will be the number of milliseconds since the beginning of the last full second. The members ***timezone*** and ***dstflag*** are not very useful.

Using the function **ftime** one can measure the execution time of a part of a program (although it is not the best way to do it):

P99: tim.cpp Using library structures

```
1. #include <iostream>
2. #include <cmath>          // sin, cos
3. #include <sys/timeb.h>    // ftime
4. using namespace std;
```

```
5.
6. int main() {
7.     timeb start, now;                                    ❶
8.     double res = 0;
9.
10.    ftime(&start);                                       ❷
11.
12.    for (int i = 0; i <= 90000000; ++i) {
13.        if (i%10000000 == 0) {
14.            ftime(&now);                                 ❸
15.            time_t sec = now.time - start.time;
16.            int msec = now.millitm;
17.            msec -= start.millitm;
18.            if (msec < 0) {
19.                --sec;
20.                msec += 1000;
21.            }
22.            cout << "After " << i << " iterations: "
23.                 << sec << "s and " << msec << "ms\n";
24.        }
25.        res = cos(res+sin(i));
26.    }
27.    cout << "Useless result: " << res << endl;
28. }
```

We define (line ❶) two variables of type **timeb**: ***start*** and ***now***. The first is filled out with current data by a call to the function **ftime** (line ❷). The second is updated every 10 million iterations of the loop (❸) and the difference between the current time and the start time of the program is printed:

```
After 0 iterations: 0s and 0ms
After 10000000 iterations: 0s and 875ms
After 20000000 iterations: 1s and 746ms
After 30000000 iterations: 2s and 618ms
After 40000000 iterations: 3s and 492ms
After 50000000 iterations: 4s and 366ms
After 60000000 iterations: 5s and 241ms
After 70000000 iterations: 6s and 120ms
After 80000000 iterations: 6s and 998ms
After 90000000 iterations: 7s and 885ms
Useless result: 0.953078
```

The variable ***res*** is calculated here just to give the program something to work on...

Structures can contain, as their members, other structures. Of course, not of the same type, as these would also have to contain such substructures, and those in turn would also have to contain such substructures, and so on *ad infinitum*.

Suppose we want to have a type describing points on a plane. A structure consisting of two fields of type **double** (corresponding to Cartesian coordinates of the point) could provide a natural representation:

```
struct Point {
    double x, y;
};
```

Now we can define the triangle as a structure containing three points corresponding to its vertices:

```
struct Triangle {
    Point A, B, C;
};
```

C-structures do not contain any methods or constructors. In order to operate on them, one has to use global functions. In the example below, we define a function which represents the operation of rotation of points around the origin. A rotation through the angle φ (in radians) can be described in terms of coordinates by the formulae:

$$x' = x \cos\varphi - y \sin\varphi \qquad y' = x \sin\varphi + y \cos\varphi$$

where coordinates of rotated points are marked with a prime. The operation is performed by the function **rot** declared on line ❸ of the program:

P100: rotat.cpp Structures as members of structures

```
1.  #include <iostream>
2.  #include <cmath>     // sin, cos
3.  using namespace std;
4.
5.  struct Point {
6.      double x, y;
7.  };
8.
9.  struct Triangle {
10.     Point A, B, C;
11. };
12.
13. void info(const Point*);     ❶
14. void info(const Triangle*); ❷
15. void rot(Point*,double);     ❸
16. void rot(Triangle*,double); ❹
17.
18. int main() {
19.     Point A;
20.     A.x = -1;
21.     A.y =  2;
22.
23.     Point B = { -1, 1 };
24.
25.     Point C = { 2 };
26.     C.y = -1;
27.
28.     Triangle T = { A, B };
29.     T.C = C;
30.
31.     cout << "Initial points: ";
32.     info(&A); info(&B); info(&C);
33.     cout << "\nTriangle: ";
34.     info(&T);
35.     cout << endl;
36.
37.     rot(&A,90); rot(&B,90); rot(&C,90);                ❺
38.     cout << "A, B, C after rot. through 90 deg:\n    ";
39.     info(&A); info(&B); info(&C);
40.     cout << endl;
41.
42.     rot(&T,90); rot(&T,90);                            ❻
43.     cout << "T after rot. through  90 deg. twice:\n    ";
44.     info(&T);
45.
46.     rot(&T,180);                                       ❼
47.     cout << "T after rot. through 180 deg. again:\n    ";
48.     info(&T);
49. }
50.
51. void info(const Point* pP) {
52.     cout << "(" << pP->x << ", " << pP->y << ") ";
53. }
54.
55. void info(const Triangle* pT) {
56.     cout << "A="; info(&pT->A);                         ❽
57.     cout << "B="; info(&pT->B);
58.     cout << "C="; info(&pT->C);
```

```
59.     cout << endl;
60. }
61.
62. void rot(Point* pP, double phi) {
63.     static double conver = atan(1.)/45;
64.     phi = phi*conver; // degrees -> radians
65.
66.     double c = pP->x;
67.     pP->x = pP->x * cos(phi) - pP->y * sin(phi);
68.     pP->y =     c * sin(phi) + pP->y * cos(phi);
69. }
70.
71. void rot(Triangle* pT, double phi) {
72.     rot( &pT->A, phi);                                  ❾
73.     rot( &pT->B, phi);
74.     rot( &pT->C, phi);
75. }
```

The function **rot** takes a point by pointer, so it can work on the original and modify it (that is why we use “arrows” and not dots in its definition). Let us define three points A = (- 1, 2), B = (- 1, 1) and C = (2, - 1), as on the left hand side of the figure:

The points are created at the beginning of the **main** function. They are either initialized completely, or only partly, or not at all — uninitialized members can then be assigned values “by hand”.

We then define a triangle with vertices in ***A***, ***B*** and ***C***. Note that assigning values of ***A***, ***B*** and ***C*** to the members of ***T*** implies copying them — the original points and the members of ***T*** are independent. Printing information on the triangle and points we see that the situation corresponds to the left hand side of the figure.

The three points are then rotated by the function **rot** through $90°$ (line ❺). The functions calculates new coordinates and changes the members of the input objects — as the points were passed by pointer, it means that the original points become modified. New location of points is depicted in the middle part of the figure; note that the triangle ***T*** has not been modified as its members are copies of the original points before rotation.

```
Initial points: (-1, 2) (-1, 1) (2, -1)
Triangle: A=(-1, 2) B=(-1, 1) C=(2, -1)

A, B, C after rot. through 90 deg:
    (-2, -1) (-1, -1) (1, 2)
T after rot. through  90 deg. twice:
    A=(1, -2) B=(1, -1) C=(-2, 1)
T after rot. through 180 deg. again:
    A=(-1, 2) B=(-1, 1) C=(2, -1)
```

Now we rotate the triangle (line ❻). Note that the function rotating triangles has the name **rot**, exactly the same as the function which rotates the points (❸ and ❹). However, its parameters are of different type, so the compiler will know which one is meant (similar overloading is used for two functions **info**: for points ❶ and for triangles ❷). The function **rot** for triangles is very simple and does not require any knowledge of trigonometry: it just

rotates all three vertices using the previous version of the **rot** function and passing to it the addresses of the three members of a triangle for which it was invoked (which themselves are points). The strange looking construct `&pT->A` on lines ❽ and ❾ is correct: it denotes the address of the point ***A*** which is a member of an object of type **Triangle** pointed to by ***pT***. The notation `&(pT->A)` would be perhaps more legible, but is not necessary, as the precedence of member selection operator (' `->`') is higher than that of the address operator (' `&`').

After rotating the triangle twice through 90°, i.e., through 180°, it is located as shown in the right hand side of the figure: the printout shows that this is in fact the case. Rotating the triangle through 180° again (line ❼) reverts it to the initial position (see the last line of the printout).

For obvious reason, a structure cannot contain members of its own type. However, it *can* contain, as its member, pointers to objects of its type. This is often used for building *lists*. Each object (element) of a list contains some data and also the address of the next element of this list — what we get is called one-directional, or singly linked, list (there are also two-directional, or doubly linked, lists, where each object contains addresses of the next and previous elements of the list).

Let us now construct such a (simplified) singly linked list. We define a structure **Node** which contains, as fields, some data. In our example these are just two **double** s ***wdth*** and ***hght*** — very often it would rather be a pointer to an object which holds more complicated data. Besides the data, there will be an additional field ***next*** of type **Node***. It will hold the address of the node which comes next in the list. For the last element in the list there is no next element, so the member ***next*** of this node will be set to **nullptr**. Sometimes it is more convenient if it points to the node itself, or to the first node — we then get a cyclic list).

P101: simplist.cpp A simple list

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Node {
5.      double wdth;
6.      double hght;
7.      Node *next;
8.  };
9.
10. void put_data(Node* n, double s, double w, Node* next);
11. void print_list(const Node* n);
12. void print_list_reverse(const Node* n);
13.
14. int main() {
15.     Node A = {4, 44, nullptr};          ❶
16.     Node B, D, *head;
17.
18.     Node* pC = new Node;                ❷
19.
20.     put_data(&B,3,33,&A);
21.     put_data(pC,2,22,&B);
22.     put_data(&D,1,11,pC);
23.
24.     head = &D;                          ❸
25.
26.     print_list(head);
27.     print_list_reverse(head);
```

```
28.
29.     delete pC;
30. }
31.
32. void put_data(Node* n, double s, double w, Node* next) {
33.     n->wdth = s;
34.     n->hght = w;
35.     n->next = next;
36. }
37.
38. void print_list(const Node* n) {
39.     for ( ; n; n = n->next )
40.         cout << n->wdth << " " << n->hght << "; ";
41.     cout << endl;
42. }
43.
44. void print_list_reverse(const Node* n) {
45.     if (n == nullptr) return;  // empty list
46.     if (n->next != nullptr)            ❹
47.         print_list_reverse(n->next);
48.     cout << n->wdth << " " << n->hght << "; ";
49. }
```

We create a variable ***A*** of type **Node** on line ❶ and initialize its three members using the syntax allowed for C-structures (but not for more complicated structures in C++). In the next line we define two more nodes, ***B*** and ***D***, and a pointer to objects of this structure ***head***. On line ❷ we allocate on the heap another object; the address returned by **new** is assigned to ***pC***. The objects are then filled with data by invoking the function **put_data**. It takes the address of an object which is to be filled together with data, represented by two **double** s, and address of another object which will be assigned to the member ***next*** as the address of the next object of the list. In order to extract the address of objects ***A***, ***B*** and ***D***, we use the address operator ' &'; for ***pC*** we do not do it because it is already a pointer with an address as its value (the object which is pointed to by ***pC*** is on the heap and does not have any name). Therefore, ***B*** contains the address of ***A*** as the next element, the object pointed to by ***pC*** has the address of ***B***, and for ***D*** it is ****pC*** which is the next. The situation is depicted in the figure

Note that the first element of the list is ***D*** (such element, or rather pointer to it, is called the **head** of the list). The last element is ***A*** (this is the **tail** of the list). Its ***next*** member contains the empty pointer (**nullptr**).

The address of the first element is remembered in the variable ***head*** (line ❸). Note that in order to have access to the whole list it is sufficient to know ***head***; in its ***next*** member we can find the address of the next element, etc.

Two functions illustrating some basic operations on lists are defined in the program: **print_list** and **print_list_reverse**. They both take only the ***head***: the address of the first

element of the list. The first of them loops over the elements of the list printing data from the nodes. After each iteration, the pointer ***n***, pointing to the current node, is assigned the value of '`n->next`', so it points to the next node. The loop terminates when ***n*** becomes **nullptr** — this must eventually happen because the last element (which is ***A***) has **nullptr** as the value of its ***next*** member.

The second of the two functions, **print_list_reverse**, is more interesting. Its task is to print data from the nodes but in reverse order: from the tail to the head. This is nontrivial, as nodes only know their successors, but not predecessors. The recurrence comes to the rescue here: we call the function passing the head (which is the address of ***D***). Before printing information on the current node, however, the function calls itself passing the pointer to its successor (line ❹); this incarnation of the function invokes itself again, this time passing the pointer to *its* successor, and so on. For ***A*** (the tail), the condition on line ❹ becomes false, so printing is executed and the last incarnation of the function returns to the incarnation invoked for ***B*** — now information on ***B*** is printed and the function returns to its incarnation for ***pC** etc. We see that traversing the list backwards was ensured by stack rewinding after consecutive returns from the function:

```
1 11; 2 22; 3 33; 4 44;
4 44; 3 33; 2 22; 1 11;
```

In the example above, we created some nodes on the stack and others on the heap (using new). This is not a good practice: normally, all nodes should be created on the heap (see section on templates of structures .)

Objects of C-structures can also be passed as a function argument or the value returned. However, one should remember that such objects can be quite large and passing them by value can be rather ineffective. It is then better to work on pointers to structures (or references), not the structures themselves; copying or moving objects physically in the computer's memory should then be avoided. The same applies to arrays of objects — it is often better to operate on arrays of pointers rather than of objects themselves.

In the program below, object of type **Writer** are at least 44 bytes long (40 bytes for the member ***name*** and 4 bytes for the integer member ***born***). However, the pointers are only 4 (or 8) bytes long:

P102: writers.cpp Using pointers for sorting arrays

```
1.  #include <iostream>
2.  #include <iomanip>   /* setw */
3.  using namespace std;
4.
5.  struct Writer{
6.      int   born;
7.      char  name[40];
8.  };
9.
10. void insertionSort(Writer*[],int);
11.
12. int main() {
13.     Writer gl = {1896,"Giuseppe Tomasi di Lampedusa"},
14.            ws = {1564,          "William Shakespeare"},
15.            ib = {1894,                   "Isaak Babel"},
```

```
16.             jg = {1749,  "Johann Wolfgang von Goethe"},
17.             fk = {1883,                  "Franz Kafka"},
18.             bs = {1892,                 "Bruno Schulz"};
19.
20.     Writer* writers[] = { &gl,&ws,&ib,&jg,&fk,&bs };     ❶
21.
22.     const int ile = sizeof(writers)/sizeof(writers[0]);
23.
24.     cout << "sizeof(Writer )=" << sizeof(Writer ) << endl;
25.     cout << "sizeof(Writer*)=" << sizeof(Writer*) << endl
26.                                                  << endl;
27.     insertionSort(writers, ile);
28.
29.     for ( int i =0; i < ile; i++ )                       ❷
30.         cout << setw(28) << writers[i]->name
31.              << setw(5)  << writers[i]->born << endl;
32. }
33.
34. void insertionSort(Writer* a[], int size) {
35.     if ( size <= 1 ) return;
36.
37.     for ( int i = 1 ; i < size ; ++i ) {
38.         int j = i;
39.         Writer* v = a[i];
40.         while ( j >= 1 && v->born < a[j-1]->born ) {     ❸
41.             a[j] = a[j-1];                               ❹
42.             j--;
43.         }
44.         a[j] = v;
45.     }
46. }
```

Suppose we want to sort writers according to their year of birth. It is better to sort a array of *pointers* to objects representing writers than an array of **Writer** objects. Sorting means copying and moving objects, but objects of type **Writer** are long while pointers are short. Therefore we build an array of pointers to **Writer** objects (line ❶) and pass it to a sorting function. The criterion of sorting is based on the value of the member ***born*** of the object pointed to by a pointer being an element of the array of pointers (line ❸), but only pointers from this array are copied and moved around, never the objects themselves (as on line ❹). Still, the the job has been done — we can print information on writers in the sorted order (❷):

```
sizeof(Writer )=44
sizeof(Writer*)=8

         William Shakespeare 1564
  Johann Wolfgang von Goethe 1749
                 Franz Kafka 1883
                Bruno Schulz 1892
                 Isaak Babel 1894
Giuseppe Tomasi di Lampedusa 1896
```

The manipulator **setw** has been used here just to better format the printout. The sorting function implements *insertion sort* algorithm (without sentinel).

Sometimes we want, or have to, declare a structure (union, class) without actually defining it. As the definition, and the size of objects in particular, of this type is not known, one cannot create any object of it. Such a type is, as we say, **incomplete**. The declaration is called a **forward declaration** and is sometimes unavoidable. The name of a structure which has been declared but lacks definition can be used in situations where a definition and size of objects is not necessary — e.g., to define pointers to such objects (but not to initialize them, as one cannot create any object which could be pointed to).

Suppose, for example, that a structure **AA** has a field which is the pointer to objects of type **BB**, and **BB** has a field which is the pointer to objects of type **AA**. Something like

```
1.      struct AA {
2.          BB *b;  // WRONG
3.          // ...
4.      };
5.
6.      struct BB {
7.          AA *a;
8.          // ...
9.      }
```

would be wrong because in the second line it is not known what the name **BB** denotes. Changing the order of definitions will not help, as we will have the same problem with **AA**. One has to use a *declaration* to tell the compiler that the name **BB** is the name of a structure which will be defined later. From now on one can define pointers to object of type **BB**, although one cannot create any object of this type. The declaration of a structure has the form

```
struct AName;
```

where **AName** is the name of a declared structure (the same applies to unions and classes). In our example we should write

```
1.      struct BB;
2.
3.      struct AA {
4.          BB *b;
5.          // ...
6.      };
7.
8.      struct BB {
9.          AA *a;
10.         // ...
11.     }
```

and now it is known that the name **BB** in the definition of **AA** is the name od a structure which will be defined later. The size of an object of **AA** can be calculated by the compiler because it contains only a pointer to **BB** but not an object of **BB**; the size of pointers is of course known to the compiler.

Let us consider an example:

P103: wife.cpp Forward declarations of structures

```
1.  #include <iostream>
2.  #include <cstring>   // strcpy
3.  using namespace std;
4.
5.  struct Husband;
6.  struct Wife;
7.
8.  void prinfo(const Husband*);
9.  void prinfo(const Wife*);
10.
11. struct Wife {
12.     Husband *hus;
13.     char name[20];
14. } heather = { 0, "Heather" } ;
15.
16. struct Husband {
17.     Wife *wif;
```

```
18.     char name[20];
19. } anthony  = { 0 } ;
20.
21. int main() {
22.     strcpy(anthony.name, "Anthony");          ❶
23.
24.     anthony.wif    = &heather;
25.     heather.hus = &anthony;
26.
27.     Husband zachary  = { 0, "Zachary" };
28.     Wife    cecilia = { 0, "Cecilia"};
29.
30.     prinfo(&anthony);
31.     prinfo(&heather);
32.     prinfo(&zachary);
33.     prinfo(&cecilia);
34. }
35.
36. void prinfo(const Husband *h) {
37.     cout << "Man:   " << h->name;
38.     if ( h->wif )
39.         cout << "; wife    "
40.              << h->wif->name << "\n";         ❷
41.     else
42.         cout << "; (single)\n";
43. }
44.
45. void prinfo(const Wife *w) {
46.     cout << "Woman: " << w->name;
47.     if ( w->hus )
48.         cout << "; husband "
49.              << (*((*w).hus)).name << "\n";  ❸
50.     else
51.         cout << "; (single)\n";
52. }
```

Two structures, **Husband** and **Wife**, are first declared. The declarations are necessary in order to declare the functions **prinfo** which have, as parameters, pointers to these structures. We cannot define these functions here, because in their body we will use names of the structures' members, which are not known yet. Note that we declare *two* functions of the same name. They differ sufficiently in the type of their parameter, so this overloading is valid.

We then define the structures **Husband** and **Wife**. This is the case when either of them contains a field which is the pointer to the other – therefore, declaring them was necessary. Note that it would be impossible for the structure **Wife** to contain a field of type **Husband** (and not **Husband***) if **Husband** would contain **Wife** — in such a case every **Wife** object would contain **Husband** which in turn would contain **Wife**, which in turn would contain **Husband**, etc., *ad infinitum*. The function **strcpy** on line ❶ copies C-strings and will be described in section on strings . The rest of the program should be clear; it produces:

```
Man:   Anthony; wife    Heather
Woman: Heather; husband Anthony
Man:   Zachary; (single)
Woman: Cecilia; (single)
```

Let us look at lines ❷ and ❸. On line ❷ the expression

```
h->wif->name
```

denotes the value of the member ***name*** of the object pointed to by the pointer ***wif*** which in turn is a member of the object pointed to by ***h***. In this way, having a pointer to a husband, we are able to extract the name of his wife. Line ❸ contains an analogous expression

```
(*((*w).hus)).name
```

which allows us to access the name of the husband given a pointer to his wife. Note how the notation with an “arrow” simplifies such expressions – all parentheses are necessary here.

Prev: 13. C-structures and unions **Up:** 13. C-structures and unions **Next:** 13.2 Templates of structures

13.2 Templates of structures

As functions, we can also define structures/classes in the form of *templates* depending on one or more type parameters. When we call a function defined in the form of a template, usually we don't have to specify types which should be substituted for the type parameters of our template: the compiler can infer necessary types itself. In the case of structures/classes, we usually (although not always) have to specify these types. In the example below, **Node** is the name of a template depending on one type parameter. When we create object of concrete types, we havei to specify this type: names of concrete types will be **Node<int>**, **Node<std::string>**, etc. This is illustrated in the following example:

P104: lists.cpp Simple singly-linked list as a template

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. template <typename T>
6. struct Node {
7.     T data;
8.     Node *next;
9. };
10.
11. template <typename T>
12. void addFront(Node<T>*& head, T data) {    ❶
13.     head = new Node<T>{data, head};
14. }
15.
16. template <typename T>
17. void addBack(Node<T>*& head, T data) {
18.     if (head == nullptr) addFront(head, data);
19.     else {
20.         Node<T>* tmp = head;               ❷
21.         while (tmp->next != nullptr)
22.             tmp = tmp->next;
23.         tmp->next = new Node<T>{data, nullptr};
24.     }
25. }
26.
27. template<typename T>
28. void printList(const Node<T>* h) {         ❸
29.     std::cout << "[ ";
30.     while (h != nullptr) {
31.         std::cout << h->data << " ";
32.         h = h->next;
33.     }
34.     std::cout << "]\n";
35. }
36.
37. template <typename T>
38. void deleteList(Node<T>*& h) {             ❹
39.     while (h != nullptr) {
40.         Node<T>* t = h->next;
41.         delete h;
42.         h = t;
43.     }
44. }
45.
46. int main() {
```

```
47.         // "somestring"s will be a literal of type std::string
48.     using namespace std::literals;        ❺
49.
50.     Node<int>* headI{nullptr};
51.     addBack(headI,  3);
52.     addBack(headI,  4);
53.     addFront(headI, 2);
54.     addFront(headI, 1);
55.     printList(headI);
56.     deleteList(headI);
57.
58.     Node<std::string>* headS{nullptr};
59.     addBack(headS,  "hearts"s);
60.     addBack(headS,  "spades"s);
61.     addFront(headS, "diamonds"s);
62.     addFront(headS, "clubs"s);
63.     printList(headS);
64.     deleteList(headS);
65. }
```

The **Node** class template describes one node of a singly-linked list. The type of data is not specified — it is a type parameter of the template (denoted here as **T**). We define two functions (again, as templates) which add new elements to a list. In the **main** function, the list is represented by the pointer to its first node (traditionally called ***head***) — its value **nullptr** corresponds to an empty list.

The **addFront** function gets the head and data: it then creates a new node containing this data and its ***next*** pointer is set to the current value of the head; the new node now becomes the head and the previous head will be now the second element of the list. Notice that the head is passed by reference (❶), so the function can modify it.

The **addBack** is similar, but adds a new element at the end of the list. If the list is empty, **addFront** is invoked, as it handles such case without any problem. Otherwise, we iterate over the list stopping at the last node – the one which has **nullptr** in its ***next*** field. This field is then set to the address of the new node, which now becomes the last one. Notice that in order to iterate over the list, we had to make a copy of the head (❷). This is because we got the head by reference (original), so we cannot modify it in an arbitrary way!
It's different in the **printList** function (❸) — here the pointer is passed by value, so we modifications inside the function do not have any impact on the original.
As all nodes have been created on the heap (using the **new** operator), we cannot forget about the function **deleteList**, which deletes all the nodes one by one (❹).

In the program, we create a list with data of type **int**, and then a list with data of type **string**. Note that the strings, which are passed as data to the latter, had to be written as literals of type **string** — without the 's' suffix, they would be of type **const char***, which would not agree with the type of elements of the list. In order to use this suffix, we had to "open" the `std::literals` namespace (❺). The program prints

```
[ 1 2 3 4 ]
[ clubs diamonds hearts spades ]
```

The next, similar, example illustrates how one can implement a stack using a singly-linked list

P105: StackTmplt.cpp Implementation of the stack with templates

```
1. #include <iostream>
```

```
#include <string>

template <typename T>
struct Node {
    T     data;
    Node* next;
};

template <typename T>
void push(Node<T>*& head, T d) {
    head = new Node<T>{d, head};
}

template <typename T>
T pop(Node<T>*& head) {
    T d{head->data};
    Node<T>* n{head->next};
    delete head;
    head = n;
    return d;
}

template <typename T>
bool empty(Node<T>* head) {
    return head == nullptr;
}

int main() {
      // "something"s is now a literal of type std::string
    using namespace std::literals;

    Node<int>* headI{nullptr};
    Node<std::string>* headS{nullptr};
    push(headI, 3); push(headS, "3"s);
    push(headI, 2); push(headS, "2"s);
    push(headI, 1); push(headS, "1"s);
                    push(headS, "0"s);
    while(!empty(headI))
        std::cout << pop(headI) << " ";
    std::cout << std::endl;

    while(!empty(headS))
        std::cout << pop(headS) << " ";
    std::cout << std::endl;
}
```

Templates allow us to build stacks of elements of various types (**int** and **string** in the example above). The output of the program is

```
1 2 3
0 1 2 3
```

Prev: 13.1 C-structures **Up:** 13. C-structures and unions **Next:** 13.3 Unions

13.3 Unions

Unions provide another example of compound types. They are somewhat similar to structures — there is however an important difference:

> **all members of an object of union type are located at the same address in the computer's memory. Hence, at every moment, only one of all these members is accessible.**

Assigning a value to one of the members of a union object overwrites its previous contents. It follows that the size of a union object must be at least equal to the size of its longest field, but does not have to be larger. Actually, quite often it *is* larger than the longest field: the exact size can depend on implementation and the computer's architecture, in particular on its **alignment** requirements.

In the following example

P106: uns.cpp Unions

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  union Bag;
5.
6.  void put(Bag*,float);
7.  void put(Bag*,long double);
8.  void inf(const Bag*);
9.
10. union Bag {
11.     float       numberF;
12.     long double numberLD;
13. } bag ;
14.
15. int main() {
16.     cout << "      sizeof(float)=" << sizeof(float) << endl;
17.     cout << "sizeof(long double)="
18.          << sizeof(long double) << endl;
19.     cout << "        sizeof(Bag)=" << sizeof(Bag) << endl;
20.
21.     put(&bag, 3.14F);                    ❶
22.     inf(&bag);
23.
24.     put(&bag, 3.14L);                    ❷
25.     inf(&bag);
26. }
27.
28. void put(Bag *w, float f) {
29.     w->numberF = f;
30. }
31.
32. void put(Bag *w, long double ld) {
33.     w->numberLD = ld;
34. }
35.
36. void inf(const Bag *w) {
```

```
37.     cout << "\nnumberF : " << w->numberF  << endl;
38.     cout <<   "numberLD: " << w->numberLD << endl;
39. }
```

any object of the union **Bag** holds a number of type **float** (4 bytes) *or* of type **long double** (12 or 16 bytes), but not both of them simultaneously. Every assignment to the member ***numberF*** will overwrite *both* members, because they are located at the same address — the same will happen if we assign a value to the member ***numberLD***.

We assign the value 3.14 to the member ***numberF*** of the object ***bag*** on line ❶. The function **inf** print the values of both members; ***numberF*** is in fact 3.14, but the value of ***numberLD*** seems to be completely random:

```
      sizeof(float)=4
sizeof(long double)=16
        sizeof(Bag)=16

numberF : 3.14
numberLD: 3.93143e-4942

numberF : 1.90232e+17
numberLD: 3.14
```

On line ❷ we assign the same value to the member ***liczbaLD*** of the same object ***bag***. Now ***numberLD*** is 3.14, but ***numberF*** looks as a random number (this is the value corresponding to the bit contents of the first four bytes occupied by ***bag*** interpreted as a **float**).

Note that the function **put** is overloaded. The compiler can choose the correct version on the basis of the type of the argument; that is why when invoking this function we had to specify this type by appending a suffix 'F' or 'L' to the numerical literals (see the chapter on data types). The parameter in both version is of type **Bag***, so the functions can modify the original object and not only a copy, what would be the case if the argument were passed by value (of course, we could have used references). The function **inf** does not modify the object passed to it, so passing by value would be formally correct, although using pointer can be slightly more efficient. To guarantee that the object will not be changed, the type of the parameter is not **Bag***, but rather **const Bag***

As we can see from the output of the program, the size of a **Bag** object is 16 bytes; this is equal to the size of the longer member (but of course less than the sum of lengths of both members).

More realistic example can be found in the following program:

P107: unions.cpp Anonymous union; type control

```
1. #include <iostream>
2. #include <cassert>
3. using namespace std;
4.
5. struct Bag;
6. enum Kind {NUMBER, POINTER, CHAR};          ❶
7.
8. void put(Bag*,double);
9. void put(Bag*,int*);
```

```
10. void put(Bag*,char);
11.
12. void get(const Bag*,double&);
13. void get(const Bag*,int*&);
14. void get(const Bag*,char&);
15.
16. void info(const Bag&);
17.
18. struct Bag {
19.     Kind kind;
20.     union {                                  ❷
21.         double dbl;
22.         int    *pnt;
23.         char    chr;
24.     };
25. };
26.
27. int main() {
28.     Bag bag;
29.     double x = 3.14, y;
30.     int    i = 10, *pi = &i;
31.     char   c ='a', b;
32.     cout << "sizeof(bag) = " << sizeof(bag)
33.          << " bytes\nAddresses of members:\n dbl: "
34.          << &bag.dbl << "\n pnt: " <<  &bag.pnt
35.          << "\n chr: " << (void*)&bag.chr << endl;
36.
37.     put(&bag,x);
38.     info(bag);
39.     get(&bag,y);
40.     cout << "From function main - y  = " <<    y << endl;
41.
42.     put(&bag,&i);
43.     info(bag);
44.     get(&bag,pi);
45.     cout << "From function main - *pi = " << *pi << endl;
46.
47.     put(&bag,c);
48.     info(bag);
49.     get(&bag,b);
50.     cout << "From function main - b   = " <<   b << endl;
51. }
52.
53. void put(Bag *w, double x) {
54.     w->kind   = NUMBER;                      ❸
55.     w->dbl = x;
56. }
57.
58. void put(Bag *w, int *pi) {
59.     w->kind   = POINTER;
60.     w->pnt = pi;
61. }
62.
63. void put(Bag *w, char c) {
64.     w->kind   = CHAR;
65.     w->chr = c;
66. }
67.
68. void get(const Bag *w, double& x) {
69.     assert(w->kind == NUMBER);
70.     x  = w->dbl;
71. }
72.
73. void get(const Bag *w, int*& pi) {
74.     assert(w->kind == POINTER);
75.     pi = w->pnt;
76. }
77.
78. void get(const Bag *w, char& c) {
79.     assert(w->kind == CHAR);                 ❹
80.     c  = w->chr;
81. }
82.
83. void info(const Bag &w) {
84.     cout << "\nFrom function info - ";
85.     switch (w.kind) {
```

```
86.     case NUMBER:
87.         cout << "Number:    " << w.dbl    << endl;
88.         break;
89.     case POINTER:
90.         cout << "Pointer:   " << *(w.pnt) << endl;
91.         break;
92.     case CHAR:
93.         cout << "Character: " << w.chr    << endl;
94.         break;
95.     }
96. }
```

The structure **Bag** has two fields: one, named ***kind***, of type **Kind** which is an enumeration (❶), and the other, without any name (*sic!*), which is of type of an anonymous union defined locally in the structure (❷). This is an anonymous type: between the keyword **union** and a brace opening the definition there is no name given. Just after the closing brace, we do *not* define any variable of this type. In such situation, a little known feature of anonymous unions shows up: if we define an anonymous union but we do *not* define any variable of this type (right after the definition and before the closing semicolon), then the compiler will create a single instantiation of the union and will move the names of its members into the surrounding scope. Therefore, in our example, the names ***dbl***, ***pnt*** and ***chr*** will be accessible directly in the scope of the structure **Bag**.

The enumeration **Kind** has three values: ***NUMBER***, ***POINTER*** and ***CHAR***. They will be used to indicate the type of data which is currently held in the union member of the object. The function **put** is overloaded. Its version will be chosen by the compiler based on the type of the second argument in invocation: the value of the argument will be assigned to the union member of the object of type **Bag** passed by pointer. At the same time, the functions **put** ensure that when data is written into an appropriate member, information on its type is written into the member ***kind*** of type **Kind** (as on line ❸).

The functions **get** are also overloaded. They are used to get the data from the object of type **Bag**. The data is fetched through the second argument by reference; the type of this argument decides which from the overloaded versions will be chosen (using references here has additional advantage: it will limit a possibilty of unwanted conversions). Each version of **get** checks, using the macro **assert**, if the type requested is consistent with the type of data currently held in the union member of the object (e.g., line ❹):

```
sizeof(bag) = 16 bytes
Addresses of members:
 dbl: 0x7fff6780a7e8
 pnt: 0x7fff6780a7e8
 chr: 0x7fff6780a7e8

From function info - Number:    3.14
From function main - y  = 3.14

From function info - Pointer:   10
From function main - *pi = 10

From function info - Character: a
From function main - b   = a
```

Printing addresses of all members of the union contained in the object ***bag*** at the beginning of the program, one can see that indeed they are all equal – they correspond to the same location in the computer's memory.

T.R. Werner, September 27, 2023; 22:32

14. Classes (I)

In this chapter we introduce the notion of classes: the main building block of the object-oriented programming.

Subsections

- 14.1 Introduction
- 14.2 Accessibility of class members
- 14.3 Fields
- 14.4 Methods
- 14.5 Static member functions
- 14.6 Constructors
- 14.7 Destructors
- 14.8 Creating objects
- 14.9 Arrays of objects
- 14.10 Bit fields

T.R. Werner, September 27, 2023; 22:32

14.1 Introduction

C++, being an object-oriented programming language, provides mechanisms which allow the programmer to define new data types. User defined types can be much richer than simple C-structures which contained just some raw data without the means to operate on them.

Class is a generalization of type; well defined class can behave as a built-in type. In fact, the designers of C++ put much effort to ensure that both built-in and user-defined types could be treated on the same footing, with identical syntax (this is not the case for, e.g., Java). For example, for built-in types, like **int** or **double**, it is possible to define and initialize a variable as if it were an object of a full fledged class with a one-parameter constructor:

```
int k(5);
```

or

```
double *p = new double(6.5);
```

This syntax works, even though types **int** or **double** are not classes and formally there are no constructors for them.

The class essentially consists of fields and functions, collectively called its **members**. However, the class defines also its own name space: inside a class one can define new types, enumerations, **typedef** aliases, etc., and they are accessible from the outside of the class only if their names are qualified by the class name (using the scope-resolution operator ' `::` ').

Classes are usually defined in the global scope, outside other classes or functions, but this is not a requirement of the language: one can define a class even inside a function, although it is seldom useful.

Definition of a class has the form

```
class AClass {
    // ... fields, methods...
};
```

or

```
struct AClass {
    // ... fields, methods...
};
```

One should not forget about a semicolon closing the statement! The new type is already defined at the closing brace; after it, and before the semicolon, one can put definitions of variables of this newly defined type. For example, after

```
struct AClass {
    // ...
} x, y, z;
```

a class named **AClass** is defined and also three objects of type **AClass**, named ***x***, ***y*** and ***z***

are created (if the class **AClass** has a public default constructor, what we will discuss shortly).

Prev: 14. Classes (I) **Up:** 14. Classes (I) **Next:** 14.2 Accessibility of class members

14.2 Accessibility of class members

The members of a class can be accessed by their names in all members of the same class (e.g., in the bodies of member functions) without qualification; some complications only arise when a local name shadows (being identical to) the name of a member. This is a consequence of the fact that classes define their own name spaces.

The situation becomes more complex when a member is to be accessed from the outside of the class, e.g., from a global function or a member function of another class. Each member of a class has an **accessibility level** associated with it. This level is defined by one of three keywords: **public**, **private**, or **protected**..

The definition of a class is divided into **sections**; the accessibility level of a member is determined by the section it was declared in (this is different than in Java, where accessibility has to be specified for each member separately). Each section starts with one of the keywords **public**, **private** or **protected** followed by a colon and ends at the end of class' definition or at the beginning of another section. There may be more than one sections of the same kind. The first section, just at the beginning of the class' definition, does not have to be specified explicitly: its accessibility level will assume the default value (see below). For example:

```
class AClass {
    int s1;
public:
    int s2;
    double d2;
private:
    double s3;
    void fun3(int,double);
public:
    int s4;
    char c4;
};
```

The field ***s1*** will have the default value, according to the following rule:

> **Definitions of classes in C++ can start with the keywords class or struct. The only difference is the default accessibility level: if class has been used, then the default is private; if struct has been used, then the default is public.**

In the example above there are four sections: the first and the third **private** and the second and fourth **public**. Thus, ***s1***, ***s3*** and the function (method) **fun3** are private, while the fields ***s2***, ***d2***, ***s4*** and ***c4*** are public.

Members of a class can be:

- **public**, i.e., accessible from everywhere where the definition of the class is visible (is in scope);
- **private**, i.e., accessible only in this class' scope: in members of the same class (or its

"friends", what will be described later on);

- **protected**, i.e., accessible in this class' scope (and its friends) but also in derived classes' scope; otherwise they behave like **private** (derived classes will be discussed later).

The same rules apply to other names which are declared in the class' scope, like names of types, including enumerations, or aliases defined by **typedef**. They all belong the scope of the class. One can refer to them from the outside of the class by specifying an object of this class (nonstatic members) or by qualifying their names with the name of the class and using the scope resolution operator ' `::` ' (static members, types and aliases).

Limiting the accessibility of members of a class applies to their *names*, not, for example, to the regions of memory they occupy!

Let us consider an example:

P108: greeting.cpp Accessibility of members

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Greeting {
5.      int k1;                                ❶
6.  public:
7.      enum Country { PL, DE, FR };
8.      int k2;                                ❷
9.      void fun(Country country) {
10.         switch (country) {
11.             case PL:
12.                 cout << "Dzien dobry\n"; k1 = 1; break;
13.             case DE:
14.                 cout << "Guten Tag\n";   k1 = 2; break;
15.             case FR:
16.                 cout << "Bonjour\n";     k1 = 3; break;
17.         }
18.     }
19. };
20.
21. int main() {
22.     Greeting dd;                           ❸
23.
24.     dd.fun(Greeting::DE);                  ❹
25.
26.     int *pk1 = &dd.k2 - 1;                 ❺
27.
28.     cout << "sizeof(dd) = " << sizeof(dd) << endl;   ❻
29.     cout << "dd.k1      = " << *pk1       << endl;   ❼
30. }
```

We define a class **Greeting**. The field ***k1*** is private (❶) while ***k2*** (❷) is public, as is the definition of enumeration type **Country** and method (function) **fun**. In the main function we create an object, ***dd***, of this class (❸). Note, that we used the same syntax as when defining, for example, an **int** — just the name of a type and the name of a variable being defined. On line ❹, we invoke the method **fun** on this object (using the name **fun** qualified by the name of an object: the details will be discussed below). The argument has to be of type **Country**, but this type was defined inside the definition of **Greeting**. Thus, we cannot use just the name ***DE*** which is the name of an element of the enumeration — we have to specify in which scope this name is to be searched for. Therefore, the name ***DE*** must be **qualified** with the name of the class and the scope resolution operator ' `::` '. We thus write

Greeting::DE to inform the compiler that what we mean is the name ***DE*** from the scope (name space) of class **Greeting**.

The function **fun**, invoked with ***Greeting::DE*** as the argument, assigned value 2 to the private member ***k1*** of the object ***dd***, on which it was invoked. This was possible, because **fun** is public (so we could have called it) and, being a member of the class, it has access to all other members of the object. But can we access the member ***dd.k1*** from **main** and check that its value is now indeed 2? On line ❻, we print the size of the object ***dd***:

```
Guten Tag
sizeof(dd) = 8
dd.k1 = 2
```

It is 8 bytes; this is not a surprise because the object should contain two integers and nothing more. On line ❺, we take the address of the member ***k2***, what is possible, because it is public. We substract 1 from it, what, according to pointer arithmetic, shoud give us the address of an integer preceding ***dd.k2*** inside the object, i.e., the private member ***dd.k1***. Having the address, we can print the value of this integer (❼): it is 2, as expected.

Thus we see that protecting data by declaring it as **private** does *not* mean that this data is shielded from the "outside world". It only protects the *name* under which data can be accessed. If one finds a way to refer to the data without using a private name, there is no problem with reading or modifying it. Accessibility levels help the programmer to write code which is safer and easier to maintain; they are not meant to be a protection against crackers. If one tries hard enough to crack them, one can always find a way to do so (making the program incomprehensible, dangerous and impossible to maintain).

As we already have said, all member functions of a class — static, nonstatic, constructors, destructor — have unlimited access to members of object of this class. This applies not only to members of the object that a given function was called on; also members of other objects can be accessed if only they belong to the same class (are of the same type). Here is an example:

P109: acc.cpp Access to members of another object of a class

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Vector {
5.      double x, y, z;
6.  public:
7.      void set(double xx, double yy, double zz) {
8.          x = xx;
9.          y = yy;
10.         z = zz;
11.     }
12.     double dot_product(const Vector& w) {
13.         return x*w.x + y*w.y + z*w.z;
14.     }
15. };
16.
17. int main() {
18.     Vector w1, w2;
19.     w1.set(1, 1, 2);
20.     w2.set(1,-1, 2);
21.     cout << "w1*w2 = " << w1.dot_product(w2) << endl;  ❶
22. }
```

Function **dot_product**, being a member of the class, has access to private members ***x***, ***y***, ***z*** of the object ***w1*** on which it was invoked (which are referred to just by their unqualified names), as well as of the object ***w2*** which was passed to it by reference (❶). The program can be compiled and executed without any problem, and prints

```
w1*w2 = 4
```

as expected.

Prev: 14.1 Introduction **Up:** 14. Classes (I) **Next:** 14.3 Fields

14.3 Fields

Fields of a class define data which will be contained in every object of the type specified by this class. They can be of any type, built-in or user-defined (defined in the same program or in a library that the program uses). There is, of course, one exception: nonstatic fields cannot be of the type defined by the class they are member of (we have talked about this when describing structures), but can be pointers to such objects.

Declarations of fields have the form of normal definitions, but often do *not* contain initializations:

```
class AClass {
    int    k1, k2;
    double x,  y;
    // ...
};
```

was and is correct, but

```
class AClass {
    int    k1, k2 = 1;
    double x = 1.5 , y = 3.5; // now OK
    // ...
};
```

caused compilation error till C++11 version, as initialization of a field in its declarations was invalid (there were exceptions, though). However, such initializations *are* possible now. Declaring a nonstatic field means that every object of this type will contain a member of the type and name specified in this declaration. Definition of a class itself does not create any objects — this is just a definition which can, but does not have to, be used later.

Fields can be declared in any order, before, between or after functions; a member function can refer to any field, even if declaration of the function comes lexically before the declaration of the field. However, the order of declarations of fields is sometimes important, so one should remember that when an object is being created, its members will always be created in the order of their declaration in the class' definition and destroyed in the reverse order during the destruction of the object.

What we have said applies to nonstatic fields. **Static fields** fields are declared with the specifier **static**. Static field will correspond to a single variable no matter how many objects of a class we defined or will define. In this respect it is like a global variable, but in the scope of a class (and, perhaps, with limited accessibility). Static variables can exist and can be refered to even if no objects of the class have been created.

The declaration of a static variable does *not* create it. Nonstatic members belong to objects and are created when an object is being created. So when static members declared in a class will be allocated? We have to *define* them (allocate memory for them) *outside* of the class. As they belong to the scope of a class, when being defined outside the class they have to be refererred to by their qualified name, i.e., their name prefixed with the name of a class and the scope resolution operator, ' `::`'. The specifier **static** *cannot* be repeated

in their definition. When defining a static member of a class, one can initialize it, but it is not required. For example,

```
1.      class D {
2.          static int k1;   // declarations
3.          static int k2;
4.      };
5.      int D::k1;       // definition; initialized with zero
6.      int D::k2 = 7;   // definition with initialization
```

Static members ***k1*** and ***k2*** are *declared* in lines 2 and 3, in the definition of the class **D**. Outside the definition, on lines 5 and 6, these variables are *defined*, i.e., memory is allocated for them. Initializer can be omitted; static variables are then initialized automatically with default value zero (of an appropriate type).

Exceptionally, static members can be defined and initialized inside a class, directly at the declaration point, but only if they are constant (**const**) and of an integer type:

```
class D {
    static const int k1, k2 = 2;
    // ...
};
const int D::k1 = 1;
```

Here, ***k2*** is defined and initialized inside the definition of class **D** while ***k1*** is only declared there — the definition in the last line is necessary as well as the initialization; ***k1*** is a constant so must be initialized when being defined.

Static members are physically created after loading the program, but before entering the **main** function (like global variables).

Static members can be refered to in the program by their qualified name (like ***D::k1***) or by name qualified with the name of an object or a pointer to an object (with a dot or an "arrow"). Of course, as there is only one copy of a static member, it does not matter which object is then used.

Unlike nonstatic members, a static member of a class *may* be of type of this class, because physically it will not be contained in objects (only one copy exists). For example, in the following program, the class (structure) **Point** has a static member of this same type: nonstatic members ***x*** and ***y*** represent co-ordinates of a point and may be different for each object, but ***center*** is only one and can represent a point relative to which distances will be calculated:

P110: statmem.cpp Static members

```
1.  #include <iostream>
2.  #include <cmath>    // sqrt
3.  using namespace std;
4.
5.  struct Point {
6.      double x, y;
7.      static Point center;  ❶
8.  };
9.  Point Point::center;      ❷
10.
11. void set_center(double,double);
12. double dist_from_cen(const Point&);
13.
14. int main() {
```

```
15.     Point P = {3, 4};      ❸
16.     cout << "Point P = (" << P.x << "," << P.y << ")\n";
17.
18.     set_center(0,0);
19.     cout << "Dist P-center: " << dist_from_cen(P) << endl;
20.
21.     set_center(9,-4);
22.     cout << "Dist P-center: " << dist_from_cen(P) << endl;
23. }
24.
25. void set_center(double xx,double yy) {
26.     Point::center.x = xx; ❹
27.     Point::center.y = yy; ❺
28.     cout << "Center in (" << xx << "," << yy << ")\n";
29. }
30.
31. double dist_from_cen(const Point& p) {
32.     return
33.         sqrt((p.x-Point::center.x)*(p.x-Point::center.x) +
34.              (p.y-Point::center.y)*(p.y-Point::center.y));
35. }
```

The static member ***center*** is defined on line ❷ and declared on line ❶ (note that the specifier **static** is not repeated in the definition).

An object ***p*** of type **Point** is created and its members ***x*** and ***y*** initialized on line ❸ of the main program. The structure **Point** is not a pure C-structure, but it is an *aggregate* and this form of initialization may be used here (we will tell more about aggregates later).

Using the function **set_center** we set the "center" in $(0, 0)$. The function is not a member of **Point**, so refering to ***center*** we have to qualify its name — lines ❹ and ❺. The variable ***center*** is itself an object of type **Point**, so it has (nonstatic) members ***x*** and ***y***.

The function **dist_from_cen** calculates the distance between a given point passed to it by reference and the current center, represented by the static member of the class:

```
Point P = (3,4)
Center in (0,0)
Dist P-center: 5
Center in (9,-4)
Dist P-center: 10
```

Static fields are often used when one needs some kind of counters, parameters which are common to all objects of a given class (price, current dollar/euro rate, a file descriptor), objects whose value can be used for communication between objects of a class, etc.

14.4 Methods

Fields of a class decribe type of data which will be contained in each object of the class: type of data is common for all objects, but particular values can, of course, be different from object to object. **Methods** define operations that can be performed on objects and data they contain. To define a method, one has to declare and define a nonstatic function as a member of the class.

Declaration of a method is similar to declaration of global functions, but it must be contained inside the definition of a class. The corresponding *definition* of the function may be given outside of the class; what counts is where a function was declared. If we define a method outside of the class, we must qualify its name with the class' name, as this name belongs to the scope of the class it was declared in. There is a difference between declaring *and* defining a method inside the class and declaring it in the class but defining outside of it:

For functions (static, nonstatic, constructors, destructor) declared *and defined* in a class, the modifier inline is assumed by default.

Therefore, the compiler will try to inline such functions; as we remember from section on inline function , this does *not* mean that inlining will actually be performed. Member functions which are declared in a class but defined out of them will not be inlined, unless inlining is explicitly requested in definition (but not in declaration — inlining does not belong to the *contract* between programmer and the user; users do not have to know if a given function is or is not inlined).

As we remember, one can define default arguments of functions — this still applies to class methods. If the definition of a method is outside the class' definition, default arguments must be declared only in the declaration of the method, but, according to general rules, not repeated in its definition (default arguments, of course, *do* belong to the contract). To give an example of defining methods outside the class, we can rewrite the `acc.cpp` program:

P111: accout.cpp Defining methods outside classes

```
1. #include <iostream>
2. using namespace std;
3.
4. class Vector {
5.     double x, y, z;
6.
7. public:
8.     void set(double xx, double yy, double zz);
9.     double dot_product(const Vector& w);
10. };
11.
12. void Vector::set(double xx, double yy, double zz) {
13.     x = xx;
14.     y = yy;
15.     z = zz;
```

```
16. }
17. double Vector::dot_product(const Vector& w) {
18.     return x*w.x + y*w.y + z*w.z;
19. }
20.
21. int main() {
22.     Vector w1, w2;
23.     w1.set(1, 1, 2);
24.     w2.set(1,-1, 2);
25.     cout << "w1*w2 = " << w1.dot_product(w2) << endl;
26. }
```

Note that in definitions of methods **Vector::set** and **Vector::dot_product**, which are now moved out of the class' definition, their names must have been qualified. The function **Vector::set** declares default arguments: in the definition they are *not* repeated.

Methods, i.e., nonstatic member functions which are not a constructor, are always invoked "on" an existing object of the class they are members of. If a method **fun** is called from a function which is not another member of the same class, we have to specify an object it is called on; this can assume two forms:

```
a.fun()
pa->fun()
```

In the first form, ***a*** must be an object of the class of this method or a reference to such object; in the second form, ***pa*** must be a pointer to such object. As long as we disregard polymorphism and inheritance, all these forms ("by pointer", "by reference" and "by object") are equivalent.

One can imagine (quite close to reality), that methods have hidden parameter of type *constant pointer to object of the class*. The pointer to object for which the method is called plays then the rôle of the argument associated with this parameter during invocation (in some languages, e.g., in Python, indexprogramming languages!Python such a parameter is not hidden — it is explicitly specified on the parameter list of methods). Now matter how it is implemented, the pointer to object for which a method has been invoked is accessible inside the method and has the name **this**.

In C++ this is the name of the constant *pointer* to the object that a method was called on; the object itself is then **this*. The keyword this may be used in methods (nonstatic member functions), in constructors and in the destructor of a class.

The pointer **this** cannot be modified: assignement any value to it would be illegal. However, the object it points to, *this* object, can be modified; in particular an assignement to ****this*** is acceptable.

Of course, in a method we can refer to other members of the same object that the method is working on; it is then unnecassary to use ***this*** or ****this*** explicitly: instead of `this->member` or `(*this).member` we can just use `member`. However, it can happen that the name of a local variable, e.g., a variable associated with a parameter of the method, shadows (being the same) the name of a member we want to refer to — in such cases the notation with **this** is required to distinguish between the two. A name without any qualification is then assumed to be the name of a *local* variable of that name, if such a variable exists.

Let us consider an example:

P112: met.cpp Pointer *this* in methods

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Number {
5.      double x;
6.  public:
7.      void    set(double);
8.      Number& add(double);
9.      Number& subtract(double);
10.     Number* multiply(double);
11.     Number* divide(double);
12.     void    info(const char*);
13. };
14. void Number::set(double x) {
15.     this->x = x;                                    ❶
16. }
17. inline Number& Number::add(double x) {
18.     this->x += x;
19.     return *this;
20. }
21. inline Number& Number::subtract(double x) {
22.     this->x -= x;
23.     return *this;
24. }
25. inline Number* Number::multiply(double x) {
26.     this->x *= x;
27.     return  this;
28. }
29. inline Number* Number::divide(double x) {
30.     this->x /= x;
31.     return  this;
32. }
33. void Number::info(const char* s) {
34.     cout << s << " " << x << endl;
35. }
36.
37. int main() {
38.     Number L;
39.
40.     L.set(10);
41.     L.info("set                 :");
42.     L.add(5).subtract(7).info("add + subtract    :"); ❷
43.     L.multiply(2)->divide(4)->                       ❸
44.                     info("multiply + divide :");
45. }
```

All methods of class **Number** are only declared in it; they are defined outside the class (with names qualified with the class' name). Note that in the method **set**, the name ***x*** of the parameter clashes with the name of a member of this class. Therefore, we have to be specific which is meant: on line ❶ `this->x` on the left-hand side refers to the member ***x*** of *this* object, while just ***x*** on the right-hand side refers to a local variable corresponding to the argument.

The functions **add** and **subtract** return by reference the object they were called for (*this* object). The return type is hence **Number&**, and the function returns ****this***. Therefore, the expression `L.add(5)` (❷) is a reference, i.e., another name, of the object ***L*** *after* the method **add** has been executed with argument 5 (setting the value of the member ***x*** of object ***L*** equal to 5). As this is the name of an object of class **Number**, one can call in turn the method **subtract** on the same object, and then, in the same way, the function **info**.

A similar cascade of calls can be performed with the functions **multiply** and **divide**. They, however, return *this* object not by reference, but by pointer (and hence are of type **Number***). Consequently, `L.multiply(2)` in line ❸ is a *pointer* to ***L*** after the method **multiply** has been executed with argument 2: that is why the notation with “arrow” must have been used here. The result

```
set               : 10
add + subtract    : 8
multiply + divide : 4
```

is what we would expect. Note that there is only one object of class **Number** in the whole program: we refer to it by name, by references and by pointers.

Prev: 14.3 Fields **Up:** 14. Classes (I) **Next:** 14.5 Static member functions

14.5 Static member functions

Member functions of a class can be declared **static**. Such functions can be invoked even when there is no objext of the given class. Their names belong to the class' scope, so invokingthem from the outside of the class requires qualified names (with the name of the class and the scope operator ' `::` '). The member selection operator (i.e., a dot) may also be used: only the class of an object is then relevant, not the object itself. As static member functions are *not* invoked for an object, there is no **this** available in their body. For the same reason, static functions cannot directly refer to any nonstatic members of the class. This should be obvious: no particular object is associated with an invocation so it would be impossible to determine the “host” of a member (i.e., a member of which object is referred). It is quite legitimate, however, to refer to other static members of the same class; no object is then needed, because static members “belong to class” and not to an object.

Therefore, static member functions behave much like normal global functions, with an inportant difference: they belong to the scope of the class so they have access to names declared in this class (even if they are **private**).

For example, the `acc.cpp` program can be rewritten to use static member function calculateing the dot product of two vectors:

P113: memstat.cpp Static member functions

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Vector {
5.      double x, y, z;
6.  public:
7.      void set(double xx = 0, double yy = 0, double zz =0) {
8.          x = xx;
9.          y = yy;
10.         z = zz;
11.     }
12.     static double dot_product(const Vector& w1,
13.                               const Vector& w2) {
14.         return w1.x * w2.x + w1.y * w2.y + w1.z * w2.z;
15.     }
16. };
17.
18. int main() {
19.     Vector w1, w2, ww;
20.     w1.set(1, 1, 2);
21.     w2.set(1,-1, 2);
22.
23.     cout << "w1*w2 = "
24.          << Vector::dot_product(w1, w2) << endl;  ❶
25.
26.     cout << "w1*w2 = "
27.          << ww.dot_product(w1, w2)      << endl;  ❷
28. }
```

Note that on line ❶, the name of function is qualified with the name of class **Vector**, but on line ❷ the same function is invoked for object ***ww*** — actually, no information about ***ww*** is passed to the function (it has not even been initialized!). The only purpose it serves is to specify the class in the scope of which to look for the name of the function; any other object of class **Vector** could have been used here with the same effect. All the information about vectors to be multiplied goes through arguments.

Prev: 14.4 Methods **Up:** 14. Classes (I) **Next:** 14.6 Constructors

14.6 Constructors

Generally, every class should have a **constructor**: a method which will be invoked when an object is being created. We do not have to define constuctors ourselves; if we do not, the compiler will provide one — the **default constructor** which will be public, with no parameters and which does nothing.

> **No default constructor will be defined automatically if *any* constructor has been defined by the programmer. A constructor, generated automatically or defined by the programmer, is the default constructor if, and only if, it can be invoked without any arguments.**

The last remark does not mean that the member function defining the default constructor must be parameterless; it can have any numbers of parameters, but, if it has them, they all must have default values (see see section on default arguments).

This means, of course, that there can be only one default constructor in any class. As we have already noted, if it has been generated automatically, its body is empty and it is **public**.

A constructor (not necessarily the default one) is invoked when object is being created (strictly speaking, at the very end of this process). Constructors cannot be called “by hand” for an already existing object.

All constructors must have names identical to the name of the class that they are declared in (names are, of course, case sensitive). They do not return any value. Exceptionally, we do *not* indicate this fact by the keyword **void** — no return value, even **void**, can be specified for a constructors.

As other functions, constructors can have default arguments — actually, default arguments are particularly useful in this case. Many constructors can be declared and defined in the same class, i.e., they can be overloaded. All have the same name — the name of the class. Which one will be selected, depends on number and type of arguments — the rules here are identical to those pertaining other overloaded functions (see section on function overloading).

Note that we can define an object in the global scope. As global variables are created before invoking **main** function, in the order specified by the order of their definitions, constructors of global objects will also be invoked *before* the flow of control enters the **main**. The same applies to variables which are not global, but declared **static** in a class — if such a variable is an object of a class, its constructor will be called before **main** as well.

When constructor is entered, the object it works on is already created, i.e., the memory has been physically allocated and, most importantly, all its members already exist. They are

created in the order of declarations of the corresponding fields in the definition of the class.

As a consequence, it *does* make sense to use the pointer **this** in the body of constructors: the members of *this* object already exist. However, until the constructor exits, the object is considered still in the process of creation and is not accessible from the “outer world” (for example, in multithreaded environment).

Prev: 14.5 Static member functions **Up:** 14. Classes (I) **Next:** 14.7 Destructors

14.7 Destructors

Constructors are functions which are called when an object is being created. However, there is a symmetry in C++ in this respect: “symmetrically” to constructors, there are also **descrtuctors** — functions which are invoked when objects of a class are removed from memory. Destructors do *not* destruct anythig “by themselves”! They are just invoked when objects are being removed. Such a removal of object created on the stack (as a local variable) takes place when the flow of control leaves the innermost block in which the object has been defined (e.g., the body of a function). For example, all variables defined in a function, in particular in the **main**, are removed *after* the function returned; this means that destructors of object created in the **main** will be called after the program has finished its task! Variables on the stack, also of an object type, are removed starting from its top, i.e., variables which were created later will be removed earlier — the order is reverse to the order of creation.

Objects created on the heap, using **new** operator, are destroyed (removed) when the memory that they occupy is released “manually” by using **delete** operator. They have to be removed by the program itself — this means that their destructors will never be invoked automatically, one has to delete the object explicitly (see section on memory management).

We do not have to define a destructor in a class. If it is not defined, the system will provide a default public destructor which does nothing. If, however, we do want to define a destructor, its name must be identical to the name of the enclosing class, prepended by a tilde character, like **~Name**. As for constructors, we do not specify any return value of a destructor, not even **void**. It must always be parameterless, what also means that it *cannot* be overloaded. That it must be so is understandable: destructor is a function which will be called by the system automatically when object is deleted; the system has no way of knowing what arguments we would like to use.

Destructors, unlike constructors, behave like normal methods of classes. In particular, they *can* be invoked manually, by their name. The only difference is that they will be called automatically when objects are destroyed. When invoked, they do exactly what is coded in their definition, no more, no less — in particular they do not have to “destruct” anything. Although calling destructor manually is usually not needed, one can do it if one insists:

```
AClass* object = new AClass();
// ...
object -> ~AClass();
```

This will *not* do any harm to the object; the body of the destructor will just be executed, as it would be for other methods. Note that invoking *constructors* for existing objects is *not* permitted.

14.8 Creating objects

Defining (creating) objects one has to specify which constructor is to be used (as constructors can be overloaded). Like for other overloaded function, the version to be selected will be determined by a parenthesized list of arguments which are passed to the constructor. If the required constructor is the default one, i.e., with no arguments, then the situation is somewhat more complicated: sometimes the parentheses *have to*, sometimes *may*, and sometimes *must not* be omitted!

In the following example (a very important one!), we demonstrate several ways of defining objects, as well as the order of invoking their constructors and destructors.

We define a class **AClass**. It has a constructor with one integer parameter (❷). Therefore, no default (parameterless) constructor will be created automatically. We can define it ourselves, though (❶).

P114: creatob.cpp Creating objects

```
#include <iostream>
using namespace std;

class AClass {
    static char ID;
    int          a;
    char        id;
public:
    AClass() {                                  ❶
        id = ID++;
        a  = 0;
        cout << "Ctor()     " << id << a << endl;
    }

    AClass(int aa) {                            ❷
        id = ID++;
        a  = aa;
        cout << "Ctor(int) " << id << a << endl;
    }

    ~AClass() {                                 ❸
        cout << "Dtor      " << id << a << endl;
    }
};
char AClass::ID = 'A';                          ❹

AClass k1;                          // <- A
//AClass ka();  // WRONG!
//AClass ka{};  //    OK!

int main() {
    cout << "Entering \'main\'" << endl;

    // AClass kb = AClass; // WRONG!
    {
        AClass k3 = AClass();       // <- C
        AClass k4 = AClass(4);      // <- D
    }

```

```
40.     AClass* pk5 = new AClass;         // <- E
41.     AClass* pk6 = new AClass();       // <- F
42.     AClass* pk7 = new AClass(7);      // <- G
43.
44.     delete pk6;
45.     delete pk7;
46.
47.     cout << "Leaving  \'main\'" << endl;
48. }
49.
50. AClass k2(2);                         // <- B
```

The fields of the class are of type **int** (the field ***a***) and of type **char** (***id***). These are nonstatic fields, what means that every object of the class will contain members with these names and types. One field is declared as static: it is ***ID*** of type **char**. As we know, for static fields the declaration is not enough: outside the class we have to define it (❹).

Both constructors contain the statement '`id = ID++;`', which puts into the member ***id*** of the created object the current value of static variable ***ID*** (which, being static, exists in one copy only — it is not a member of objects). After that, the value of ***ID*** is incremented by one, so it will correspond to a character with the next ASCII code (it was initialized on line ❹ by the ASCII code of the letter 'A'). In this way every object of the class will have a unique identifier represented by its character member ***id***. The constructors and the desctructor (❸) all print a message, so we will be able to see in what order the objects are created and destroyed during the execution of the program.

Let us now see how to create objects.

The instruction '`AClass k1;`' on line A creates a global variable ***k1***. The syntax is as if we were creating a **double** or an **int** (e.g., '`int k;`'): first we specify a type (**AClass** in this case) and then a name of an object to be created. No arguments were passed to a constructor, so the default one will be used (it must exist!). When one uses this form of creating objects with default constructor, empty parentheses *must be* omitted! The nest line, which was commented out here, ('`AClass ka();`') would be illegal. This is so because such a line would be interpreted not as a *definition* of a variable ***ka***, but rather as a *declaration* of a function **ka** with no parameters and returning, by value, an object of type **AClass** (according to the standard, if something can be interpreted as a declaration, it is a declaration). However, the form with empty curly braces would be legal (this cannot be interpreted as a declaration.)

On the last line, we create another object, ***k2***, of the same class: we use '`AClass k2(2);`'. What is specified in parentheses is an argument for a constructor. It is an **int**, so the one-parameter constructor will be used. Note that it looks as if ***k2*** were the name of a function (the constructor). However, the presence of type name on the left indicates that this is not the case: the compiler will understand that what we want is to create an object of class **AClass**, give it the name ***k2*** and use a constructor which can be invoked with one **int** as an argument.

Yet another form of object creation is illustrated on line C: '`AClass k3 = AClass();`'. Now it is not the name of an object being created, but rather the name of the class which is used as the name of a function to be invoked (this time it corresponds to the “real” name of a constructor). The name ***k3*** is here the name of the object, not of a reference or a pointer to an object, as it would be in Java (in Java objects never have names; only references to objects can be assigned an identifier). There are no arguments in parentheses, so the default constructor will be used. However, empty parentheses *must* be given in this form;

line, commented out here, '`AClass kb = AClass;`' would be illegal! The same form can be used to create an object with arguments for a constructor: line D defines ***k4*** using the one-parameter constructor.

Three objects of class **AClass** are created on lines E, F and G; this time on the heap, by using **new** operator. As this operator returns the *address* of a newly created object, we assign it to variables of pointer type **AClass***. Names ***pk5***, ***pk6*** and ***pk7*** are names of pointers, *not* of the objects — the objects themselves are anonymous. Note that this time we can (line F), but we do not have to (line E) use parentheses when the default constructor is required.

There are more forms of defining objects in C++. For example, if ***k1*** is the name of an existing object, one could define two more objects like this:

```
AClass k8 = k1;
AClass k9(k1);
```

These forms will be described in section on copy-constructors .

Summarizing, objects of classes can be defined in one of the following ways (we assume that a public default constructor and a constructor accepting one argument of type **int** exist; in the last two cases, a copy-constructor must exist):

```
AClass a;
AClass a(5);

AClass a = AClass();
AClass a = AClass(5);

AClass* pa = new AClass;
AClass* pa = new AClass();
AClass* pa = new AClass(5);

AClass b = a;
AClass b(a);
```

Let us now draw our attention on the order of creation and destruction of objects: we can keep track of it due to the fact that both constructors and the destructor leave traces in the printout of the program (*ctor* and *dtor* are traditional abbreviations of the words *constructor* and *destructor*).

```
Ctor()     A0
Ctor(int)  B2
Entering 'main'
Ctor()     C0
Ctor(int)  D4
Dtor       D4
Dtor       C0
Ctor()     E0
Ctor()     F0
Ctor(int)  G7
Dtor       F0
Dtor       G7
Leaving  'main'
Dtor       B2
Dtor       A0
```

As one can see, global objects are created first, in the order of their definitions: first ***k1*** identified by ***A*** and then ***k2*** identified by ***B*** and with the member ***a*** equal to 2. The definition of ***k2*** appears at the very end of the program, after **main**, but belongs to the global scope so is executed before entering **main** (thus the constructor will also be run before **main** — we can see it from the printout).

In the **main** function we create two more variables: ***k3*** and ***k4*** with identifiers ***C*** and ***D***. They are both defined inside a block delimited by braces. Therefore, when the flow of control leaves the block, the two variables, being local in this block, are destroyed and their destructor is called — in the order reverse to the order of their creation (see lines 4-7 of the output).

The statements after the block create three objects with identifiers ***E***, ***F*** and ***G***. Two of them are then destroyed "by hand" by using **delete** operator and now the **main** function finishes its task, what can be seen from the printout. Now, *after* the program has already returned from the **main** function, the two global variables are destroyed: first that identified by ***B*** which was defined later, and then the one created as the first, i.e., that identified by ***A***.

The object pointed to by ***pk5*** (identified by ***E***) has been created on the heap (using **new** operator) but has not been deleted by **delete**. Therefore, it will *not* be deleted at all; from the output one can see that its destructor has not been invoked.

Not all classes need constructors to initialize objects. Some simple classes admit initialization in the form which we have already described for C-structures (see section on C-structures): by a list of initializers of consecutive members, enclosed in braces (the same form is also used for static arrays — see section on static arrays). This method of initialization can only be used for the so called **aggregates**. These are classes that meet the following conditions:

- there are no user defined constructors or a destructor;
- all members are public;
- if they are derived (inherit) from a class, this base class is also an aggregate;
- they are not polymorphic — none of the methods is virtual; this will also become clear later;
- if there are fields which are themselves of an object type, the classes of these fields are also aggregates.

In the following program

P115: aggreg.cpp Aggregates

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  class A  {
6.  public:
7.      int  ia;
8.      char ca;
9.      void print() {
10.         cout << "A: ia = " << ia << " ca = " << ca << endl;
11.     }
12. };
13.
14. struct B {
15.     A    obA;                          ❶
16.     double x;
17.     void print() {
18.         cout << "B: x = " << x << " ";
19.         obA.print();
20.     }
21. };
22.
```

```
23. int main() {
24.     B b{ {4,'a'}, 7.5 };                ❷
25.     b.print();
26. }
```

both classes, **A** and **B**, are aggregates. In particular, class **B** in an aggregate, although it contains a field of object type — the field ***obA*** is of type **A** (❶), but this class is an aggregate. We define an object of class **B** on line ❷. It is initialized with values given in braces. As the first field of **B** is of type **A**, which itself contains two fields, the firts element of the list of initializers for object ***b*** of type **B** is itself a list which is to be used as the list initializing the member ***obA*** of type **A**. The braces enclosing the sublist may in some situations be elided, but it is better to always write them out explicitly. The program prints

```
B: x = 7.5 A: ia = 4 ca = a
```

As one can see, unlike pure C-structures, aggregates *may* contain methods, as long as these methods are not virtual (polymorphic).

The example below shows that aggregates may be created without explicit initialization of their fields

P116: iniagg.cpp Initializing aggregates

```
1. #include <iostream>
2.
3. class A {
4. public:
5.     int i;
6.     double x;
7. };
8.
9. void pr(const A* p) {
10.     std::cout << p->i << ", " << p->x << '\n';
11. }
12.
13. int main() {
14.     A a1{1, 2.5};
15.     A a2;
16.     A a3{};
17.     A* pa4 = new A{3, 4.5};
18.     A* pa5 = new A{};
19.     pr(&a1); pr(&a2); pr(&a3); pr(pa4); pr(pa5);
20. }
21.
```

From the output of the program

```
1, 2.5
0, 2.07351e-317
0, 0
3, 4.5
0, 0
```

one can see that fields od ***a2*** have not been initialized(***x*** has some random value), but creating ***a3***, we used the (empty) brace-list initializer — now members of primitive types *are* default initialized (0/ **false**/ **nullptr**).
As the example shows, aggregates created on the heap (using **new**) can be initialized in the

same way.

Prev: 14.7 Destructors **Up:** 14. Classes (I) **Next:** 14.9 Arrays of objects

14.9 Arrays of objects

Objects can be grouped into arrays, exactly like **int**'s or **double** s (this is different from Java, where there is no such thing like an array of objects — one can only define arrays of pointers instead).

How can one initialize such arrays?

First, let us assume that the class at hand is an aggregate class. Then an array of objects of this type is also an aggregate and can be initialized by using a list of initializers in braces. As for static arrays, if the number of initializers is less than the number of elements, the rest of them will be initialized with zeros of appropriate types; if no initializers are specified, no initialization takes place and all members of all objects in the array have default values (which, for primitive types, is “undefined”).

In the following program, class **AClass** is an aggregate, so an array ***arr*** defined on line ❶ is an aggregate as well. We initialize it using a list, but specifying values only for the first four elements, so the fifth will be filled with zeros:

P117: agrarr.cpp Arrays of aggregates

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class AClass {
5.  public:
6.      char name[4];
7.      int  age;
8.  };
9.
10. int main() {
11.     AClass arr[5] = {{"Joe",11},{"Sue",17},     ❶
12.                      {"Ian",26},{"Jim",29}};
13.     arr[4].age = 22;                            ❷
14.
15.     for (int i = 0; i < 5; i++)
16.         cout << arr[i].name << ";  "
17.              << arr[i].age  << " years old" << endl;
18. }
```

On line ❷ we assign a value to the member ***age*** of the last, fifth element. The member ***name*** of this object will remain filled with zeros: it will correspond to an empty but otherwise well defined C-string:

```
Joe;  11 years old
Sue;  17 years old
Ian;  26 years old
Jim;  29 years old
;  22 years old
```

Let us turn now to classes which are not aggregates.

One can create arrays of objects of such classes without any initialization. Each element will then be initialized automatically using the default constructor — it follows, that a default constructor has to exist!

Another possibility, only for static arrays, is to specify a list of initializers where anonymous objects are just created “in place” — an arbitrary constructor can then be used. This is illustrated by the program below (❶):

P118: classarr.cpp Arrays of objects

```
 1. #include <iostream>
 2. #include <string>
 3. using namespace std;
 4.
 5. class AClass {
 6.     string name;
 7.     int    age;
 8. public:
 9.     AClass(const string& name = "No Name", int age = 100) {
10.         this->name = name;
11.         this->age = age;
12.         cout << "ctor: " << this->name << endl;
13.     }
14.
15.     int     getAge() { return age; }
16.
17.     string getName() { return name; }
18. };
19.
20. int main() {
21.     AClass ob("Adelaide");
22.
23.     AClass ktab[5] = { AClass("Eleonora", 17), ❶
24.                        AClass("Felicity"),
25.                        AClass("Merrilyn", 26),
26.                        ob                      ❷
27.                      };
28.
29.     for (int i = 0; i < 5; i++)
30.         cout << ktab[i].getName() << "; "
31.              << ktab[i].getAge()  << "y. old" << endl;
32. }
```

The array's size is 5, but we explicitly construct only three elements. The fourth is to be a copy of the previously defined object ***ob*** (❷), and the fifth will be created by the default constructor. Such a constructor must exist, and it does exist, because the only constructor defined in the class has default values for all arguments. If all five elements were initialized, no default constructor would be necessary. The printout is:

```
ctor: Adelaide
ctor: Eleonora
ctor: Felicity
ctor: Merrilyn
ctor: No Name
Eleonora; 17y. old
Felicity; 100y. old
Merrilyn; 26y. old
Adelaide; 100y. old
No Name; 100y. old
```

One more remark about the construct used on line ❷: the fourth element of the array is to be a *copy* of object ***ob***. For this to be possible, a public copy-constructor must be defined in the class. Actually, it *is* there, although we do not see it. Copy-constructors will be

explained in the next chapter.

We can also define arrays of objects dynamically, on the heap, using the **new** operator in the usual way described before. In this case there is no way of passing any information to any constructor: all elements will be initialized by the default constructor, which, of course, must then exist.

Prev: 14.8 Creating objects **Up:** 14. Classes (I) **Next:** 14.10 Bit fields

14.10 Bit fields

One can define **bit fields** as fields of a class. The members if this type can store integer values in a preassigned number of bits. Values are represented in binary coding like "normal" integer-type variables, only the number of bits they occupy is different. Consequently, the range of possible values of a bit field depends on its declared length (called also the *width* of a bit field). For example, declaring a bit field as unsigned and of length 3 bits, one can store eight values, from 0 to 7:

```
  7    6    5    4    3    2    1    0
111  110  101  100  011  010  001  000
```

Bit fields are declared as signed or unsigned **int** s with the width specified after their identifier by a colon and number of bits required: for example

```
struct A {
    unsigned color : 4;
    // ...
};
```

declares a bit field ***color*** of width 4. The corresponding member of objects can be then assigned values from the range $[0, 15]$. What are bit fields good for? Couldn't we just define ***color*** as an **int**. The point is that the compiler can pack several bit fields into one computer word: our ***color*** needs 4 bits only, so several other bit fields can fit in the same computer word. For example, if there are bit fields of widths 4, 7 and 3 declared in the same class, they will all fit in one four-byte integer, thus saving memory needed to store them.

In the following example we define a class describing fonts: information on face, weight and color is stored in bit fields:

P119: bitfields.cpp Bit fields

```
1. #include <iostream>
2. using namespace std;
3.
4. class Font {
5.     unsigned  face : 3;
6.     unsigned weight: 1;
7.     unsigned color : 2;
8. public:
9.     enum Face   { HELVETICA, TIMES, ARIAL,      ❶
10.                  COURIER, BOOKMAN, SYMBOL};
11.     enum Weight { NORMAL, BOLD };
12.     enum Color  { BLACK, RED, GREEN, BLUE };
13.
14.     Font(Face face, Weight weight, Color color) {
15.         this->face   = face;
16.         this->weight = weight;
17.         this->color  = color;
18.     }
19.
20.     void describe() {
21.         cout << "Face # " << face << "; weight # "
22.              << weight << "; color # " << color << endl;
```

```
    }
};

int main() {
    Font title(Font::ARIAL,  Font::BOLD,   Font::RED);
    Font text (Font::TIMES,  Font::NORMAL, Font::BLACK);
    Font symb (Font::SYMBOL, Font::BOLD,   Font::BLUE);
    title.describe();
    text.describe();
    symb.describe();
    cout << "Size of object: " << sizeof(Font) << endl;
}
```

The field selecting the face has three bits and so can store values from 0 to 7. These values (actually, the first six of them) have been given names by defining the enumeration **Face** (❶). In this way the user does not have to remember which number corresponds to which face — he/she can refer to faces just by their names. Similarly, the weight is represented by a bit field of width 1 (one bit, so it has only two possible values: 0 and 1), and color by a bit field of width 2 (four possible values). The printout of the program

```
Face # 2; weight # 1; color # 1
Face # 1; weight # 0; color # 0
Face # 5; weight # 1; color # 3
Size of object: 4
```

shows that one object of class **Font** occupies only four bytes — as an **int**. This is because all three bit fields together only need 6 bits, so they were easily packed into one four-bytes word. How this was done is implementation dependent. Consequently, it does *not* make sense to extract the address of a bit field (using '`&`'); for this reason there are no pointers to members of bit-field type!

Also, bit fields cannot be static members of any class.

The standard library provides the class **bitset** (actually, a class template), which has similar functionality as bit fields and is (or, at least, should be) implemented very efficiently and provides convenient interface.

---

15. Classes (II)

We introduced basic notion of classes in the previous chapter. We have learnt how to define classes and how to create objects of classes. We now proceed to the details which are necessary to use classes effectively and in a safe way. We will describe constant methods, copy-constructors, initialization lists, friend functions, etc.

Subsections

- 15.1 Constant methods
  - 15.1.1 *mutable* fields
- 15.2 Volatile methods
- 15.3 Constructors – further details
  - 15.3.1 Copy constructors
  - 15.3.2 Initialization lists
- 15.4 Friend functions
- 15.5 Nested classes
- 15.6 Pointers to class members

T.R. Werner, September 27, 2023; 22:32

Subsections

- 15.1.1 *mutable* fields

15.1 Constant methods

Methods of classes can be declared as constant. Such declaration informs the compiler that a method will not modify the state of the object that this method has been invoked on, i.e., that no member of the object will be changed by this method. The compiler, of course, checks if the object is really not modified. We declare method as constant by placing the keyword **const** just after the parenthesis closing the list of parameters of the method but *before*

- a brace opening the definition;
- a semicolon ending the declaration.

Usually we only declare methods in the definition of a class, and define it elsewhere. In such a case,

if declaration and definition are separated, the const keyword must be specified in both of them.

Note, that a constant method cannot change the state of the object if was invoked on, but *can* change the state of other accessible objects of the same class.

Let us consider an example

P120: constmet.cpp Constant methods

```
1. #include <iostream>
2. using namespace std;
3.
4. class Point {
5.     double x, y;
6. public:
7.     Point(double x, double y) {
8.         this->x = x;
9.         this->y = y;
10.     }
11.     Point translate(double dx, double dy) const;
12.     void  translate(double dx, double dy);
13. };
14.
15. Point Point::translate(double dx, double dy) const {
16.     cout << "const translate\n";
17.     return Point(x+dx,y+dy);
18. }
19.
20. void Point::translate(double dx, double dy) {
21.     cout << "nonconst translate\n";
```

```
22.     x += dx;
23.     y += dy;
24. }
25.
26. int main() {
27.     const Point p1(1,1);
28.     Point p2(2,2);
29.
30.     p1.translate(3,3);
31.     p2.translate(4,4);
32. }
```

In class **Point**, we declare two methods **translate** (with the same name and the same number and types of arguments!), but the first is **const** and the second is not; this makes their types sufficiently different for such overloading to be valid. We then define two point in the program: ***p1*** which is **const** and ***p2*** which is not. On both of them we invoke **translate** ignoring possible returned value. As we can see from the output

```
const translate
nonconst translate
```

in the first case, when the object was **const**, the compiler selected the veresion which "promises" that it will not modify the object. Without such version the program could not be compiled, as

only methods declared as constant can be invoked for constant objects.

In the second case, when the object is not **const**, compiler will select non-const method as better match. However, without this version the program would compile — constant version would have been selected. Therefore, when we write a method which does not change the object it is invoked on, it is better to write it as **const**; it will be possible to invoke such method on both constant and non-constant objects, while a non-const method can be called *only* on non-const objects, even if it actually does not modify anything.

For obvious reasons, constuctors can never be declared as constant — their task is to initialize, i.e., modify the object they are working on.

15.1.1 *mutable* fields

It sometimes happens that a method should behave as constant, but, for some reasons, e.g., for the sake of efficiency of implementation, it still should be capable of changing some members of the object it has been called for.

This is possible if the corresponding fields of the class have been declared as **mutable**. Such declaration means that this member *can* be modified even by constant methods (i.e., declared as **const**).

To see that such an apparent inconsequence can actually have some sense, let us consider a class which describes, in a very simplified way, customers of a bank:

P121: mutable.cpp *mutable fields*

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  struct FullInfo {
6.      string address;
7.
8.      FullInfo(string name) {
9.          cout << "Fetching address from data base" << endl;
10.         address = "Mr " + name + "\'s address";
11.     }
12. };
13.
14. class Customer {
15.     string  name;
16.     mutable FullInfo *fullInfo;
17. public:
18.     Customer(string n) {
19.         name     = n;
20.         fullInfo = nullptr;
21.     }
22.
23.     string getInfo() const {
24.         return name;
25.     }
26.
27.     string getFullInfo() const {
28.         if (fullInfo == nullptr)                    ❶
29.             fullInfo = new FullInfo(name);
30.         return name + ", " + fullInfo->address;
31.     }
32.
33.     ~Customer() {
34.         delete fullInfo;
35.         cout << "deleting " + name << endl;
36.     }
37. };
38.
39. int main() {
40.     Customer customer("Smith");
41.     cout << customer.getInfo()     << endl;      ❷
42.     cout << customer.getFullInfo() << endl;      ❸
43.     cout << "End of \'main\'\n";
44. }
```

Every object of class **Customer** contains some information on a customer (just his name in the member ***name***). We can imagine that this information is in most cases sufficient. But from time to time, a more detailed information is needed — this additional information is described by the class **FullInfo** (in our simplified example, it is a customer's address). This information, however, can be expensive to obtain, because, e.g., it requires a connection to a remote data base or a complicated authorization procedure.

From the point of view of a client of the class **Customer** this is an implementation detail, of no interest to them. What he/she knows is that any object can be asked for both short and full information about the customer it represents. The method **getInfo**, used on line ❷, provides the short-form information: certainly it should be constant, because it only returns information, without modifying it.

The same applies to the method **getFullInfo**, used on line ❸. However, its implementation is based on the **lazy evaluation** strategy: do not fetch the full information until it is really needed — since for most objects of class **Customer** it will never be requested, it would not make much sense to fetch it for all created objects. Therefore, the method **getFullInfo** checks if this is the first invocation (in which case the member ***fullInfo*** is **nullptr**, line ❶)

and, if so, cretaes an object of type **FullInfo**, what requires establishing a connection with a data base. Of course, next time the same information is requested, the member ***fullInfo*** is not **nullptr** and no action is needed: the full information is already there and can be returned immediately. We see that the method **getFullInfo** should be constant, but at the same time it should be able to modify the member ***fullInfo*** at the first invocation. That is why the field ***fullInfo*** had to be declared as **mutable**. Without this specifier, the program couldn't be even compiled:

```
cpp> g++ -pedantic-errors -Wall mutable.cpp
mutable.cpp: In member function
    `std::string Customer::getFullInfo() const':
mutable.cpp:29: error: assignment of data-member
    `Customer::fullInfo' in read-only structure
```

With the field ***fullInfo*** declared as **mutable**, the program compiles smoothly and gives the output

```
Smith
Fetching address from data base
Smith, Mr Smith's address
End of 'main'
deleting Smith
```

which additionally illustarates the fact that the object created on the stack in the **main** function is deleted (and its destructor is called) after the flow of control has left that function.

Prev: 15. Classes (II) **Up:** 15. Classes (II) **Next:** 15.2 Volatile methods

Subsections

- 15.1.1 *mutable* fields

15.1 Constant methods

Methods of classes can be declared as constant. Such declaration informs the compiler that a method will not modify the state of the object that this method has been invoked on, i.e., that no member of the object will be changed by this method. The compiler, of course, checks if the object is really not modified. We declare method as constant by placing the keyword **const** just after the parenthesis closing the list of parameters of the method but *before*

- a brace opening the definition;
- a semicolon ending the declaration.

Usually we only declare methods in the definition of a class, and define it elsewhere. In such a case,

if declaration and definition are separated, the const keyword must be specified in both of them.

Note, that a constant method cannot change the state of the object if was invoked on, but *can* change the state of other accessible objects of the same class.

Let us consider an example

P120: constmet.cpp Constant methods

```
1. #include <iostream>
2. using namespace std;
3.
4. class Point {
5.     double x, y;
6. public:
7.     Point(double x, double y) {
8.         this->x = x;
9.         this->y = y;
10.     }
11.     Point translate(double dx, double dy) const;
12.     void  translate(double dx, double dy);
13. };
14.
15. Point Point::translate(double dx, double dy) const {
16.     cout << "const translate\n";
17.     return Point(x+dx,y+dy);
18. }
19.
20. void Point::translate(double dx, double dy) {
21.     cout << "nonconst translate\n";
```

```
22.     x += dx;
23.     y += dy;
24. }
25.
26. int main() {
27.     const Point p1(1,1);
28.     Point p2(2,2);
29.
30.     p1.translate(3,3);
31.     p2.translate(4,4);
32. }
```

In class **Point**, we declare two methods **translate** (with the same name and the same number and types of arguments!), but the first is **const** and the second is not; this makes their types sufficiently different for such overloading to be valid. We then define two point in the program: ***p1*** which is **const** and ***p2*** which is not. On both of them we invoke **translate** ignoring possible returned value. As we can see from the output

```
const translate
nonconst translate
```

in the first case, when the object was **const**, the compiler selected the veresion which "promises" that it will not modify the object. Without such version the program could not be compiled, as

only methods declared as constant can be invoked for constant objects.

In the second case, when the object is not **const**, compiler will select non-const method as better match. However, without this version the program would compile — constant version would have been selected. Therefore, when we write a method which does not change the object it is invoked on, it is better to write it as **const**; it will be possible to invoke such method on both constant and non-constant objects, while a non-const method can be called *only* on non-const objects, even if it actually does not modify anything.

For obvious reasons, constuctors can never be declared as constant — their task is to initialize, i.e., modify the object they are working on.

15.1.1 *mutable* fields

It sometimes happens that a method should behave as constant, but, for some reasons, e.g., for the sake of efficiency of implementation, it still should be capable of changing some members of the object it has been called for.

This is possible if the corresponding fields of the class have been declared as **mutable**. Such declaration means that this member *can* be modified even by constant methods (i.e., declared as **const**).

To see that such an apparent inconsequence can actually have some sense, let us consider a class which describes, in a very simplified way, customers of a bank:

P121: mutable.cpp *mutable fields*

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  struct FullInfo {
6.      string address;
7.
8.      FullInfo(string name) {
9.          cout << "Fetching address from data base" << endl;
10.         address = "Mr " + name + "\'s address";
11.     }
12. };
13.
14. class Customer {
15.     string  name;
16.     mutable FullInfo *fullInfo;
17. public:
18.     Customer(string n) {
19.         name     = n;
20.         fullInfo = nullptr;
21.     }
22.
23.     string getInfo() const {
24.         return name;
25.     }
26.
27.     string getFullInfo() const {
28.         if (fullInfo == nullptr)                    ❶
29.             fullInfo = new FullInfo(name);
30.         return name + ", " + fullInfo->address;
31.     }
32.
33.     ~Customer() {
34.         delete fullInfo;
35.         cout << "deleting " + name << endl;
36.     }
37. };
38.
39. int main() {
40.     Customer customer("Smith");
41.     cout << customer.getInfo()     << endl;      ❷
42.     cout << customer.getFullInfo() << endl;      ❸
43.     cout << "End of \'main\'\n";
44. }
```

Every object of class **Customer** contains some information on a customer (just his name in the member ***name***). We can imagine that this information is in most cases sufficient. But from time to time, a more detailed information is needed — this additional information is described by the class **FullInfo** (in our simplified example, it is a customer's address). This information, however, can be expensive to obtain, because, e.g., it requires a connection to a remote data base or a complicated authorization procedure.

From the point of view of a client of the class **Customer** this is an implementation detail, of no interest to them. What he/she knows is that any object can be asked for both short and full information about the customer it represents. The method **getInfo**, used on line ❷, provides the short-form information: certainly it should be constant, because it only returns information, without modifying it.

The same applies to the method **getFullInfo**, used on line ❸. However, its implementation is based on the **lazy evaluation** strategy: do not fetch the full information until it is really needed — since for most objects of class **Customer** it will never be requested, it would not make much sense to fetch it for all created objects. Therefore, the method **getFullInfo** checks if this is the first invocation (in which case the member ***fullInfo*** is **nullptr**, line ❶)

and, if so, cretaes an object of type **FullInfo**, what requires establishing a connection with a data base. Of course, next time the same information is requested, the member ***fullInfo*** is not **nullptr** and no action is needed: the full information is already there and can be returned immediately. We see that the method **getFullInfo** should be constant, but at the same time it should be able to modify the member ***fullInfo*** at the first invocation. That is why the field ***fullInfo*** had to be declared as **mutable**. Without this specifier, the program couldn't be even compiled:

```
cpp> g++ -pedantic-errors -Wall mutable.cpp
mutable.cpp: In member function
    `std::string Customer::getFullInfo() const':
mutable.cpp:29: error: assignment of data-member
    `Customer::fullInfo' in read-only structure
```

With the field ***fullInfo*** declared as **mutable**, the program compiles smoothly and gives the output

```
Smith
Fetching address from data base
Smith, Mr Smith's address
End of 'main'
deleting Smith
```

which additionally illustarates the fact that the object created on the stack in the **main** function is deleted (and its destructor is called) after the flow of control has left that function.

Prev: 15. Classes (II) **Up:** 15. Classes (II) **Next:** 15.2 Volatile methods

15.2 Volatile methods

Methods can also be declared as **volatile**. Such a method is obliged to access (for reading or modifying) members of object every time it follows from the code of the method. This means that many usual optimizations are not permitted: changes of members have to be effectuated immediately (without buffering) and caching cannot be used when reading their current values. Such behaviour of methods is needed in situations when values of members can be accessed asynchronously by another process (in multithreaded environment) or by signal-handling procedures.

The syntax for declaring methods as **volatile** is the same as it was for constant methods, only the keyword **const** should be replaced by **volatile**. Methods can be both **const** and **volatile** at the same time.

Constructors cannot be volatile.

Subsections

- 15.3.1 Copy constructors
- 15.3.2 Initialization lists

15.3 Constructors – further details

Constructors have already been introduced in the preceding chapter . That is, however, not all what one has to know about them. Very important rôle is played by the so called copy-constructors. In particular, they are needed when a class contains fields of pointer types — in such situations, destructors and assignment operator overloading are usually necessary as well. Many constructors contain the so called initialization lists, which are sometimes just very convenient, but often necessary. These notions will be described in this section.

15.3.1 Copy constructors

The copy-constructor creates a new object using as a template another, already existing, object of the same class. The object being created should be in a sense identical to the template object, and a copy-constructor just defines in what sense they are “identical”. The two objects — the one created with a copy-constructor and the one used as a template — should be independent of each other, i.e., a modification of one of them should not affect the other.

In a class **A**, the copy-constructor has signature '`A(const A&)`'. The **const** modifier is not necessary here, what will be explained shortly. Actually, a copy-constructor can have more parameters, but then all the remainig parameters must have default values assigned — what is important is that it *can* be invoked with one argument of type defined by the class it belongs to. It is very important that the argument must be passed *by reference* and not by value

One could think that constuctors of this kind will not necessarily be useful. However, as we will see, most often they *are* used, even if do not realize that. For this reason copy-constructors are always created — if they are not defined by the programmer, they will be provided by the compiler automatically, even if other constructors have been defined.

Copy-constructors are always created — by the programmer or by the compiler automatically.

As we remember, default — i.e., parameterless — constructors are *not* created if there are other constructors explicitly defined. This is why we *had* to define a default constructor in our example (lines 7-10); otherwise the definition '`A a;`' (line 25) would have been illegal, as it requires that a default constructor exists. Both default and copy constructors are

public if they are created automatically — of course they do not have to be **public** if they are created explicitly by the programmer.

Why are copy-constructors so important? Note that copying of objects must always be performed when an object is passed to a function *by value* — a copy must be made and put on the stack. Similarly, a copy will have to be constructed when an object is *returned* from a function by value (and not by reference or pointer). In all these cases, the copy-constructor will be invoked. Let us see that this is a case by considering the following program, which does not do anything useful except for printing information when constructors are invoked:

P122: copycon.cpp Copy-constructor

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class A {
5.      double x;
6.  public:
7.      A(double x = 1) {
8.          this->x = x;
9.          cout << "In default constructor" << endl;
10.     }
11.
12.     A(const A& a) {
13.         x = a.x;
14.         cout << "In copy-constructor" << endl;
15.     }
16. };
17.
18. A fun(A a) {
19.     cout << "In function fun" << endl;
20.     return a;
21. }
22.
23. int main() {
24.     cout << "**1**" << endl;
25.     A a;
26.
27.     cout << "**2**" << endl;
28.     A b = a;                    ❶
29.
30.     cout << "**3**" << endl;
31.     A c(b);                     ❷
32.
33.     cout << "**4**" << endl;    ❸
34.     c = fun(a);                 ❹
35. }
```

The program illustrates also two more ways of creating an object, not mentioned in section on creating objects . The instruction '`A b = a;`' (❶) creates on the stack, i.e., as a local variable, an object ***b*** which is to be a copy of already existing object ***a***. When creating this object, the copy-constructor will actually be invoked! Note that this is *not* an assignment — one can assign to existing variables only, while ***b*** does not exist here: it is just being created. The statement on line ❷ ('`A c(b);`') has the same meaning, but is explicit: it tells the compiler *create an object **c** invoking the copy-constructor and passing **b** as the argument*. In fact there is a small difference: in the first case invocation of the copy-constructor is *implicit* and in the second *explicit*. As we will see, constructors can be declared as explicit only, and in this case implicit invocation will not work.

Let us look at the printout:

```
**1**
In default constructor
**2**
In copy-constructor
**3**
In copy-constructor
**4**
In copy-constructor
In function fun
In copy-constructor
```

Note that after the string '`**4**`' had been printed, i.e., after the statement from line ❸ had been executed, the copy-constructor was invoked two more times, although it seems that no other objects were created in the program. Why then was the copy-constructor called? It was invoked when the program was handling the function call on line ❹:

- to make a copy of the argument ***a***, in order to push this copy onto the stack and associate it with the local variable ***x*** in the function **fun**;
- to make a copy of ***x*** in order to associate it with the result value returned by the function.

Would it be possible not to define any copy-constructor in our class? In this case yes: a copy-constructor would have been created automatically even if other constructors would have been defined (this was the case for default constructor), and such a constructor just copies values of members of the template objects onto members of an object being created. As our class contains just one **double** field, such a mechanical copying would be sufficient. However, there are situations when such mechanical copying is *not* what we want; defining copy-constructors is then necessary, as we will show in the next example.

Before that, let us explain why the argument of a copy-constructor has to be passed by reference and cannot be passed by value. This is quite easy to understand: if it were an object (not a reference), its copy would have to be created and pushed onto the stack. To make a copy, the copy-constructor would have to be invoked and a copy of the argument produced. But in order to do this, the copy-constructor would have to be..., and so on, *ad infinitum*.

Therefore, the copy-constructor must get a reference to an existing object. This, however, can be dangerous, because usually we do not want to modify the object passed as a template, what could happen, as the constructor has access to the original object and not to its copy. That is why we usually declare parameters of copy-constructors as **const** — the compiler will then check if we do not try to modify the original template object in the body of copy-constructor; moreover, it will be then possible to pass, as the argument, the reference to a **const** object.

in the body of the copy-constructor, as for other constructors as well, it is possible to refer to members of the object (which already must exist) and to call other methods of the class.

As we noted, automatically generated copy-constructor copies a template object “field by field”. Note that if a field is of a pointer type, its value is just an address and this is this address which will be copied, not the object or the array which is pointed to!

This is what is known as **shallow copy**. Very often, a shallow copy is *not* what we want. What we want is to copy the object (or array) which is pointed to by a pointer, and not just the value of the pointer itself. Such a copy is called a **deep copy**.

Let us consider a class **Person** with fields describing the age and name of a given person:

```
class Person {
    int    age;
    char* name;
    // ...
};
```

The field ***age*** is of type **int** and does not pose any problem. However, ***name*** is supposed to hold a C-string. Its type is declared as **char*** — a pointer to an array of characters. Let us ask ourselves what the size of such array should be. That we do not know — some names are short and some can be quite long. In principle, we could have set a maximum size once for all

```
class Person {
    int   age;
    char name[20];
    // ...
};
```

but then *every* object of this class would contain an array of 20 characters, although most names are much shorter. We cannot make the array shorter, however, because from time to time very long names do happen.

The problem can be solved by using pointer fields. The object will contain only a pointer of type **char***, i.e., the address of a C-string, while the array (C-string) itself will be stored on the heap and will occupy only as many bytes as it is necessary to hold a given name. The array will be allocated in constructors, when the size of name is known — it will be short for short names and long for long names. A name will be passed to constructor as a pointer to an existing C-string (ending with the character '\0').

Why should one allocate an array? It seems that we could do something like this

```
class Person {
    int    age;
    char* name;
public:
    Person(char* n) {
        name = n;
    }
    // ...
};
```

This could be formally correct, but in most cases will not work. Note that in this way we remember in ***name*** the *address* of an array which was created somewhere else and passed to the constructor. We do not know what happens then to this array: it can be modified, or it can simply be deleted leaving us with an object remembering in its member ***name*** the address of an array which does not exist any more!

It is then better to allocate in the constructor an array, copy the string passed to the constructor to this array and remember its address in a member of the object:

P123: person1.cpp Allocating arrays in constructor

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person1 { // not a very good class...
6. public:
7.     int    age;
8.     char* name;
```

```
 9.     Person1(int w, const char* n) {
10.         age  = w;
11.         name = new char[strlen(n)+1];  ❶
12.         strcpy(name,n);                ❷
13.     }
14. };
15.
16. int main() {
17.     char name[] = "Bill";
18.
19.     Person1 bill(29, name);
20.
21.     name[0] = 'J';                     ❸
22.
23.     cout << "From original: " << name      << endl;
24.     cout << "From object  : " << bill.name << endl;
25. }
```

Two functions from the header `cstring` have been used here:

- **strlen** — returns the length of a C-string pointed to by an argument of type **char***, *excluding* the '\0' character which terminates the string;
- **strcpy** — copies a C-string pointed to by the second argument to an array of characters pointed to by the first argument (note the order!), *including* the '\0' charecter terminating the string; returns the address passed as the first argument.

In the constructor we measure the string ***n*** passed through the argument (❶) and allocate an array of the length by one byte larger than the length of the string (what is necessary, since we must provide space for the terminating '\0' character). Then, on line ❷, we copy the original string to the newly created array. The program prints

```
From original: Jill
From object  : Bill
```

what shows us that indeed the object stores the address of its own copy of the person's name; modification of the original name from `Bill` to `Jill` (line ❸) did not affect the copy the address of which is remembered in the member ***name*** of the object.

Note that logically name belongs to objects describing persons, but physically it does not: it is stored somewhere on the heap while the object itself contains only the address of this name. The object is small: 4 bytes for an integer ***age*** and 4 (or 8) for the *address* ***name*** — and information on the person's name, residing somewhere on the heap, occupies only as much space as is necessary.

We have not defined a copy-constructor in our class, so the compiler will provide one itself. Let us see if it works as might be expected:

P124: person2.cpp Pointer fields and copy-constructor

```
 1. #include <iostream>
 2. #include <cstring>
 3. using namespace std;
 4.
 5. class Person2 { // still not a good class...
 6. public:
 7.     int    age;
 8.     char* name;
 9.     Person2(int w, const char* n) {
10.         age = w;
11.         name = new char[strlen(n)+1];
```

```
12.         strcpy(name,n);
13.     }
14. };
15.
16. int main() {
17.     char name[] = "Bill";
18.
19.     Person2 bill(29, name);
20.     Person2 jill(bill);            // invoking copy-ctor
21.
22.     cout << "After creation: bill " << bill.name << endl;
23.     cout << "                jill " << jill.name << endl;
24.
25.     jill.name[0] = 'J';
26.
27.     cout << "After change  : bill " << bill.name << endl;
28.     cout << "                jill " << jill.name << endl;
29. }
```

We create object ***bill*** with name 'Bill'. In the next line we create another object, ***jill***, passing ***bill*** as the pattern — the automatically created copy-constructor will then be invoked. Both objects have name 'Bill', as can be seen from the first two lines of the output:

```
After creation: bill Bill
                jill Bill
After change  : bill Jill
                jill Jill
```

Then we modify the name of ***jill***. The output shows that this caused a modification of ***bill***'s name as well! Of course, we know why this happened. The copy-constructor copied the *address* which was the value of pointer ***name*** contained in object ***bill*** to the corresponding member of object ***jill*** — the address, but not the string itself. There is only one string with the name allocated on the heap, with both objects containing pointers to this same array of characters!

This is of course not what we would want to happen. Therefore, this is a situation when we have to define our own copy-constructor. It should create a separate array of characters (C-string) for the object being created to ensure that each object has its own name that can be modified independently:

P125: person3.cpp Defining copy-constructor

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person3 {    // a better class
6. public:
7.     int     age;
8.     char* name;
9.     Person3(int w, const char* n) { // ctor
10.         age  = w;
11.         name = new char[strlen(n)+1];
12.         strcpy(name,n);
13.     }
14.     Person3(const Person3& p) {      // copy-ctor
15.         age  = p.age;
16.         name = new char[strlen(p.name)+1];
17.         strcpy(name,p.name);
18.     }
19.     ~Person3() {                     // dtor
20.         delete [] name;
21.     }
22. };
```

```
23.
24. int main() {
25.     char name[] = "Bill";
26.
27.     Person3 bill(29, name);
28.     Person3 jill(bill);      // invoking copy-ctor
29.
30.     cout << "After creation: bill " << bill.name << endl;
31.     cout << "                jill " << jill.name << endl;
32.
33.     jill.name[0] = 'J';
34.
35.     cout << "After change  : bill " << bill.name << endl;
36.     cout << "                jill " << jill.name << endl;
37. }
```

Here we define “normal” constructor, but then also a copy-constructor; it copies the member ***age***, measures the length of ***name*** name in object ***p*** passed to the constructor, allocates space on the heap for a copy of this name, and copies this name to the allocated region of memory. The printout

```
After creation: bill Bill
                jill Bill
After change  : bill Bill
                jill Jill
```

shows that this time the strings pointed to by members ***name*** of the two objects are two *separate* character arrays: a modification of one of them does not affect the other.

We have also added a destructor to our class. This is necessary here: creating an object, using any of the two constructors, we invoke **new** operator which allocates a memory region on the heap. When an object of the class is removed from the stack, all its members are removed, among them the pointer ***name***. The pointer, but not the memory region it points to! To reclaim this memory region one has to use **delete** operator — we should do it in the destructor, since we know that it is invoked automatically when the object is being removed.

Our class is still not perfect: as we will see in the chapter on operator overloading , this class should additionally overload at leat the assignment operator.

15.3.2 Initialization lists

As we have already emphasized, when a constructor starts working, the object should have been physically created and all its members should already exist — they can be then changed by the constructor, but not created.

There is a problem which arises here. How should a constant member (declared with **const** modifier) be initialized? Or a member which is a reference? In both cases initialization should take place exactly when such a member is created — this means that it is too late to do it in the body of a constructor: all members should exist before a constructor is entered. We will have a similar problem with members which are objects of a class without a default constructor: how to tell the compiler which constructor of a member object shoud be used before entering the body of a constructor of an object which contains this member?

In all these cases the compilation will fail if we try to postpone initialization until a constructor takes care of it.

One can try to work around the problem with constant members by declaring, instead of a constant, an enumeration containing one defined symbol (the enumeration itself can be anonymous):

```
class A {
    enum { dim = 10 };
    int tab[dim];
public:
    ...
    void fun() {
        ...
        for (int i = 0; i < dim; ++i) { ... }
        ...
    }
};
```

However, such a constant must be defined directly in the code and will have the same value for all objects.

The correct solution of the problems we have just mentioned are **initalization lists** of constructors.

Initialization list can be specified in the definition of a constructor *before* its body but after the parameter list. It has the form of a comma separated list of member initializations; each member initialization consists of its name followed by parenthesized list of expressions whose values are to be used as arguments passed to a constructor of this member. Very often these are just names of parameters of the constructor in which the initialization list is defined. As we have already mentioned, built-in types can also be treated as classes, in the sense that variables can be created with the syntax corresponding to creation of an object by using the copy-constuctor.

Initialization list appears in the *definition* of a constructor after the parameter list but before the left brace starting the body of this constructor. It is separated from the parameter list by a colon.

Initialization lists cannot be specified in *declarations* of constructors, if they are not, at the same time, their definitions.

This should be quite clear: the list belongs to implementation and is *not* a part of the class's interface ("contract"). Users of a class do have to know if initialization lists have been actually used or not. As we remember, the opposite was true for default arguments for constructors (and functions in general): their existence and values *do* belong to the contract, because users must know them to use the function correctly; therefore, these values had to be specified in the declaration.

All members of an object are always initialized in the order they appear in class's definition.

This is true even if one has used a different order in a initialization list.

One does *not* have to specify all members of an object in a list. All the remaing members, not mentioned there, will be initialized (before entering the body of the constructor) by:

- their default constructors (these constructors must therefore exist!);
- standard initialization for members of built-in types. This ususally consists of creating

a variable, but without assigning it any particular value. From the user's point of view, the member exists but has a random value.

Let us consider an example. Suppose we have a class **A** with one two-parameter constructor defined; as we know, no default constructor will be then automatically added by the system (but a copy-constructor *will*!).

```
class A {
    //...
    A(int x, int y) { ... }
    //...
};
```

Suppose also that objects of class **A** are members of every object of a class **B** (the class **B** has fields of type **A**). As these members have to be created *before* entering a constructor, and there is no way to create them without passing two arguments to their constructor, one *has* to use initialization lists:

```
1.    class B {
2.        A mem1, mem2;
3.        //...
4.        B(A a, int x, int y)
5.            : mem1(a), mem2(x,y)
6.        {   }
7.        //  ...
8.    };
```

Constructor of **B** with an initialization list is defined on lines 4-6 — its body is empty, because everything which is to be done is done by the list initialization. Member ***mem1*** will be initialized by the system-generated copy-constructor of the class **A**, because ***a*** is of type **A**. For the member ***mem2***, on the other hand, the user-defined constructor of **A** taking two integers will be used.

Similar situation occurrs when a field is declared as **const**. Constants must be initialized at creation time, so it is too late to do it in a constructor. There is no other way for initializing constant members than to do it through an initialization list. The same applies to members which are references: as we remember, see section on references , references, as constants, must also be initialized at creation time.

Constant fields are rarely used — normally the usual protection provided by making members **private** is more convenient. References as members of objects are usually not a good idea either: a member is then just another name of something outside the object — this creates an unnecessary and potentially dangerous link between an object and the "outside world". However, members of an object type, as in the example above, are quite common.

In the following example the class **Point** does not have a default constructor. The class has two other constructors: a user-defined constructor which takes two **double** s (coordinates of a point), and a copy-constructor provided by the compiler, which is sufficient in this case, because there are no pointer members. Note that a destructor is not needed either — there is no data logically belonging to objects but allocated on the heap.

P126: triangs.cpp Initialization lists

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      double x, y;
6.
7.      Point(double x, double y)
8.          : x(x), y(y)
9.      { }
10.
11.     void show() const {
12.         cout << "(" <<  x << "," << y << ")";
13.     }
14. };
15.
16. struct Triangle {
17.     Point a, b, c;
18.
19.     Triangle(const Point&,const Point&,const Point&);a  ❶
20.     Triangle(double,double,double,double,double,double);❷
21.
22.     void show() const {
23.         cout << "Triangle ";
24.         a.show(); cout << "-";
25.         b.show(); cout << "-";
26.         c.show(); cout << endl;
27.     }
28. };
29.
30. Triangle::Triangle(const Point &a, const Point &b,
31.                                 const Point &c)
32.     : a(a), b(b), c(c)                                  ❸
33. { }
34.
35. Triangle::Triangle(double x1, double y1, double x2,
36.                 double y2, double x3, double y3)
37.     : a(x1,y1), b(x2,y2), c(x3,y3)                      ❹
38. { }
39.
40. int main() {
41.     Point a1(1,1), b1(2,2), c1(3,3);
42.
43.     Triangle T1(a1,b1,c1);
44.
45.     Triangle T2(11,22,22,33,33,44);
46.
47.     T1.show();
48.     T2.show();
49. }
```

The body of the constructor of class **Point** is empty: everything is done through the initialization list. The expression `x(x)` means “initialize the member ***x*** (this is the ***x*** outside the parentheses) passing to its constructor the value of (local) ***x***, i.e., the value of the argument passed to this constuctor — this is the ***x*** inside the parentheses”. Formally, this is an invocation of the copy-constructor for ***x*** — in our case the member ***x*** is of a built-in type, but as we know such syntax is valid universally.

Note that this constructor of class **Point** does not need an initialization list: there would be no problem with creating members of built-in types, and their initialization could have been taken care of in the body of the constructor.

The situation is different for the class **Triangle** Its three fields, representing the vertices of a triangle, are of type **Point**. Therefore, when an object of class **Triangle** is being created, they have to exist before the body of any constructor is executed. But the class **Point** has no default constructor, so the only possibility is to use initialization list where arguments to constructors of **Point** s can be specified. Consequently, both constructors of **Triangle** have initialization lists. They are declared on lines ❶ and ❷ and defined outside the class. Note

that initialization lists only appear in the definitions, but not in the declarations. The first constructor takes three points and passes them (line ❸) through an initialization list to the automatically synthesised copy-constructor of **Point** s. The second one takes six numbers and passes them, in three pairs, to the constructor of **Point** which takes two numbers (line ❹). The result of the program

```
Triangle (1,1)-(2,2)-(3,3)
Triangle (11,22)-(22,33)-(33,44)
```

has been output with the help of methods **show** in both classes. Note that the method in class **Triangle** uses explicitly the same named method in class **Point**. Both methods are **const** ant, as their task is to display information without modifying the object.

We will see an example with constant and reference members in the next section.

Starting with version C++11,, one can call, from initialization list of one constructor, another constructor of the same class (we then say that one constructor “delegates” its work to the other). In this case, on the initialization list of one constructor we put only one element: the name of the class at hand with arguments for the other constructor. This other constructor will be then executed first (its initialization list and its body); afterwards the flow of control will return to the delegating constuctor. For example, in the program

P127: delegconstr.cpp Delegating constructors

```
1.  #include <iostream>
2.
3.  class Point {
4.      double x, y;
5.  public:
6.      Point(double x, double y) : x(x), y(y) {
7.          std::cerr << "CTOR 1: (double,double)\n";
8.      }
9.
10.     Point(double x) : Point(x,0) {
11.         std::cerr << "CTOR 2: (double)\n";
12.     }
13.     Point() : Point(0) {
14.         std::cerr << "CTOR 3: ()\n";
15.     }
16. };
17.
18. int main() {
19.     std::cerr <<   "Point p1(1,1)\n";
20.     Point p1(1,1);
21.     std::cerr << "\nPoint p2(2)\n";
22.     Point p2(2);
23.     std::cerr << "\nPoint p3\n";
24.     Point p3;
25. }
```

as one can see from its output

```
Point p1(1,1)
CTOR 1: (double,double)

Point p2(2)
CTOR 1: (double,double)
CTOR 2: (double)

Point p3
CTOR 1: (double,double)
```

```
CTOR 2: (double)
CTOR 3: ()
```

when we create the object ***p3***, the default constructor will delegate to the second and this in turn will delegate to the first; flow of control then returns to the second constructor and at the and back to the third (default) one.

Prev: 15.2 Volatile methods **Up:** 15. Classes (II) **Next:** 15.4 Friend functions

Subsections

- 15.3.1 Copy constructors
- 15.3.2 Initialization lists

15.3 Constructors – further details

Constructors have already been introduced in the preceding chapter . That is, however, not all what one has to know about them. Very important rôle is played by the so called copy-constructors. In particular, they are needed when a class contains fields of pointer types — in such situations, destructors and assignment operator overloading are usually necessary as well. Many constructors contain the so called initialization lists, which are sometimes just very convenient, but often necessary. These notions will be described in this section.

15.3.1 Copy constructors

The copy-constructor creates a new object using as a template another, already existing, object of the same class. The object being created should be in a sense identical to the template object, and a copy-constructor just defines in what sense they are “identical”. The two objects — the one created with a copy-constructor and the one used as a template — should be independent of each other, i.e., a modification of one of them should not affect the other.

In a class **A**, the copy-constructor has signature '`A(const A&)`'. The **const** modifier is not necessary here, what will be explained shortly. Actually, a copy-constructor can have more parameters, but then all the remainig parameters must have default values assigned — what is important is that it *can* be invoked with one argument of type defined by the class it belongs to. It is very important that the argument must be passed *by reference* and not by value

One could think that constuctors of this kind will not necessarily be useful. However, as we will see, most often they *are* used, even if do not realize that. For this reason copy-constructors are always created — if they are not defined by the programmer, they will be provided by the compiler automatically, even if other constructors have been defined.

Copy-constructors are always created — by the programmer or by the compiler automatically.

As we remember, default — i.e., parameterless — constructors are *not* created if there are other constructors explicitly defined. This is why we *had* to define a default constructor in our example (lines 7-10); otherwise the definition '`A a;`' (line 25) would have been illegal, as it requires that a default constructor exists. Both default and copy constructors are

public if they are created automatically — of course they do not have to be **public** if they are created explicitly by the programmer.

Why are copy-constructors so important? Note that copying of objects must always be performed when an object is passed to a function *by value* — a copy must be made and put on the stack. Similarly, a copy will have to be constructed when an object is *returned* from a function by value (and not by reference or pointer). In all these cases, the copy-constructor will be invoked. Let us see that this is a case by considering the following program, which does not do anything useful except for printing information when constructors are invoked:

P122: copycon.cpp Copy-constructor

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class A {
5.      double x;
6.  public:
7.      A(double x = 1) {
8.          this->x = x;
9.          cout << "In default constructor" << endl;
10.     }
11.
12.     A(const A& a) {
13.         x = a.x;
14.         cout << "In copy-constructor" << endl;
15.     }
16. };
17.
18. A fun(A a) {
19.     cout << "In function fun" << endl;
20.     return a;
21. }
22.
23. int main() {
24.     cout << "**1**" << endl;
25.     A a;
26.
27.     cout << "**2**" << endl;
28.     A b = a;                    ❶
29.
30.     cout << "**3**" << endl;
31.     A c(b);                     ❷
32.
33.     cout << "**4**" << endl;    ❸
34.     c = fun(a);                 ❹
35. }
```

The program illustrates also two more ways of creating an object, not mentioned in section on creating objects . The instruction '`A b = a;`' (❶) creates on the stack, i.e., as a local variable, an object ***b*** which is to be a copy of already existing object ***a***. When creating this object, the copy-constructor will actually be invoked! Note that this is *not* an assignment — one can assign to existing variables only, while ***b*** does not exist here: it is just being created. The statement on line ❷ ('`A c(b);`') has the same meaning, but is explicit: it tells the compiler *create an object **c** invoking the copy-constructor and passing **b** as the argument*. In fact there is a small difference: in the first case invocation of the copy-constructor is *implicit* and in the second *explicit*. As we will see, constructors can be declared as explicit only, and in this case implicit invocation will not work.

Let us look at the printout:

```
**1**
In default constructor
**2**
In copy-constructor
**3**
In copy-constructor
**4**
In copy-constructor
In function fun
In copy-constructor
```

Note that after the string '`**4**`' had been printed, i.e., after the statement from line ❸ had been executed, the copy-constructor was invoked two more times, although it seems that no other objects were created in the program. Why then was the copy-constructor called? It was invoked when the program was handling the function call on line ❹:

- to make a copy of the argument ***a***, in order to push this copy onto the stack and associate it with the local variable ***x*** in the function **fun**;
- to make a copy of ***x*** in order to associate it with the result value returned by the function.

Would it be possible not to define any copy-constructor in our class? In this case yes: a copy-constructor would have been created automatically even if other constructors would have been defined (this was the case for default constructor), and such a constructor just copies values of members of the template objects onto members of an object being created. As our class contains just one **double** field, such a mechanical copying would be sufficient. However, there are situations when such mechanical copying is *not* what we want; defining copy-constructors is then necessary, as we will show in the next example.

Before that, let us explain why the argument of a copy-constructor has to be passed by reference and cannot be passed by value. This is quite easy to understand: if it were an object (not a reference), its copy would have to be created and pushed onto the stack. To make a copy, the copy-constructor would have to be invoked and a copy of the argument produced. But in order to do this, the copy-constructor would have to be..., and so on, *ad infinitum*.

Therefore, the copy-constructor must get a reference to an existing object. This, however, can be dangerous, because usually we do not want to modify the object passed as a template, what could happen, as the constructor has access to the original object and not to its copy. That is why we usually declare parameters of copy-constructors as **const** — the compiler will then check if we do not try to modify the original template object in the body of copy-constructor; moreover, it will be then possible to pass, as the argument, the reference to a **const** object.

in the body of the copy-constructor, as for other constructors as well, it is possible to refer to members of the object (which already must exist) and to call other methods of the class.

As we noted, automatically generated copy-constructor copies a template object “field by field”. Note that if a field is of a pointer type, its value is just an address and this is this address which will be copied, not the object or the array which is pointed to!

This is what is known as **shallow copy**. Very often, a shallow copy is *not* what we want. What we want is to copy the object (or array) which is pointed to by a pointer, and not just the value of the pointer itself. Such a copy is called a **deep copy**.

Let us consider a class **Person** with fields describing the age and name of a given person:

```
class Person {
    int    age;
    char* name;
    // ...
};
```

The field ***age*** is of type **int** and does not pose any problem. However, ***name*** is supposed to hold a C-string. Its type is declared as **char*** — a pointer to an array of characters. Let us ask ourselves what the size of such array should be. That we do not know — some names are short and some can be quite long. In principle, we could have set a maximum size once for all

```
class Person {
    int   age;
    char name[20];
    // ...
};
```

but then *every* object of this class would contain an array of 20 characters, although most names are much shorter. We cannot make the array shorter, however, because from time to time very long names do happen.

The problem can be solved by using pointer fields. The object will contain only a pointer of type **char***, i.e., the address of a C-string, while the array (C-string) itself will be stored on the heap and will occupy only as many bytes as it is necessary to hold a given name. The array will be allocated in constructors, when the size of name is known — it will be short for short names and long for long names. A name will be passed to constructor as a pointer to an existing C-string (ending with the character '\0').

Why should one allocate an array? It seems that we could do something like this

```
class Person {
    int    age;
    char* name;
public:
    Person(char* n) {
        name = n;
    }
    // ...
};
```

This could be formally correct, but in most cases will not work. Note that in this way we remember in ***name*** the *address* of an array which was created somewhere else and passed to the constructor. We do not know what happens then to this array: it can be modified, or it can simply be deleted leaving us with an object remembering in its member ***name*** the address of an array which does not exist any more!

It is then better to allocate in the constructor an array, copy the string passed to the constructor to this array and remember its address in a member of the object:

P123: person1.cpp Allocating arrays in constructor

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person1 { // not a very good class...
6. public:
7.     int    age;
8.     char* name;
```

```
 9.     Person1(int w, const char* n) {
10.         age  = w;
11.         name = new char[strlen(n)+1];  ❶
12.         strcpy(name,n);                ❷
13.     }
14. };
15.
16. int main() {
17.     char name[] = "Bill";
18.
19.     Person1 bill(29, name);
20.
21.     name[0] = 'J';                     ❸
22.
23.     cout << "From original: " << name      << endl;
24.     cout << "From object  : " << bill.name << endl;
25. }
```

Two functions from the header `cstring` have been used here:

- **strlen** — returns the length of a C-string pointed to by an argument of type **char***, *excluding* the '\0' character which terminates the string;
- **strcpy** — copies a C-string pointed to by the second argument to an array of characters pointed to by the first argument (note the order!), *including* the '\0' charecter terminating the string; returns the address passed as the first argument.

In the constructor we measure the string ***n*** passed through the argument (❶) and allocate an array of the length by one byte larger than the length of the string (what is necessary, since we must provide space for the terminating '\0' character). Then, on line ❷, we copy the original string to the newly created array. The program prints

```
From original: Jill
From object  : Bill
```

what shows us that indeed the object stores the address of its own copy of the person's name; modification of the original name from `Bill` to `Jill` (line ❸) did not affect the copy the address of which is remembered in the member ***name*** of the object.

Note that logically name belongs to objects describing persons, but physically it does not: it is stored somewhere on the heap while the object itself contains only the address of this name. The object is small: 4 bytes for an integer ***age*** and 4 (or 8) for the *address* ***name*** — and information on the person's name, residing somewhere on the heap, occupies only as much space as is necessary.

We have not defined a copy-constructor in our class, so the compiler will provide one itself. Let us see if it works as might be expected:

P124: person2.cpp Pointer fields and copy-constructor

```
 1. #include <iostream>
 2. #include <cstring>
 3. using namespace std;
 4.
 5. class Person2 { // still not a good class...
 6. public:
 7.     int    age;
 8.     char* name;
 9.     Person2(int w, const char* n) {
10.         age = w;
11.         name = new char[strlen(n)+1];
```

```
12.         strcpy(name,n);
13.     }
14. };
15.
16. int main() {
17.     char name[] = "Bill";
18.
19.     Person2 bill(29, name);
20.     Person2 jill(bill);            // invoking copy-ctor
21.
22.     cout << "After creation: bill " << bill.name << endl;
23.     cout << "                jill " << jill.name << endl;
24.
25.     jill.name[0] = 'J';
26.
27.     cout << "After change  : bill " << bill.name << endl;
28.     cout << "                jill " << jill.name << endl;
29. }
```

We create object ***bill*** with name 'Bill'. In the next line we create another object, ***jill***, passing ***bill*** as the pattern — the automatically created copy-constructor will then be invoked. Both objects have name 'Bill', as can be seen from the first two lines of the output:

```
After creation: bill Bill
                jill Bill
After change  : bill Jill
                jill Jill
```

Then we modify the name of ***jill***. The output shows that this caused a modification of ***bill***'s name as well! Of course, we know why this happened. The copy-constructor copied the *address* which was the value of pointer ***name*** contained in object ***bill*** to the corresponding member of object ***jill*** — the address, but not the string itself. There is only one string with the name allocated on the heap, with both objects containing pointers to this same array of characters!

This is of course not what we would want to happen. Therefore, this is a situation when we have to define our own copy-constructor. It should create a separate array of characters (C-string) for the object being created to ensure that each object has its own name that can be modified independently:

P125: person3.cpp Defining copy-constructor

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person3 {    // a better class
6. public:
7.     int     age;
8.     char* name;
9.     Person3(int w, const char* n) { // ctor
10.         age  = w;
11.         name = new char[strlen(n)+1];
12.         strcpy(name,n);
13.     }
14.     Person3(const Person3& p) {      // copy-ctor
15.         age  = p.age;
16.         name = new char[strlen(p.name)+1];
17.         strcpy(name,p.name);
18.     }
19.     ~Person3() {                     // dtor
20.         delete [] name;
21.     }
22. };
```

```
23.
24. int main() {
25.     char name[] = "Bill";
26.
27.     Person3 bill(29, name);
28.     Person3 jill(bill);      // invoking copy-ctor
29.
30.     cout << "After creation: bill " << bill.name << endl;
31.     cout << "                jill " << jill.name << endl;
32.
33.     jill.name[0] = 'J';
34.
35.     cout << "After change  : bill " << bill.name << endl;
36.     cout << "                jill " << jill.name << endl;
37. }
```

Here we define “normal” constructor, but then also a copy-constructor; it copies the member ***age***, measures the length of ***name*** name in object ***p*** passed to the constructor, allocates space on the heap for a copy of this name, and copies this name to the allocated region of memory. The printout

```
After creation: bill Bill
                jill Bill
After change  : bill Bill
                jill Jill
```

shows that this time the strings pointed to by members ***name*** of the two objects are two *separate* character arrays: a modification of one of them does not affect the other.

We have also added a destructor to our class. This is necessary here: creating an object, using any of the two constructors, we invoke **new** operator which allocates a memory region on the heap. When an object of the class is removed from the stack, all its members are removed, among them the pointer ***name***. The pointer, but not the memory region it points to! To reclaim this memory region one has to use **delete** operator — we should do it in the destructor, since we know that it is invoked automatically when the object is being removed.

Our class is still not perfect: as we will see in the chapter on operator overloading , this class should additionally overload at leat the assignment operator.

15.3.2 Initialization lists

As we have already emphasized, when a constructor starts working, the object should have been physically created and all its members should already exist — they can be then changed by the constructor, but not created.

There is a problem which arises here. How should a constant member (declared with **const** modifier) be initialized? Or a member which is a reference? In both cases initialization should take place exactly when such a member is created — this means that it is too late to do it in the body of a constructor: all members should exist before a constructor is entered. We will have a similar problem with members which are objects of a class without a default constructor: how to tell the compiler which constructor of a member object shoud be used before entering the body of a constructor of an object which contains this member?

In all these cases the compilation will fail if we try to postpone initialization until a constructor takes care of it.

One can try to work around the problem with constant members by declaring, instead of a constant, an enumeration containing one defined symbol (the enumeration itself can be anonymous):

```
class A {
    enum { dim = 10 };
    int tab[dim];
public:
    ...
    void fun() {
        ...
        for (int i = 0; i < dim; ++i) { ... }
        ...
    }
};
```

However, such a constant must be defined directly in the code and will have the same value for all objects.

The correct solution of the problems we have just mentioned are **initalization lists** of constructors.

Initialization list can be specified in the definition of a constructor *before* its body but after the parameter list. It has the form of a comma separated list of member initializations; each member initialization consists of its name followed by parenthesized list of expressions whose values are to be used as arguments passed to a constructor of this member. Very often these are just names of parameters of the constructor in which the initialization list is defined. As we have already mentioned, built-in types can also be treated as classes, in the sense that variables can be created with the syntax corresponding to creation of an object by using the copy-constuctor.

Initialization list appears in the *definition* of a constructor after the parameter list but before the left brace starting the body of this constructor. It is separated from the parameter list by a colon.

Initialization lists cannot be specified in *declarations* of constructors, if they are not, at the same time, their definitions.

This should be quite clear: the list belongs to implementation and is *not* a part of the class's interface ("contract"). Users of a class do have to know if initialization lists have been actually used or not. As we remember, the opposite was true for default arguments for constructors (and functions in general): their existence and values *do* belong to the contract, because users must know them to use the function correctly; therefore, these values had to be specified in the declaration.

All members of an object are always initialized in the order they appear in class's definition.

This is true even if one has used a different order in a initialization list.

One does *not* have to specify all members of an object in a list. All the remaing members, not mentioned there, will be initialized (before entering the body of the constructor) by:

- their default constructors (these constructors must therefore exist!);
- standard initialization for members of built-in types. This ususally consists of creating

a variable, but without assigning it any particular value. From the user's point of view, the member exists but has a random value.

Let us consider an example. Suppose we have a class **A** with one two-parameter constructor defined; as we know, no default constructor will be then automatically added by the system (but a copy-constructor *will*!).

```
class A {
    //...
    A(int x, int y) { ... }
    //...
};
```

Suppose also that objects of class **A** are members of every object of a class **B** (the class **B** has fields of type **A**). As these members have to be created *before* entering a constructor, and there is no way to create them without passing two arguments to their constructor, one *has* to use initialization lists:

```
1.    class B {
2.        A mem1, mem2;
3.        //...
4.        B(A a, int x, int y)
5.            : mem1(a), mem2(x,y)
6.        {   }
7.        //  ...
8.    };
```

Constructor of **B** with an initialization list is defined on lines 4-6 — its body is empty, because everything which is to be done is done by the list initialization. Member ***mem1*** will be initialized by the system-generated copy-constructor of the class **A**, because ***a*** is of type **A**. For the member ***mem2***, on the other hand, the user-defined constructor of **A** taking two integers will be used.

Similar situation occurrs when a field is declared as **const**. Constants must be initialized at creation time, so it is too late to do it in a constructor. There is no other way for initializing constant members than to do it through an initialization list. The same applies to members which are references: as we remember, see section on references , references, as constants, must also be initialized at creation time.

Constant fields are rarely used — normally the usual protection provided by making members **private** is more convenient. References as members of objects are usually not a good idea either: a member is then just another name of something outside the object — this creates an unnecessary and potentially dangerous link between an object and the "outside world". However, members of an object type, as in the example above, are quite common.

In the following example the class **Point** does not have a default constructor. The class has two other constructors: a user-defined constructor which takes two **double** s (coordinates of a point), and a copy-constructor provided by the compiler, which is sufficient in this case, because there are no pointer members. Note that a destructor is not needed either — there is no data logically belonging to objects but allocated on the heap.

P126: triangs.cpp Initialization lists

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      double x, y;
6.
7.      Point(double x, double y)
8.          : x(x), y(y)
9.      { }
10.
11.     void show() const {
12.         cout << "(" <<  x << "," << y << ")";
13.     }
14. };
15.
16. struct Triangle {
17.     Point a, b, c;
18.
19.     Triangle(const Point&,const Point&,const Point&);a  ❶
20.     Triangle(double,double,double,double,double,double);❷
21.
22.     void show() const {
23.         cout << "Triangle ";
24.         a.show(); cout << "-";
25.         b.show(); cout << "-";
26.         c.show(); cout << endl;
27.     }
28. };
29.
30. Triangle::Triangle(const Point &a, const Point &b,
31.                                 const Point &c)
32.     : a(a), b(b), c(c)                                  ❸
33. { }
34.
35. Triangle::Triangle(double x1, double y1, double x2,
36.                 double y2, double x3, double y3)
37.     : a(x1,y1), b(x2,y2), c(x3,y3)                      ❹
38. { }
39.
40. int main() {
41.     Point a1(1,1), b1(2,2), c1(3,3);
42.
43.     Triangle T1(a1,b1,c1);
44.
45.     Triangle T2(11,22,22,33,33,44);
46.
47.     T1.show();
48.     T2.show();
49. }
```

The body of the constructor of class **Point** is empty: everything is done through the initialization list. The expression `x(x)` means “initialize the member ***x*** (this is the ***x*** outside the parentheses) passing to its constructor the value of (local) ***x***, i.e., the value of the argument passed to this constuctor — this is the ***x*** inside the parentheses”. Formally, this is an invocation of the copy-constructor for ***x*** — in our case the member ***x*** is of a built-in type, but as we know such syntax is valid universally.

Note that this constructor of class **Point** does not need an initialization list: there would be no problem with creating members of built-in types, and their initialization could have been taken care of in the body of the constructor.

The situation is different for the class **Triangle** Its three fields, representing the vertices of a triangle, are of type **Point**. Therefore, when an object of class **Triangle** is being created, they have to exist before the body of any constructor is executed. But the class **Point** has no default constructor, so the only possibility is to use initialization list where arguments to constructors of **Point** s can be specified. Consequently, both constructors of **Triangle** have initialization lists. They are declared on lines ❶ and ❷ and defined outside the class. Note

that initialization lists only appear in the definitions, but not in the declarations. The first constructor takes three points and passes them (line ❸) through an initialization list to the automatically synthesised copy-constructor of **Point** s. The second one takes six numbers and passes them, in three pairs, to the constructor of **Point** which takes two numbers (line ❹). The result of the program

```
Triangle (1,1)-(2,2)-(3,3)
Triangle (11,22)-(22,33)-(33,44)
```

has been output with the help of methods **show** in both classes. Note that the method in class **Triangle** uses explicitly the same named method in class **Point**. Both methods are **const** ant, as their task is to display information without modifying the object.

We will see an example with constant and reference members in the next section.

Starting with version C++11,, one can call, from initialization list of one constructor, another constructor of the same class (we then say that one constructor “delegates” its work to the other). In this case, on the initialization list of one constructor we put only one element: the name of the class at hand with arguments for the other constructor. This other constructor will be then executed first (its initialization list and its body); afterwards the flow of control will return to the delegating constuctor. For example, in the program

P127: delegconstr.cpp Delegating constructors

```
1.  #include <iostream>
2.
3.  class Point {
4.      double x, y;
5.  public:
6.      Point(double x, double y) : x(x), y(y) {
7.          std::cerr << "CTOR 1: (double,double)\n";
8.      }
9.
10.     Point(double x) : Point(x,0) {
11.         std::cerr << "CTOR 2: (double)\n";
12.     }
13.     Point() : Point(0) {
14.         std::cerr << "CTOR 3: ()\n";
15.     }
16. };
17.
18. int main() {
19.     std::cerr <<   "Point p1(1,1)\n";
20.     Point p1(1,1);
21.     std::cerr << "\nPoint p2(2)\n";
22.     Point p2(2);
23.     std::cerr << "\nPoint p3\n";
24.     Point p3;
25. }
```

as one can see from its output

```
Point p1(1,1)
CTOR 1: (double,double)

Point p2(2)
CTOR 1: (double,double)
CTOR 2: (double)

Point p3
CTOR 1: (double,double)
```

```
CTOR 2: (double)
CTOR 3: ()
```

when we create the object ***p3***, the default constructor will delegate to the second and this in turn will delegate to the first; flow of control then returns to the second constructor and at the and back to the third (default) one.

Prev: 15.2 Volatile methods **Up:** 15. Classes (II) **Next:** 15.4 Friend functions

Subsections

- 15.3.1 Copy constructors
- 15.3.2 Initialization lists

15.3 Constructors – further details

Constructors have already been introduced in the preceding chapter . That is, however, not all what one has to know about them. Very important rôle is played by the so called copy-constructors. In particular, they are needed when a class contains fields of pointer types — in such situations, destructors and assignment operator overloading are usually necessary as well. Many constructors contain the so called initialization lists, which are sometimes just very convenient, but often necessary. These notions will be described in this section.

15.3.1 Copy constructors

The copy-constructor creates a new object using as a template another, already existing, object of the same class. The object being created should be in a sense identical to the template object, and a copy-constructor just defines in what sense they are “identical”. The two objects — the one created with a copy-constructor and the one used as a template — should be independent of each other, i.e., a modification of one of them should not affect the other.

In a class **A**, the copy-constructor has signature '`A(const A&)`'. The **const** modifier is not necessary here, what will be explained shortly. Actually, a copy-constructor can have more parameters, but then all the remainig parameters must have default values assigned — what is important is that it *can* be invoked with one argument of type defined by the class it belongs to. It is very important that the argument must be passed *by reference* and not by value

One could think that constuctors of this kind will not necessarily be useful. However, as we will see, most often they *are* used, even if do not realize that. For this reason copy-constructors are always created — if they are not defined by the programmer, they will be provided by the compiler automatically, even if other constructors have been defined.

Copy-constructors are always created — by the programmer or by the compiler automatically.

As we remember, default — i.e., parameterless — constructors are *not* created if there are other constructors explicitly defined. This is why we *had* to define a default constructor in our example (lines 7-10); otherwise the definition '`A a;`' (line 25) would have been illegal, as it requires that a default constructor exists. Both default and copy constructors are

public if they are created automatically — of course they do not have to be **public** if they are created explicitly by the programmer.

Why are copy-constructors so important? Note that copying of objects must always be performed when an object is passed to a function *by value* — a copy must be made and put on the stack. Similarly, a copy will have to be constructed when an object is *returned* from a function by value (and not by reference or pointer). In all these cases, the copy-constructor will be invoked. Let us see that this is a case by considering the following program, which does not do anything useful except for printing information when constructors are invoked:

P122: copycon.cpp Copy-constructor

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class A {
5.      double x;
6.  public:
7.      A(double x = 1) {
8.          this->x = x;
9.          cout << "In default constructor" << endl;
10.     }
11.
12.     A(const A& a) {
13.         x = a.x;
14.         cout << "In copy-constructor" << endl;
15.     }
16. };
17.
18. A fun(A a) {
19.     cout << "In function fun" << endl;
20.     return a;
21. }
22.
23. int main() {
24.     cout << "**1**" << endl;
25.     A a;
26.
27.     cout << "**2**" << endl;
28.     A b = a;                    ❶
29.
30.     cout << "**3**" << endl;
31.     A c(b);                     ❷
32.
33.     cout << "**4**" << endl;    ❸
34.     c = fun(a);                 ❹
35. }
```

The program illustrates also two more ways of creating an object, not mentioned in section on creating objects . The instruction '`A b = a;`' (❶) creates on the stack, i.e., as a local variable, an object ***b*** which is to be a copy of already existing object ***a***. When creating this object, the copy-constructor will actually be invoked! Note that this is *not* an assignment — one can assign to existing variables only, while ***b*** does not exist here: it is just being created. The statement on line ❷ ('`A c(b);`') has the same meaning, but is explicit: it tells the compiler *create an object **c** invoking the copy-constructor and passing **b** as the argument*. In fact there is a small difference: in the first case invocation of the copy-constructor is *implicit* and in the second *explicit*. As we will see, constructors can be declared as explicit only, and in this case implicit invocation will not work.

Let us look at the printout:

```
**1**
In default constructor
**2**
In copy-constructor
**3**
In copy-constructor
**4**
In copy-constructor
In function fun
In copy-constructor
```

Note that after the string '`**4**`' had been printed, i.e., after the statement from line ❸ had been executed, the copy-constructor was invoked two more times, although it seems that no other objects were created in the program. Why then was the copy-constructor called? It was invoked when the program was handling the function call on line ❹:

- to make a copy of the argument ***a***, in order to push this copy onto the stack and associate it with the local variable ***x*** in the function **fun**;
- to make a copy of ***x*** in order to associate it with the result value returned by the function.

Would it be possible not to define any copy-constructor in our class? In this case yes: a copy-constructor would have been created automatically even if other constructors would have been defined (this was the case for default constructor), and such a constructor just copies values of members of the template objects onto members of an object being created. As our class contains just one **double** field, such a mechanical copying would be sufficient. However, there are situations when such mechanical copying is *not* what we want; defining copy-constructors is then necessary, as we will show in the next example.

Before that, let us explain why the argument of a copy-constructor has to be passed by reference and cannot be passed by value. This is quite easy to understand: if it were an object (not a reference), its copy would have to be created and pushed onto the stack. To make a copy, the copy-constructor would have to be invoked and a copy of the argument produced. But in order to do this, the copy-constructor would have to be..., and so on, *ad infinitum*.

Therefore, the copy-constructor must get a reference to an existing object. This, however, can be dangerous, because usually we do not want to modify the object passed as a template, what could happen, as the constructor has access to the original object and not to its copy. That is why we usually declare parameters of copy-constructors as **const** — the compiler will then check if we do not try to modify the original template object in the body of copy-constructor; moreover, it will be then possible to pass, as the argument, the reference to a **const** object.

in the body of the copy-constructor, as for other constructors as well, it is possible to refer to members of the object (which already must exist) and to call other methods of the class.

As we noted, automatically generated copy-constructor copies a template object “field by field”. Note that if a field is of a pointer type, its value is just an address and this is this address which will be copied, not the object or the array which is pointed to!

This is what is known as **shallow copy**. Very often, a shallow copy is *not* what we want. What we want is to copy the object (or array) which is pointed to by a pointer, and not just the value of the pointer itself. Such a copy is called a **deep copy**.

Let us consider a class **Person** with fields describing the age and name of a given person:

```
class Person {
    int    age;
    char* name;
    // ...
};
```

The field ***age*** is of type **int** and does not pose any problem. However, ***name*** is supposed to hold a C-string. Its type is declared as **char*** — a pointer to an array of characters. Let us ask ourselves what the size of such array should be. That we do not know — some names are short and some can be quite long. In principle, we could have set a maximum size once for all

```
class Person {
    int   age;
    char name[20];
    // ...
};
```

but then *every* object of this class would contain an array of 20 characters, although most names are much shorter. We cannot make the array shorter, however, because from time to time very long names do happen.

The problem can be solved by using pointer fields. The object will contain only a pointer of type **char***, i.e., the address of a C-string, while the array (C-string) itself will be stored on the heap and will occupy only as many bytes as it is necessary to hold a given name. The array will be allocated in constructors, when the size of name is known — it will be short for short names and long for long names. A name will be passed to constructor as a pointer to an existing C-string (ending with the character '\0').

Why should one allocate an array? It seems that we could do something like this

```
class Person {
    int    age;
    char* name;
public:
    Person(char* n) {
        name = n;
    }
    // ...
};
```

This could be formally correct, but in most cases will not work. Note that in this way we remember in ***name*** the *address* of an array which was created somewhere else and passed to the constructor. We do not know what happens then to this array: it can be modified, or it can simply be deleted leaving us with an object remembering in its member ***name*** the address of an array which does not exist any more!

It is then better to allocate in the constructor an array, copy the string passed to the constructor to this array and remember its address in a member of the object:

P123: person1.cpp Allocating arrays in constructor

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person1 { // not a very good class...
6. public:
7.     int    age;
8.     char* name;
```

```
 9.     Person1(int w, const char* n) {
10.         age  = w;
11.         name = new char[strlen(n)+1];  ❶
12.         strcpy(name,n);                ❷
13.     }
14. };
15.
16. int main() {
17.     char name[] = "Bill";
18.
19.     Person1 bill(29, name);
20.
21.     name[0] = 'J';                     ❸
22.
23.     cout << "From original: " << name      << endl;
24.     cout << "From object  : " << bill.name << endl;
25. }
```

Two functions from the header `cstring` have been used here:

- **strlen** — returns the length of a C-string pointed to by an argument of type **char***, *excluding* the '\0' character which terminates the string;
- **strcpy** — copies a C-string pointed to by the second argument to an array of characters pointed to by the first argument (note the order!), *including* the '\0' charecter terminating the string; returns the address passed as the first argument.

In the constructor we measure the string ***n*** passed through the argument (❶) and allocate an array of the length by one byte larger than the length of the string (what is necessary, since we must provide space for the terminating '\0' character). Then, on line ❷, we copy the original string to the newly created array. The program prints

```
From original: Jill
From object  : Bill
```

what shows us that indeed the object stores the address of its own copy of the person's name; modification of the original name from `Bill` to `Jill` (line ❸) did not affect the copy the address of which is remembered in the member ***name*** of the object.

Note that logically name belongs to objects describing persons, but physically it does not: it is stored somewhere on the heap while the object itself contains only the address of this name. The object is small: 4 bytes for an integer ***age*** and 4 (or 8) for the *address* ***name*** — and information on the person's name, residing somewhere on the heap, occupies only as much space as is necessary.

We have not defined a copy-constructor in our class, so the compiler will provide one itself. Let us see if it works as might be expected:

P124: person2.cpp Pointer fields and copy-constructor

```
 1. #include <iostream>
 2. #include <cstring>
 3. using namespace std;
 4.
 5. class Person2 { // still not a good class...
 6. public:
 7.     int    age;
 8.     char* name;
 9.     Person2(int w, const char* n) {
10.         age = w;
11.         name = new char[strlen(n)+1];
```

```
12.         strcpy(name,n);
13.     }
14. };
15.
16. int main() {
17.     char name[] = "Bill";
18.
19.     Person2 bill(29, name);
20.     Person2 jill(bill);            // invoking copy-ctor
21.
22.     cout << "After creation: bill " << bill.name << endl;
23.     cout << "                jill " << jill.name << endl;
24.
25.     jill.name[0] = 'J';
26.
27.     cout << "After change  : bill " << bill.name << endl;
28.     cout << "                jill " << jill.name << endl;
29. }
```

We create object ***bill*** with name 'Bill'. In the next line we create another object, ***jill***, passing ***bill*** as the pattern — the automatically created copy-constructor will then be invoked. Both objects have name 'Bill', as can be seen from the first two lines of the output:

```
After creation: bill Bill
                jill Bill
After change  : bill Jill
                jill Jill
```

Then we modify the name of ***jill***. The output shows that this caused a modification of ***bill***'s name as well! Of course, we know why this happened. The copy-constructor copied the *address* which was the value of pointer ***name*** contained in object ***bill*** to the corresponding member of object ***jill*** — the address, but not the string itself. There is only one string with the name allocated on the heap, with both objects containing pointers to this same array of characters!

This is of course not what we would want to happen. Therefore, this is a situation when we have to define our own copy-constructor. It should create a separate array of characters (C-string) for the object being created to ensure that each object has its own name that can be modified independently:

P125: person3.cpp Defining copy-constructor

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person3 {    // a better class
6. public:
7.     int     age;
8.     char* name;
9.     Person3(int w, const char* n) { // ctor
10.         age  = w;
11.         name = new char[strlen(n)+1];
12.         strcpy(name,n);
13.     }
14.     Person3(const Person3& p) {      // copy-ctor
15.         age  = p.age;
16.         name = new char[strlen(p.name)+1];
17.         strcpy(name,p.name);
18.     }
19.     ~Person3() {                     // dtor
20.         delete [] name;
21.     }
22. };
```

```
23.
24. int main() {
25.     char name[] = "Bill";
26.
27.     Person3 bill(29, name);
28.     Person3 jill(bill);      // invoking copy-ctor
29.
30.     cout << "After creation: bill " << bill.name << endl;
31.     cout << "                jill " << jill.name << endl;
32.
33.     jill.name[0] = 'J';
34.
35.     cout << "After change  : bill " << bill.name << endl;
36.     cout << "                jill " << jill.name << endl;
37. }
```

Here we define “normal” constructor, but then also a copy-constructor; it copies the member ***age***, measures the length of ***name*** name in object ***p*** passed to the constructor, allocates space on the heap for a copy of this name, and copies this name to the allocated region of memory. The printout

```
After creation: bill Bill
                jill Bill
After change  : bill Bill
                jill Jill
```

shows that this time the strings pointed to by members ***name*** of the two objects are two *separate* character arrays: a modification of one of them does not affect the other.

We have also added a destructor to our class. This is necessary here: creating an object, using any of the two constructors, we invoke **new** operator which allocates a memory region on the heap. When an object of the class is removed from the stack, all its members are removed, among them the pointer ***name***. The pointer, but not the memory region it points to! To reclaim this memory region one has to use **delete** operator — we should do it in the destructor, since we know that it is invoked automatically when the object is being removed.

Our class is still not perfect: as we will see in the chapter on operator overloading , this class should additionally overload at leat the assignment operator.

15.3.2 Initialization lists

As we have already emphasized, when a constructor starts working, the object should have been physically created and all its members should already exist — they can be then changed by the constructor, but not created.

There is a problem which arises here. How should a constant member (declared with **const** modifier) be initialized? Or a member which is a reference? In both cases initialization should take place exactly when such a member is created — this means that it is too late to do it in the body of a constructor: all members should exist before a constructor is entered. We will have a similar problem with members which are objects of a class without a default constructor: how to tell the compiler which constructor of a member object shoud be used before entering the body of a constructor of an object which contains this member?

In all these cases the compilation will fail if we try to postpone initialization until a constructor takes care of it.

One can try to work around the problem with constant members by declaring, instead of a constant, an enumeration containing one defined symbol (the enumeration itself can be anonymous):

```
class A {
    enum { dim = 10 };
    int tab[dim];
public:
    ...
    void fun() {
        ...
        for (int i = 0; i < dim; ++i) { ... }
        ...
    }
};
```

However, such a constant must be defined directly in the code and will have the same value for all objects.

The correct solution of the problems we have just mentioned are **initalization lists** of constructors.

Initialization list can be specified in the definition of a constructor *before* its body but after the parameter list. It has the form of a comma separated list of member initializations; each member initialization consists of its name followed by parenthesized list of expressions whose values are to be used as arguments passed to a constructor of this member. Very often these are just names of parameters of the constructor in which the initialization list is defined. As we have already mentioned, built-in types can also be treated as classes, in the sense that variables can be created with the syntax corresponding to creation of an object by using the copy-constuctor.

Initialization list appears in the *definition* of a constructor after the parameter list but before the left brace starting the body of this constructor. It is separated from the parameter list by a colon.

Initialization lists cannot be specified in *declarations* of constructors, if they are not, at the same time, their definitions.

This should be quite clear: the list belongs to implementation and is *not* a part of the class's interface ("contract"). Users of a class do have to know if initialization lists have been actually used or not. As we remember, the opposite was true for default arguments for constructors (and functions in general): their existence and values *do* belong to the contract, because users must know them to use the function correctly; therefore, these values had to be specified in the declaration.

All members of an object are always initialized in the order they appear in class's definition.

This is true even if one has used a different order in a initialization list.

One does *not* have to specify all members of an object in a list. All the remaing members, not mentioned there, will be initialized (before entering the body of the constructor) by:

- their default constructors (these constructors must therefore exist!);
- standard initialization for members of built-in types. This ususally consists of creating

a variable, but without assigning it any particular value. From the user's point of view, the member exists but has a random value.

Let us consider an example. Suppose we have a class **A** with one two-parameter constructor defined; as we know, no default constructor will be then automatically added by the system (but a copy-constructor *will*!).

```
class A {
    //...
    A(int x, int y) { ... }
    //...
};
```

Suppose also that objects of class **A** are members of every object of a class **B** (the class **B** has fields of type **A**). As these members have to be created *before* entering a constructor, and there is no way to create them without passing two arguments to their constructor, one *has* to use initialization lists:

```
1.    class B {
2.        A mem1, mem2;
3.        //...
4.        B(A a, int x, int y)
5.            : mem1(a), mem2(x,y)
6.        {   }
7.        //  ...
8.    };
```

Constructor of **B** with an initialization list is defined on lines 4-6 — its body is empty, because everything which is to be done is done by the list initialization. Member ***mem1*** will be initialized by the system-generated copy-constructor of the class **A**, because ***a*** is of type **A**. For the member ***mem2***, on the other hand, the user-defined constructor of **A** taking two integers will be used.

Similar situation occurrs when a field is declared as **const**. Constants must be initialized at creation time, so it is too late to do it in a constructor. There is no other way for initializing constant members than to do it through an initialization list. The same applies to members which are references: as we remember, see section on references , references, as constants, must also be initialized at creation time.

Constant fields are rarely used — normally the usual protection provided by making members **private** is more convenient. References as members of objects are usually not a good idea either: a member is then just another name of something outside the object — this creates an unnecessary and potentially dangerous link between an object and the "outside world". However, members of an object type, as in the example above, are quite common.

In the following example the class **Point** does not have a default constructor. The class has two other constructors: a user-defined constructor which takes two **double** s (coordinates of a point), and a copy-constructor provided by the compiler, which is sufficient in this case, because there are no pointer members. Note that a destructor is not needed either — there is no data logically belonging to objects but allocated on the heap.

P126: triangs.cpp Initialization lists

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      double x, y;
6.
7.      Point(double x, double y)
8.          : x(x), y(y)
9.      { }
10.
11.     void show() const {
12.         cout << "(" <<  x << "," << y << ")";
13.     }
14. };
15.
16. struct Triangle {
17.     Point a, b, c;
18.
19.     Triangle(const Point&,const Point&,const Point&);a  ❶
20.     Triangle(double,double,double,double,double,double);❷
21.
22.     void show() const {
23.         cout << "Triangle ";
24.         a.show(); cout << "-";
25.         b.show(); cout << "-";
26.         c.show(); cout << endl;
27.     }
28. };
29.
30. Triangle::Triangle(const Point &a, const Point &b,
31.                                 const Point &c)
32.     : a(a), b(b), c(c)                                  ❸
33. { }
34.
35. Triangle::Triangle(double x1, double y1, double x2,
36.                 double y2, double x3, double y3)
37.     : a(x1,y1), b(x2,y2), c(x3,y3)                      ❹
38. { }
39.
40. int main() {
41.     Point a1(1,1), b1(2,2), c1(3,3);
42.
43.     Triangle T1(a1,b1,c1);
44.
45.     Triangle T2(11,22,22,33,33,44);
46.
47.     T1.show();
48.     T2.show();
49. }
```

The body of the constructor of class **Point** is empty: everything is done through the initialization list. The expression `x(x)` means “initialize the member ***x*** (this is the ***x*** outside the parentheses) passing to its constructor the value of (local) ***x***, i.e., the value of the argument passed to this constuctor — this is the ***x*** inside the parentheses”. Formally, this is an invocation of the copy-constructor for ***x*** — in our case the member ***x*** is of a built-in type, but as we know such syntax is valid universally.

Note that this constructor of class **Point** does not need an initialization list: there would be no problem with creating members of built-in types, and their initialization could have been taken care of in the body of the constructor.

The situation is different for the class **Triangle** Its three fields, representing the vertices of a triangle, are of type **Point**. Therefore, when an object of class **Triangle** is being created, they have to exist before the body of any constructor is executed. But the class **Point** has no default constructor, so the only possibility is to use initialization list where arguments to constructors of **Point** s can be specified. Consequently, both constructors of **Triangle** have initialization lists. They are declared on lines ❶ and ❷ and defined outside the class. Note

that initialization lists only appear in the definitions, but not in the declarations. The first constructor takes three points and passes them (line ❸) through an initialization list to the automatically synthesised copy-constructor of **Point** s. The second one takes six numbers and passes them, in three pairs, to the constructor of **Point** which takes two numbers (line ❹). The result of the program

```
Triangle (1,1)-(2,2)-(3,3)
Triangle (11,22)-(22,33)-(33,44)
```

has been output with the help of methods **show** in both classes. Note that the method in class **Triangle** uses explicitly the same named method in class **Point**. Both methods are **const** ant, as their task is to display information without modifying the object.

We will see an example with constant and reference members in the next section.

Starting with version C++11,, one can call, from initialization list of one constructor, another constructor of the same class (we then say that one constructor “delegates” its work to the other). In this case, on the initialization list of one constructor we put only one element: the name of the class at hand with arguments for the other constructor. This other constructor will be then executed first (its initialization list and its body); afterwards the flow of control will return to the delegating constuctor. For example, in the program

P127: delegconstr.cpp Delegating constructors

```
1.  #include <iostream>
2.
3.  class Point {
4.      double x, y;
5.  public:
6.      Point(double x, double y) : x(x), y(y) {
7.          std::cerr << "CTOR 1: (double,double)\n";
8.      }
9.
10.     Point(double x) : Point(x,0) {
11.         std::cerr << "CTOR 2: (double)\n";
12.     }
13.     Point() : Point(0) {
14.         std::cerr << "CTOR 3: ()\n";
15.     }
16. };
17.
18. int main() {
19.     std::cerr <<   "Point p1(1,1)\n";
20.     Point p1(1,1);
21.     std::cerr << "\nPoint p2(2)\n";
22.     Point p2(2);
23.     std::cerr << "\nPoint p3\n";
24.     Point p3;
25. }
```

as one can see from its output

```
Point p1(1,1)
CTOR 1: (double,double)

Point p2(2)
CTOR 1: (double,double)
CTOR 2: (double)

Point p3
CTOR 1: (double,double)
```

```
CTOR 2: (double)
CTOR 3: ()
```

when we create the object ***p3***, the default constructor will delegate to the second and this in turn will delegate to the first; flow of control then returns to the second constructor and at the and back to the third (default) one.

Prev: 15.2 Volatile methods **Up:** 15. Classes (II) **Next:** 15.4 Friend functions

15.4 Friend functions

All members of classes **Point** and **Triangle** in the previous example were public, as we used the keyword **struct** in their definitions. However, data in members of objects usually is, and should be, private. There is a problem connected with this issue. Some functions (or methods of other classes) should be able to operate on members of objects of a given class. Of course, one can provide public methods which give access to private members. But then these public methods could be used by *any* function in the program. It would be better if a class could decide itself which functions that are not themselves members of this class can have access to its members. Such a mechanism is provided by establishing "friendship" relation between a class and functions (global or being members of other classes).

A friend function of a class must be declared inside that class; the declaration has to be preceded with keyword **friend**:

```
class AClass {
    // ...
    friend int fun(double, const AClass&);
    // ...
};
```

The declaration can be placed in any section of the class's definition — **public**, **private** or **protected**.

Functions which are friends of a class are *not* its member functions.

As friends, they have full access to all members of all objects of the class; however, as they are not members themselves, they cannot be invoked on an object. It means, in particular, that friend functions do not have any **this** pointer defined. On the other hand, a function can be a friend of several classes: declaration of friendship must be then included in the definitions of all these classes.

Friendship can only be granted: by a class to a function. It can never be "demanded" from a class by a function. There is no way in which one could make his/her function a friend of a class if that class does not explicitly declare this friendship; in particular it is impossible to make our function a friend of a class from a library that we cannot modify and recompile.

A declaration of friendship is contained inside a class, so it falls into that class's scope. Therefore, the declaration should be repeated (without keyword **friend**) in the global scope if the function is to be used there and it has not been defined yet (because, e.g., its definition is located in another module of the program). However, in search for a function declaration, compiler includes the scopes of object-type arguments. Therefore, if a friend function has a parameter of the type of a class that it is a friend of (as is usually the case), the scope of this class will also be searched and the declaration *will* be found (this is the so called **Koenig lookup**, or ADL – *argument-dependent name lookup*. No declaration of this

friend functions outside the scope of the class will then be necessary.

A class can declare *all* member functions from another class to be its friends. For eample, after

```
class B;

class A {
    // ...
    friend class B;
    //...
};
```

all methods of **B** will have access to all members of class **A**, but not *vice versa*. We say that friendship is not a reflexive relation: the fact that **B** is a friend of **A** does not mean that **A** is a friend of **B**. Note that in the above example the forward declaration of **B** is necessary if its full declaration/definition is located after that of **A**.

We can establish mutual friendship of two classes by declaring it in both of them:

```
class A;

class B {
    // ...
    friend class A;
    // ...
};

class A {
    // ...
    friend class B;
    // ...
};
```

Friendship is not a transitive relation either. If **B** is a friend of **A** and **C** and a friend of **B** then **C** does not have to be a friend of **A**.

Finally, friendship is not inherited: derived classes do not inherit friends of their base classes.

In the following example we define two classes: **Point** describes points on the real axis (with the coordinate ***numb***) and **Range** represents ranges [***left***, ***right***]. Fields of both classes are, as they should, private.

P128: isinside.cpp Function which is a friend of two classes

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Range;                                          ❶
5.
6.  class Point {
7.
8.      int numb;
9.      friend void isInside(const Point*, const Range*);  ❷
10.
11. public:
12.     Point(int numb = 0)
13.         : numb(numb)
```

```
14.     { }
15. };
16.
17. class Range {
18.
19.     int left, right;
20.     friend void isInside(const Point*, const Range*);
21.
22. public:
23.     Range(int left = 0, int right = 0)
24.         : left(left), right(right)
25.     { }
26. };
27.
28. void isInside(const Point *p, const Range *z) {
29.     if ((p->numb >= z->left) && (p->numb <= z->right))
30.         cout << "Point " << p->numb  << " is in "
31.                 "range [" << z->left << ","
32.             << z->right  << "]\n";
33.     else
34.         cout << "Point " << p->numb  << " is out of "
35.                 "range [" << z->left << ","
36.             << z->right  << "]\n";
37. }
38.
39. int main() {
40.     Point p(7);
41.     Range z1(0,10), z2(8,20);
42.
43.     isInside(&p,&z1);
44.     isInside(&p,&z2);
45. }
```

Function **isInside** is a friend of both classes. Its task is to tell if a given point lies inside a given range or not. In order to be able to check it, the function needs access to members of both classes, which it has being their friend:

```
Point 7 is in range [0,10]
Point 7 is out of range [8,20]
```

Note that the forward declaration on line ❶ was necessary, as the name **Range** is used in definition of **Point** (line ❷).

Friend function are extensively used when overloading operators (see chapter on overloading).

Friend functions are also used as the so called object factories — they can play rôle of a constructor. Constructors are usually public, but sometimes they are made private and their functionality is provided by friend functions which produce and return an object of an appropriate type, depending on the context.

Let us now consider an example of a class with fields of reference, constant and object types:

P129: confield.cpp Members of reference, constant and object types

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. // Note: incomplete classes - assignment
```

```
6.  // operator should be overloaded here
7.
8.  class Employee;                                          ❶
9.
10. enum position {plain, boss, CEO};                        ❷
11.
12. class Person {
13.     char* name;
14.     int   birth_year;
15.
16.       // declaration of friendship
17.     friend void emplinfo(const Employee*);
18. public:
19.     Person(char* n, int r)
20.         : name(strcpy(new char[strlen(n)+1],n)),
21.           birth_year(r)
22.     { }
23.
24.       // copy-ctor
25.     Person(const Person& p)
26.         : name(strcpy(new
27.               char[strlen(p.name)+1],p.name)),
28.           birth_year(p.birth_year)
29.     { }
30.
31.       // dtor
32.     ~Person() {
33.         cout << "Deleting person " << name << endl;
34.         delete [] name;
35.     }
36. };
37.
38. class Employee {
39.     static int     ID;
40.     Person       pers;
41.     const int &income;
42.     const int      id;
43.
44.       // declaration of friendship
45.     friend void emplinfo(const Employee*);
46. public:
47.     Employee(char* n, int r, int& sal)
48.         : pers(n,r), income(sal), id(++ID)
49.     { }
50.
51.       // copy-ctor
52.     Employee(const Employee& e)
53.         : pers(e.pers), income(e.income), id(++ID)
54.     { }
55. };
56. int Employee::ID;
57.
58. void emplinfo(const Employee* empl) {
59.     cout << empl->pers.name          << " (born "
60.          << empl->pers.birth_year    << ") id="
61.          << empl->id << "; income: " << empl->income
62.                                      << endl;
63. }
64.
65. int main() {
66.     int salary[] = { 1600, 2100, 8900 };
67.
68.     Employee johny(  "Johny   ", 1978, salary[plain]);
69.     Employee billy(  "Billy   ", 1980, salary[plain]);
70.     Employee henry(  "Henry   ", 1965, salary[boss]);
71.     Employee MrHenry("Mr Henry", 1955, salary[CEO]);
72.
73.     emplinfo(&johny);
74.     emplinfo(&billy);
75.     emplinfo(&henry);
76.     emplinfo(&MrHenry);
77.
78.     cout << "\nChanging salaries\n\n";
79.
80.     salary[plain] -=  300;                               ❸
81.     salary[CEO] += 1000;                                 ❹
```

```
82.
83.     emplinfo(&johny);
84.     emplinfo(&billy);
85.     emplinfo(&henry);
86.     emplinfo(&MrHenry);
87.
88.     cout << "\nEnd of program\n\n";
89. }
```

We define an enumeration **position** on line ❷. Next, we define two classes: **Person** and **Employee**. Class **Person** is quite standard. We used initialization lists here, but it was not necessary — “normal” constructors would work equally well, although perhaps somewhat less efficiently.

In the definition of **Person** we declare that function **emplinfo** will be a friend of this class. As the parameter of the function is of type **const Employee***, and this class has not been defined yet, the forward declaration on line ❶ is required.

Class **Employee** is more sophisticated. It has one static, one object, one constant and one reference field. Except for the static member (which exists in one copy and is not physically contained in objects of the class), none of the remaining members can be initialized inside a constructor; therefore, one has to use initialization lists in every constructor of the class, including the copy-constructor.

Function **emplinfo**, which is a friend of **Person**, is a friend of **Employee** as well. It should be, because its task is to print information on an employee — being a friend it has access to private members of objects of class **Employee**. The function does not modify these objects, that is why the type of the pointer parameter was declared as **const**. Note that one of the member of **Employee** is of type **Person**, whose fields are also private. To make them accessible for the function, this class too had to declare friendship with **emplinfo**.

In **main**, we create four objects of class **Employee**. Note that we have to pass arguments which will be used to initialize the subobject of type **Person** in the object of type **Employee** — this is necessary, because **Person** does not contain a default constructor.

The reference member ***income*** of the created object will be initialized with a reference to one of elements of array ***salary***. Consequently, in the scope of an object, the name ***income*** will be just another name of an element of this array. Elements of ***salary*** are indexed with enumeration values which are converted to integers 0, 1 and 2; enumeration is very convenient here, because it protects us against using a wrong value of the index. It also allows us to use sensible and easy to remember names instead of numbers.

After creating the objects, we print information on them:

```
Johny    (born 1978) id=1; income: 1600
Billy    (born 1980) id=2; income: 1600
Henry    (born 1965) id=3; income: 2100
Mr Henry (born 1955) id=4; income: 8900

Changing salaries

Johny    (born 1978) id=1; income: 1300
Billy    (born 1980) id=2; income: 1300
Henry    (born 1965) id=3; income: 2100
Mr Henry (born 1955) id=4; income: 9900

End of program

Deleting person Mr Henry
```

```
Deleting person Henry
Deleting person Billy
Deleting person Johny
```

On lines ❸ and ❹ we modify values of elements of array ***salary*** and print information on the objects again. As we can see, their members ***income*** have been modified. But these members are “doubly” protected: they are **private** and **const**. So how was it possible to change them from **main**? This is quite clear: the members are just other names of elements of an array which exists in **main** and is perfectly modifiable here. Once more we see that in C++ these are the *names* which are protected by **const** or **private**, not physical contents of variables. If we can access a variable through an unprotected name, we are free to modify it.

The output demonstrates automatic invocation of destructors of **Person** s. Note that **Employee** does not need any destructor, as it does not allocate memory or other resources outside the object. However, objects of this class contain subobjects of class **Person** and this class has a destructor. We do not have to worry about it: when an **Employee** is deleted from the stack, destructors of its member objects will be invoked automatically.

Prev: 15.3 Constructors – further details **Up:** 15. Classes (II) **Next:** 15.5 Nested classes

15.5 Nested classes

It is possible to declare a class within the scope of another class. Such a class is called a **nested class** and the class in which it is declared is its **hosts class** or **enclosing class**. The name of a nested class belongs to the scope of its host, so to be referred to from another scope it has to be qualified with the host's name and scope resolution operator ('::'). Other than that, a nested class and its host are not connected in any particular way. Unlike it is in Java, nested classes in C++ do not have any special privileges when accessing members of host's objects, and *vice versa* — in this respect they behave as other classes. However, as their scope lies within the scope of the host, aliases of type names defined with **typedef** or enumeration types from the host are visible in its nested classes without qualification.

Let us look at an example:

P130: nestcl.cpp Nested classes

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Customer {
6.     static int ID;
7.     char*  name;
8.     const  int id;
9.     int    cr,db;
10.
11. public:
12.     class Balance {
13.         int id;
14.         int balance;
15.     public:
16.         Balance(int id, int balance)
17.             : id(id), balance(balance)
18.         {}
19.
20.         void printinfo();
21.     };
22.
23.     Customer(const char* n)
24.         : name(strcpy(new char[strlen(n)+1],n)),
25.           id(++ID), cr(0), db(0)
26.     {}
27.
28.     Customer& credit(int w) { cr += w; return *this; }
29.     Customer&  debit(int w) { db += w; return *this; }
30.
31.     Balance* getBalance();
32.
33.     ~Customer() { delete [] name; }
34. };
35. int Customer::ID = 0;
36.
37. Customer::Balance* Customer::getBalance() {
```

```
38.     return new Balance(id, cr - db);
39. }
40.
41. void Customer::Balance::printinfo() {
42.     cout << "id: " << id << " Balance: " << balance << endl;
43. }
44.
45. int main() {
46.     Customer  joh("Johnson");
47.     Customer* pshe = new Customer("Sheldon");
48.
49.     joh.credit(100).credit(50).debit(75);
50.     pshe->credit(200).debit(50).debit(25);
51.
52.     Customer::Balance* psjoh =   joh.getBalance();
53.     Customer::Balance* psshe = pshe->getBalance();
54.
55.     psjoh->printinfo();
56.     psshe->printinfo();
57.
58.     Customer::Balance anonim(9,500);
59.     anonim.printinfo();
60.
61.     delete psjoh;
62.     delete psshe;
63.     delete pshe;
64. }
```

In the scope of class **Customer** we declare a nested class **Balance**. Its method **printinfo** is declared but defined outside the class (line 41). Note that to define it there, we had to use double qualification of its name: this is function **printinfo** from the scope of **Balance**, which in turn is in the scope of **Customer**.

Two pointers of type **Balance*** are created on lines 52 and 53. As previously, we had to qualify the name this type by the name of the enclosing class **Customer**.

Note that it is *not* true that it is impossible to create objects of type **Balance** outside the scope od **Customer**. As we can see from line 58 and from the output

```
id: 1 Balance: 75
id: 2 Balance: 125
id: 9 Balance: 500
```

we can create objects of class **Balance** without referencing any objects or methods of its enclosing class (this is different than, e.g., in Java).

Nested classes are rarely used, as the are never absolutely necessary, although sometimes they may be convenient.

Concluding this subject let us mention that classes may be declared/defined in the scope of a function (**local classes**). In such a case it is not visible anywhere except the function it is declared in. Local classes are used even less frequently than nested classes.

15.6 Pointers to class members

Objects of a class have always the same size and the same structure. In particular, corresponding data members (fields) start at the same position relative to the beginning of the whole object, i.e., the shift between a member and the beginning of the object is always the same. This allows us to define a special kind of pointers, the so called **pointers to class members**. As we know, the value of a “normal” pointer is the address of an object in memory. For pointers to class members, this value is not an absolute address, but rather the shift between a member and the beginning of the object it belongs to. Knowing the absolute address of an object and the shift, the system is able to calculate the absolute address of the member.

A pointer to member, which is itself of type **Type**, of a class **AClass** can be declared as

```
Type AClass::*pointer;
```

what means that the value of ***pointer*** will be the shift between the beginning of any object of class **AClass** and its public, nonstatic data member (field) of type **Type**. As we can see, the difference between this definition and a definition of a “normal” pointer is that here the class-scope operator ('`AClass::`' in our case) must have been used.

A pointer declared in this way does not have any specific value yet, it does not point to anything useful.

Suppose now that our class **AClass** has, among others, two public, nonstatic fields, ***memb1*** and ***memb2***, both of type **Type**. We can now assign a value to our pointer:

```
pointer = &AClass::memb1;
```

or

```
pointer = &AClass::memb2;
```

and this means that now ***pointer*** has a value: it is the shift between the beginning of *any* object of class **AClass** and its public, nonstatic member ***memb1*** (or ***memb2***). This shift serves no useful purpose if we do not know what object is meant, i.e., relative to what the shifting is to be performed. After this assignment, for any specific, existing object ***ob*** of class **AClass**, we can refer to its member ***memb1*** using the operator '`.*`':

```
ob.*pointer
```

Variable ***ob*** indicates which object is meant and ***pointer*** gives the shift which specifies where, within the object, the member ***memb1*** is located. We could refer to an object by pointer rather than by its name;

```
AClass* pointer_to_object = &ob;
```

In such a case, as usually, a dot should be repaleced by an “arrow”, and we can access the same member using '`->*`' operator:

```
pointer_to_object->*pointer
```

In a similar way one can define pointers to public, nonstatic methods. It is now harder to imagine the value of such a pointer as the value of a shift in memory relative to the beginning of an object — functions are not physically contained in objects. No matter how it is implemented, however, conceptually it does not differ very much from the case of data members (fields).

Suppose there are two methods in our class **AClass**, both of type **double** → **double**,

```
double fun1(double);
double fun2(double);
```

Then a pointer to class member, say ***pf***, which can point to one of these methods could be defined as (note the parentheses!):

```
double (AClass::*pf)(double);
```

which means: ***pf*** is a pointer to class member of class **AClass** which can point to any public, nonstatic method of type **double** → **double** of this class. Again, what differs this from a definition of a "normal" function pointer (see section on function pointers) is the presence of class-scope resolution operator. After such a definition, the pointer ***pf*** exists but does not point to anything. We can now assign it a value:

```
pf = AClass::fun1;
```

and now ***pf*** will point to the method **fun1**. Note that there are no parentheses after the name of a method; parentheses would imply an invocation. The pointer points to a nonstatic method, so in order to actually invoke it we have to specify for which object it is to work. As before, we can do it in two ways: using '`.*`', if we refer to the object in question by its name (or the name of a reference)

```
(ob.*pf)(5.5)
```

or using '`->*`' if we have a pointer to this object

```
(pointer_to_object->*pf)(5.5)
```

In both cases the parentheses are necessary because of precedence of operators.

One can use pointers to member functions when handling various kinds of menus. For example, one can define an array of such pointers, each of which points to a different method of the same class; the one that is to be invoked is then selected depending on a user's input:

```
double (AClass::*pf[8])(double);
AClass menu;
// ...
pf[0] = AClass::fun1;
pf[1] = AClass::fun2;
// ...
cin >> k;
(menu.*pf[k])( argument );
// ...
```

In the following example, we define a class with two fields of type **double** and two methods of type **double** → **double**:

P131: pointmem.cpp Pointers to members

```
1.  #include <iostream>
2.  #include <cmath>
3.  using namespace std;
4.
5.  struct Point {
6.      double x, y;
7.      Point(double x = 0, double y = 0)
8.          : x(x), y(y)
9.      { }
10.     double r2() { return  x*x + y*y; }
11.     double dd() { return sqrt(r2()); }
12. };
13.
14. int main() {
15.     double  Point::*pi[2];
16.     double (Point::*pf[2])();
17.
18.     pi[0] = &Point::x;
19.     pi[1] = &Point::y;
20.
21.     pf[0] = &Point::r2;
22.     pf[1] = &Point::dd;
23.
24.     Point P(3,4), *p = &P;
25.
26.     cout << " P.*pi[0]      = " <<  P.*pi[0]      << endl;
27.     cout << " P.*pi[1]      = " <<  P.*pi[1]      << endl;
28.     cout << "(P.*pf[0])()   = " << (P.*pf[0])()   << endl;
29.     cout << "(P.*pf[1])()   = " << (P.*pf[1])()   << endl;
30.
31.     cout << endl;
32.
33.     cout << " p->*pi[0]     = " <<  p->*pi[0]     << endl;
34.     cout << " p->*pi[1]     = " <<  p->*pi[1]     << endl;
35.     cout << "(p->*pf[0])() = " << (p->*pf[0])() << endl;
36.     cout << "(p->*pf[1])() = " << (p->*pf[1])() << endl;
37. }
```

We define two arrays of pointers to members: one, ***pi***, of pointers to fields, and another one, ***pf***, of pointers to member functions (methods). They are assigned values on lines 19-23 and used on lines 27-37. The printout

```
 P.*pi[0]      = 3
 P.*pi[1]      = 4
(P.*pf[0])()   = 25
(P.*pf[1])()   = 5

 p->*pi[0]     = 3
 p->*pi[1]     = 4
(p->*pf[0])() = 25
(p->*pf[1])() = 5
```

shows that everything works as expected.

Prev: 15.5 Nested classes **Up:** 15. Classes (II) **Next:** 16. Input/Output

16. Input/Output

We will make a break in discussing classes to say in more detail about input/output mechanisms in C++. So far we have only been using the simplest method of reading data and outputting information — the one based on '>>' and '<<' operators acting on ***cin*** and ***cout***. This was quite sufficient in simple programs, but there are more mechanisms which provide flexibility and efficiency of input/output operations in more sophisticated situations.

The information flows in both directions: from the “outside world” to a program (reading), and from a program out (writing). The rôle of the source when reading and of the destination when writing has been played by the terminal (its keyboard and monitor). But it does not have to be like that — it coud be a file, a communication socket, a region of computer's memory, etc.

Input/output operations in C++ are implemented in a completely different way than they were in C. Therefore, one *cannot* use them in C programs, although, of course, one *can* still use mechanisms from C in C++ programs.

Subsections

- 16.1 Streams
  - 16.1.1 Predefined streams
- 16.2 Operators << and >>
- 16.3 Formatting
  - 16.3.1 Format flags
  - 16.3.2 Manipulators
    - 16.3.2.1 Parameterless manipulators
    - 16.3.2.2 Manipulators with arguments
- 16.4 Unformatted I/O operations
  - 16.4.1 Unformatted input
  - 16.4.2 Unformatted output
- 16.5 Files
- 16.6 Handling stream errors
- 16.7 Internal files
  - 16.7.1 C-strings as internal files
  - 16.7.2 Internal files represented by C++ strings

Subsections

- 16.1.1 Predefined streams

16.1 Streams

Input/output operations act on **streams**, which can be imagined as information, in the form of bytes, flowing from a **source** to a **destination**. There are generally two possible scenarios here:

- information flows *from* a program *to* a destination, which can be a file, terminal, socket, region of memory, pipe, system FIFO queue, modem etc. This is called an **output stream**;
- information flows *to* a program *from* a source, which can be a file, terminal, socket, region of memory, a pipe, a system FIFO queue, a modem etc. This is called an **input stream**;

Streams are represented by special classes whose fields and methods are used to implement various input/output operations. These classes are rather complicated, but fortunately, we do not have to know their details to use them.

The general, simplified scheme of I/O classes is depicted in the following figure (classes **istrstream**, **ostrstream** and **strstream**, mentioned below, do not belong to this hierarchy):

The most important class is **ios** (itself derived from **ios_base**) which is the root of the whole hierarchy of more specialized classes:

- **istream** — the basic class representing input streams. The operator '`>>`' is redefined (overloaded) in this class. An instantiation (object) ***cin*** of the class is automatically created and accessible for the programmer. It represents the standard input stream (known as **stdin**). More specific classes which handle operations on input streams are:
  - **istringstream** — represents streams whose source is an object of class **string**. It can be accessed after `#include`'ing the header file **`sstream`**.
  - **istrstream** — represents streams whose source is an array of characters terminated by '`\0`'. It can be accessed after `#include`'ing **`strstream`**; NOTE: this class does *not* belong to the C++ standard!
  - **ifstream** (intput file stream) — represents streams whose source is a file.

Declared in header file `fstream`.

- **ostream** — the basic class representing output streams. The operator '<<' is redefined (overloaded) in this class. An instatiations (objects) ***cout***, ***cerr*** and ***clog*** of the class are automatically created and accessible for the programmer. They represent the standard output stream (known as **stdout**), and the standard error stream (**stderr**) nonbuffered and buffered, respectively. Manipulators ***endl*** and ***ends*** are also defined here. More specific classes which handle operations on output streams are:
  - **ostringstream** — with destination being an object of class **string**. Accessible by `#include`'ing the header `sstream`.
  - **ostrstream** — destination is a C-string (an array of characters terminated by '\0'). From header `strstream`. NOTE: this class does *not* belong to the C++ standard!
  - **ofstream** (*output file stream*) — destination is a file; from header `fstream`.

The class **iostream**, from the header of the same name, is derived from both **istream** and **ostream**; therefore, including `iostream` we have access to both of them. Similarly, including **fstream** we get the functionality of both **ifstream** and **ofstream**, and including **sstream** of **istringstream** and **ostringstream**.

Generally:

- if all we need are simple console operations, we include `iostream`;
- if there are input/output operation using disk files, `fstream` should be included;
- to use objects of class **string** as sources or destinations of I/O operations, we include `sstream`;
- to operate on C-strings as sources and/or destinations, `strstream` should be included.

16.1.1 Predefined streams

Having included the header `iostream`, we have at our disposal four already open streams: one input, ***cin***, and three output, ***cout***, ***cerr*** and ***clog***. These are actually names of objects representing streams: consequently, one can invoke several methods using these objects. These predefined objects are

- ***cin*** — standard input stream (***stdin***) which has been assigned to the process of our program by the operating system — by default it is connected to the keyboard, but this can be changed.
- ***cout*** — standard output stream (***stdout***) which also has been assigned to the process by the operating system — by default it is connected to the console (screen), but this can be changed; actually, it is often redirected to a file. The stream is **buffered**, what means that characters which are put to the stream do not appear on the screen immediately but with some delay (or not at all if the program crashed before the buffer had been flushed).
- ***cerr*** — standard error stream (***stderr***) assigned to the process by the operating system — by default it is also connected to the console. The stream is *not* buffered, what means that characters which are put to the stream appear on the screen immediately, so they should be visible even if the program crashes right after the output operation.
- ***clog*** — also corresponds to ***stderr***, but *is* buffered.

With all these streams one can use the operators '<<' (output, i.e., writing) and '>>' (input, i.e., reading), as well as a set of useful methods which will be described later in this chapter.

Prev: 16. Input/Output **Up:** 16. Input/Output **Next:** 16.2 Operators « and »

Subsections

- 16.1.1 Predefined streams

16.1 Streams

Input/output operations act on **streams**, which can be imagined as information, in the form of bytes, flowing from a **source** to a **destination**. There are generally two possible scenarios here:

- information flows *from* a program *to* a destination, which can be a file, terminal, socket, region of memory, pipe, system FIFO queue, modem etc. This is called an **output stream**;
- information flows *to* a program *from* a source, which can be a file, terminal, socket, region of memory, a pipe, a system FIFO queue, a modem etc. This is called an **input stream**;

Streams are represented by special classes whose fields and methods are used to implement various input/output operations. These classes are rather complicated, but fortunately, we do not have to know their details to use them.

The general, simplified scheme of I/O classes is depicted in the following figure (classes **istrstream**, **ostrstream** and **strstream**, mentioned below, do not belong to this hierarchy):

The most important class is **ios** (itself derived from **ios_base**) which is the root of the whole hierarchy of more specialized classes:

- **istream** — the basic class representing input streams. The operator '`>>`' is redefined (overloaded) in this class. An instantiation (object) ***cin*** of the class is automatically created and accessible for the programmer. It represents the standard input stream (known as **stdin**). More specific classes which handle operations on input streams are:
  - **istringstream** — represents streams whose source is an object of class **string**. It can be accessed after `#include`'ing the header file `sstream`.
  - **istrstream** — represents streams whose source is an array of characters terminated by '`\0`'. It can be accessed after `#include`'ing `strstream`; NOTE: this class does *not* belong to the C++ standard!
  - **ifstream** (intput file stream) — represents streams whose source is a file.

Declared in header file `fstream`.

- **ostream** — the basic class representing output streams. The operator '<<' is redefined (overloaded) in this class. An instatiations (objects) ***cout***, ***cerr*** and ***clog*** of the class are automatically created and accessible for the programmer. They represent the standard output stream (known as **stdout**), and the standard error stream (**stderr**) nonbuffered and buffered, respectively. Manipulators ***endl*** and ***ends*** are also defined here. More specific classes which handle operations on output streams are:
  - **ostringstream** — with destination being an object of class **string**. Accessible by `#include`'ing the header `sstream`.
  - **ostrstream** — destination is a C-string (an array of characters terminated by '\0'). From header `strstream`. NOTE: this class does *not* belong to the C++ standard!
  - **ofstream** (*output file stream*) — destination is a file; from header `fstream`.

The class **iostream**, from the header of the same name, is derived from both **istream** and **ostream**; therefore, including `iostream` we have access to both of them. Similarly, including **fstream** we get the functionality of both **ifstream** and **ofstream**, and including **sstream** of **istringstream** and **ostringstream**.

Generally:

- if all we need are simple console operations, we include `iostream`;
- if there are input/output operation using disk files, `fstream` should be included;
- to use objects of class **string** as sources or destinations of I/O operations, we include `sstream`;
- to operate on C-strings as sources and/or destinations, `strstream` should be included.

16.1.1 Predefined streams

Having included the header `iostream`, we have at our disposal four already open streams: one input, ***cin***, and three output, ***cout***, ***cerr*** and ***clog***. These are actually names of objects representing streams: consequently, one can invoke several methods using these objects. These predefined objects are

- ***cin*** — standard input stream (***stdin***) which has been assigned to the process of our program by the operating system — by default it is connected to the keyboard, but this can be changed.
- ***cout*** — standard output stream (***stdout***) which also has been assigned to the process by the operating system — by default it is connected to the console (screen), but this can be changed; actually, it is often redirected to a file. The stream is **buffered**, what means that characters which are put to the stream do not appear on the screen immediately but with some delay (or not at all if the program crashed before the buffer had been flushed).
- ***cerr*** — standard error stream (***stderr***) assigned to the process by the operating system — by default it is also connected to the console. The stream is *not* buffered, what means that characters which are put to the stream appear on the screen immediately, so they should be visible even if the program crashes right after the output operation.
- ***clog*** — also corresponds to ***stderr***, but *is* buffered.

With all these streams one can use the operators '<<' (output, i.e., writing) and '>>' (input, i.e., reading), as well as a set of useful methods which will be described later in this chapter.

Prev: 16. Input/Output **Up:** 16. Input/Output **Next:** 16.2 Operators « and »

16.2 Operators << and >>

What are '<<' and '>>' operators? We remember from the chapter on operators , that, e.g., the statement ' `a+b`' is really an invocation of a function performing the addition, to which values of ***a*** and ***b*** are passed, perhaps after some conversions, as arguments. In a similar way, operators '<<' and '>>' are binary operators whose task is to call an appropriate function which will perform an I/O operation. As with addition, values of expressions appearing to the left and to the right of the operator will be passed to this function as arguments. The first argument, corresponding to the left-hand operand, will be the reference to ***cin*** (or ***cout***), the second argument will be the value of expression appearing as the right-hand side operand. The function is overloaded: selection of the proper version will be made by the compiler on the basis of the type of the right-hand side operand. For the built-in types (and many types from the standard library), appropriate functions are already known to the compiler. For user-defined types, additional versions of these functions must be provided by the programmer — we will show how this can be done in the chapter on operator overloading .

It is very important to realize that the function invoked by '<<' or '>>' operators *do* return a value, namely a reference to the same stream-object which appears on the left-hand side of the operator. Hence, in

```
(cout << x) << y;
```

the value of the expression in parentheses is equivalent to ***cout***, which becomes in this way the left operand of the next '<<' operator. As the '<<' operator is left associative anyway, the parentheses are not necessary here: we can use '<<' in a cascade

```
cout << x << y << z << v << endl;
```

or

```
cin >> x >> y >> z >> v;
```

where, as we remember, direction of an “arrow” can be seen as the direction of the flow of information.

In order to output a string representing the value of a variable of, say, type **double**, the compiler must somehow decide in what form it should be diplayed: to the base 10 or 16, with how many digits etc. The way data is formatted before outputting to a stream can be modified by the programmer, but there are some defults which often are quite sufficient. For built-in types these defaults are the following:

For ouput streams:

- integer types (**int**, **short** etc.): in decimal notation;
- characters: as single characters;
- floating point types (**float**, **double** and **long double**): in decimal notation, with 6 digit

precision. Precision means here a number of significant digits, *not* a number of digits after the decimal point. For example, 200.0/3 will be displayed as 66.6667, and not 66.666667. Trailing zeros in fractional part will be omitted. Therefore, 1.129996 will be first rounded to 6 significant digits, yielding 1.13000, and then trailing zeros will be stripped, what will result in 1.13 (but, e.g., ouputting 1.129994 will produce 1.12999). If only zeros remain after the decimal point, neither zeros nor the decimal point itself are displayed. If the number of digits *before* the decimal point is larger than six, then all of them are printed and the fractional part is omitted;
- pointer values (except values of type **char***): as integer values corresponding to addresses, but in hexadecimal notation, together with the prefix '0x';
- pointers values of type **char***: as strings of characters, interpreting bytes in memory, starting from the byte pointed to by the pointer, as codes of characters; outputting terminates when the ***NUL*** character (' \0') is encountered;
- logical values (**bool**): as integers 1 and 0.

For input streams:

- leading white characters (spaces, tabs, newlines) are ignored;
- any sequence of white characters after non-white data is treated as a separator between pieces of data; it remains in the stream and will be ignored, as leading white characters, by the next reading operation;
- integer values are assumed to be represented in decimal notation; the first character may be '+' or '-', then digits are read until a non-digit is encountered — this will be left in the stream and will become the first character read by the next read operation. For example, for the statement '`cin >> x >> y`', the data may have the form 128-25; after reading 128 onto ***x***, the process will stop leaving '-' in the stream; next operation will read -25 onto ***y***;
- floating point numbers may be read as integer numbers without a decimal point, in the format with a decimal point, in the “scientific” notation (e.g., 1e-1 corresponds to 0.1, 1.201e+2 to 120.1);
- logical values should have the form of literals 1 and 0.

Subsections

- 16.3.1 Format flags
- 16.3.2 Manipulators
  - 16.3.2.1 Parameterless manipulators
  - 16.3.2.2 Manipulators with arguments

16.3 Formatting

The way in which data is formatted on output can be changed to make it more readable or better suited to some requirements. There are two tools providing flexibility of formatting data: format flags and manipulators.

16.3.1 Format flags

Formatting is defined by a **format flag** which is an attribute of every stream.

One can modify the format flag of a stream with the help of special predefined flags which are static members of **ios_base** class (they can also be accessed by referring to a derived class **ios**), for example: ' ***ios::left***', ' ***ios::scientific***' etc. These flags are constants of type **ios::fmtflags** (some older compilers do not support this type; usually type **long** can be used instead in such a case). Note that type name **fmtflags** is declared within **ios_base** class, so must be referred to by its qualified name **ios_base::fmtflags** (or just **ios::fmtflags**).

Bit representation of flags usually contains one bit set (1) and the remaining bits unset (0). One can then construct a format flag with desired properties by “ORing” several flags with bitwise OR operator ('|').

Before we present a few examples, let us list the flags which can be used to produce a format flag of a stream. Flags ***boolalpha***, ***showbase***, ***showpoint***, ***showpos***, ***skipws***, ***unitbuf*** and ***uppercase*** have “opposite” counterparts — their names are prefixed with '***no***', e.g., ***noboolalpha***, ***noshowbase*** etc.

ios::skipws — ignore leading white characters when reading (default: YES).

ios::left, ***ios::right***, ***ios::internal*** — output data will be left or right justified within the width of a character field. “Internal” justification means “sign (or other prefix, like 0x) to the left, number to the right”. For example, if we print the number -123 within an 8 character wide field, we get with the three justifications

```
|-123    |
|    -123|
|-    123|
```

The three flags constitute one format field ***ios::adjustfield***. At most one of the flags from

the field can be set. If none is set, text will be right justified.

ios::dec, ***ios::hex***, ***ios::oct*** — defines the base to which integer values are printed: decimal (default), hexadecimal and octal. Together they constitute format field ***ios::basefield***. At most one of the flags from the field can be set. If none is set, base 10 will be used.

ios::scientific, ***ios::fixed*** — define a format for floating point numbers. Together they constitute format field ***ios::floatfield***. At most one of the flags from the field can be set. If none is set, the default “general” format will be used. In scientific notation (flag ***scientific***), one significant digit is writen before the decimal point, then as many digits as determined by the current precision, and then the letter 'e' followed by a number indicating the exponent of 10 by which the number appearing to the left of 'e' should be multiplied. Before the number and before the exponential part, a minus sign is output, if appropriate. For example, 1.123456e2 means 112.3456, while -1.123456e-3 is -0.001123456. In fixed format (flag ***fixed***), numbers are output with as many digits after the decimal point as determined by the current value of precision. General format (the default) leaves the decision to the compiler — scientific or fixed format will be used depending on which one of them will fit smaller width with the same precision.

ios::boolalpha — if ***boolalpha*** is set, boolean values will be output as literals 'true' and 'false', instead of their numerical equivalents 1 and 0 (default: NO).

ios::showbase — if set, prefixes 0 and 0x will be added if an integer number is output in octal or hexadecimal representation (default: NO).

ios::showpoint — always show decimal point for floating point numbers, even if the fractional part is zero (default: NO).

ios::showpos — add plus sign before positive numbers (default: NO).

ios::uppercase — if set, letters 'e' in scientific notation and 'x' in hexadecimal notation are output in uppercase (default: NO).

ios::unitbuf — if set, the buffer of output stream is flushed after each insertion to the stream (default: NO).

As we have already mentioned, some flags form groups, called fields. This is because they are not independent: you cannot ask for hexadecimal and octal notation simultaneously. Therefore, flags ***ios::dec***, ***ios::hex*** and ***ios::oct*** belong to field ***ios::basefield***; flags ***ios::left***, ***ios::right*** and ***ios::internal*** to field ***ios::adjustfield*** and flags ***ios::scientific*** and ***ios::fixed*** to field ***ios::floatfield***. Flags which are members of fields have to be set in a special way, which will be described in the following.

A format flag is an attribute of every object representing a stream, so to access it, methods of object's class should be invoked for this object (e.g., for ***cin*** or ***cout***). The most important of these methods are:

ios::fmtflags flags( ) — returns the format flag associated with this stream;

ios::fmtflags flags(ios::fmtflags flg) — returns the format flag and sets its new value to ***flg***.

To set new format flag for a stream, one can first construct it by ORing predefined flags and

then call the methods described above, e.g.,

```
1.        // construct new flag
2.      ios::fmtflags n = ios::hex | ios::showbase
3.                                 | ios::uppercase;
4.        // set new flag and remember
5.        // its old value
6.      ios::fmtflags o = cout.flags(n);
7.      //
8.      // ... use new settings
9.      //
10.     cout.flags(o);  // restore old settings
```

Note that on line 6 we set a new value for the flag, but we remembered its old value; in this way we were able to restore old settings in the last line.

Sometimes we do not want to change the format flag entirely, but only add one flag to it. One can do it in the following way:

```
  // retrieve old flag
ios::fmtflags oldf = cout.flags();
  // create new one
ios::fmtflags newf  = oldf | ios::showpos;
  // set new one
cout.flags(newf);
// ... use new settings
// ... and restore old settings
cout.flags(oldf);
```

This way of setting the format flag can be somewhat troublesome, therefore there exist methods allowing to do it in a more direct way.

ios::fmtflags setf(ios::fmtflags flg) — modifies the format flag by ORing it with flag ***flg***; returns the original format flag;

ios::fmtflags setf(ios::fmtflags flg, ios::fmtflags field) modifies the format flag by ORing it with flag ***flg*** which belongs to field ***field***; unsets the other flags from the same field; returns the original format flag. Flags which are members of fields should be set in this way in order to avoid a situation when two mutually exclusive flags are set simultaneously. For example

```
cout.setf(ios::scientific, ios::floatfield);
```

ios::fmtflags unsetf(ios::fmtflags flg) — unsets flag ***flg*** in the format flag by ANDing the latter with bitwise negation of ***flg***.

The following example demonstrates how these function can be used:

P132: flags.cpp Format flags

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  typedef ios_base::fmtflags FFLAG;
5.
6.  int main() {
7.      int    m =  49;
8.      double x = 21.73;
9.
10.     cout << "1. m = " << m << ", x = " << x << endl;
11.
12.     FFLAG  newf = ios::hex | ios::showbase
```

```
13.                                 | ios::showpoint;
14.     FFLAG  oldf = cout.flags(newf);
15.     cout << "2. m = " << m << ", x = " << x << endl;
16.
17.     cout.setf(ios::scientific, ios::floatfield);
18.     cout.unsetf(ios::showbase);
19.     cout << "3. m = " << m << ", x = " << x << endl;
20.
21.     cout.setf(ios::fixed, ios::floatfield);
22.     cout.setf(ios::showbase | ios::uppercase);
23.     cout << "4. m = " << m << ", x = " << x << endl;
24.
25.     cout.flags(oldf);
26.     cout << "5. m = " << m << ", x = " << x << endl;
27. }
```

The output is:

```
1. m = 49, x = 21.73
2. m = 0x31, x = 21.7300
3. m = 31, x = 2.173000e+01
4. m = 0X31, x = 21.730000
5. m = 49, x = 21.73
```

Note that the default precision is 6, but the meaning of this fact depends on formatting: for scientific and fixed formats the 6 refers to the number of digits after the decimal point (see lines 3 and 4), while for general format it denotes the total number of significant digits (line 2). Additionally, if ***ios::showpos*** is not set, trailing zeros are omitted (lines 1 and 5). On line 4 of the program above, we have assigned a name to the type **ios_base::fmtflags** using the **typedef** — in this way we can use this alias instead of typing the, somewhat lengthy, true name of this type.

Class **ios** defines also methods which allow us to specify the field width (number of character) within which a given piece of data is to be output and methods for modifying the current precision. Being methods, they always have to be called for a specific object (stream), e.g., for ***cout*** or ***cin***:

streamsize width( ) — returns the current setting for the field width (length of the string within which a piece of data is to be output). The value 0 means “as many characters as necessary, but not more”. Type **streamsize** is a synonym of a signed integer type.

streamsize width(streamsize wid) — sets the value ***wid*** for the current field width; returns original setting. Note that field width is a *minimum* length of the output sequence of characters; if a given piece of data does not fit this size, it will *not* be truncated, but the field width actually used will be expanded as needed (as if field width were 0).

streamsize precision( ) — returns current setting of the floating-point numbers precision for this stream — see comments on precision after the program `flags.cpp`.

streamsize precision(streamsize prec) — sets new value, ***prec***, for precision, returns the previous value.

char fill( ) — returns the character which is currently used for padding if output field is longer than data to be printed (a space character by default).

char fill(char chr) — sets new value, ***chr***, for padding character, returns the previous value.

It should be kept in mind that

- setting the width of output field by invoking **width(int wid)** function affects *only* to the next stream insertion (write) operation; after that the default value (which is zero, meaning “as many as necessary”) will be restored;
- on the other hand, modifying precision of padding character is persistent — new settings will apply until they are explicitly changed.

The **width(int wid)** function (method) can also be applied to input streams. Its setting will be ignored if a number is read, but influences the way in which strings are read: it determines the maximum number of characters which will be read, including the '\0' terminator, which will always be added. Therefore,

```
char str[10];
cin.width(sizeof(str));
cin >> str;
```

will read at most 9 characters from keyboard, appending '\0' as the tenth, if necessary — this guarantees that even when the user typed more characters than expected, superfluous characters will not overflow the array ***str*** (corrupting the memory). These superfluous character will then remain in the keyboard buffer and will be input during the next stream extraction (read) operation (what can lead to confusion...). We will tell how one can clear the buffer in a moment.

16.3.2 Manipulators

As we could see, format flags are not particularly convenient. There is, however, a more “user friendly” mechanism of formatting data — the so called **manipulators**. These are basically functions defined in **ios** class and invoked by their names given as elements to be inserted to or extracted from a stream. As a result of their invocation, format flags can (although do not have to) be modified.

Actually, as we will see, manipulators do not have to be implemented as functions — they usually correspond to what is known as function objects, to be discussed in section on object functions .

There are tow kinds of manipulators: with and without parameters.

16.3.2.1 Parameterless manipulators

Parameterless manipulators are put into a stream without parentheses — just their names. There are quite a few such manipulators — their names correspond to the names of flags discussed in section on flags .

hex, ***oct***, ***dec*** — set the base of integer numbers to be output, as does an invocation of the method **setf(ios::hex,ios::basefield)** etc. Modification of format flag is persistent, to change it, one has to use another manipulator or call **setf** again.

left, ***right***, ***internal*** — set the way data is justified within its field, as does the method **setf(ios::left,ios::adjustfield)** etc.

fixed, ***scientific*** — they set the format for floating-point numbers, as does **setf(ios::fixed,ios::floatfield)** etc.

showbase, ***noshowbase*** — they act as invocation of the methods **setf(ios::showbase)** and **setf(ios::noshowbase)**.

showpoint, ***noshowpoint*** — as invoking methods **setf(ios::showpoint)** or **setf(ios::noshowpoint)**.

flush — flushes output stream.

endl — sends line termination character to output stream and flushes it, so all characters are immediately output to the destination of the stream (screen, file etc).

ends — sends C-string terminator, '\0', to output stream.

For example, the program

P133: mani.cpp Parameterless manipulators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int a = 0xdf, b = 0771, c = 123;
6.
7.      cout << "dec (default): "
8.           << dec << a << " " << b << " " << c << endl;
9.
10.     cout << "hex, no showbase:   "
11.          << hex << a << " " << b << " " << c << endl;
12.
13.     cout.setf(ios::showbase);
14.
15.     cout << "hex, with showbase: "
16.          << a << " " << b << " " << c << endl;
17.
18.     cout << "oct, with showbase: "
19.          << oct << a << " " << b << " " << c << endl;
20.
21.     cout.unsetf(ios::showbase);
22.
23.     cout << "oct, no showbase:   "
24.          << a << " " << b << " " << c << endl;
25. }
```

prints

```
dec (default): 223 505 123
hex, no showbase:   df 1f9 7b
hex, with showbase: 0xdf 0x1f9 0x7b
oct, with
```

We do not specify a base on line 15, as it has already been set as ***hex*** on line 11. We have to set it again only if we want to change it, e.g., for ***oct***, as we do on line 19.

It is relatively easy to define our own parameterless manipulator. In order to do it, we have to define a function with one parameter of type '*reference to a stream*' and returning by reference exactly the same stream; e.g.,

```
ostream& my_manip(ostream& stream) {
    // ...
    return stream;
}
```

When such a function is defined, we can use our manipulator just by inserting its name, without any arguments or parentheses, into a stream. The function will be invoked automatically with the current stream as an argument. It will return a reference to the stream, so the manipulator may be followed by another stream extraction or insertion operator, whichever is appropriate for the stream.

For example, the program

P134: wmani.cpp User defined manipulators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  ostream& scient(ostream&);
5.  ostream& normal(ostream&);
6.  ostream& acomma(ostream&);
7.
8.  int main() {
9.      double x = 123.456;
10.     cout << scient << x << acomma
11.          << normal << x << endl;
12. }
13.
14. ostream& scient(ostream& str) {
15.     str.setf(ios::showpos | ios::showpoint);
16.     str.setf(ios::scientific, ios::floatfield);
17.     str.precision(12);
18.     return str;
19. }
20.
21. ostream& normal(ostream& str) {
22.     str.flags((ios::fmtflags)0);
23.     return str;
24. }
25.
26. ostream& acomma(ostream& str) {
27.     return str << ", ";
28. }
```

prints

```
+1.234560000000e+02, 123.456
```

The first manipulator, ***scient***, defined on lines 14-19, using methods already known to us, modifies format flag of the stream. The second, ***normal***, restores default settings for the format flag; they correspond to the value of the format flag equal to zero of type **ios::fmtflags** — that is why we used casting on line 22. Finally, the third manipulator, ***acomma***, does not modify any flags — it just inserts a comma and a space into the stream.

The parameter of the function defining a manipulator is in all cases of type **ostream&**. This makes the construct quite flexible: an argument does not have to be ***cout***, it could be a reference to any object of type **ostream** or any type derived form **ostream**, e.g., an object of type **ofstream** representing an output stream connected to a file.

16.3.2.2 Manipulators with arguments

There are also manipulators *with* parameters. They are used in the same way as parameterless manipulators, but they require an argument (or arguments) to be specified (in parentheses, as in a function invocation). Usually, they are implemented as function objects

(see section on function objects).

Manipulators with parameters are accessible after including the header `iomanip`.

Several manipulators with parameters are already predefined; they perform the same tasks that the methods that we have already described, the main difference being the fact that they return, as all manipulators do, a reference to the stream they are inserted into.

setw(int wid) — sets the field width for the next I/O operation, as the *method* **width(int)**, that we have already described. The method returns old setting of the field width; the manipulator, as always, returns a reference to the stream. Both set the minimum value of field width — if this is not enough to represent a given piece of data, the field width will be expanded. Default value is 0, what means "as wide as needed, but no wider".

setfill(int padd) — sets padding character that is used to fill unused space when the field width is wider than the width of data to be output (space character by deault). Corresponds to the method **fill**.

setprecision(int prec) — sets precision, as the method **precision**.

setiosflags(ios::fmtflags flag) — modifies format flag, as one-argument method **setf**, i.e., "ORs" ***flag*** with the current format flag.

resetiosflags(ios::fmtflags flag) — corresponds to the method **unsetf**;

setbase(int base) — changes current value of base used when outputting integer numbers.

Some of these manipulators are demonstrated in the following program:

P135: primatr.cpp Formatting

```
1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4.
5. void printMatrix(ostream&,double**,int,int,const char*);
6.
7. int main() {
8.     const int DIM = 5;
9.     double t[][DIM] = { {    1,  3,    5,       23, 16.42},
10.                         {4.567,  4,    6,  234.345,    98},
11.                         {  585, 34,    1,       67,  31.2},
12.                         {    1,  0,    1, 2345.967, 123.2},
13.                         {  1.2, 10, 34.1,    5.900,   0.2}
14.                       };
15.     double* tab[DIM];
16.     for (int i = 0; i < 5; i++) tab[i] = t[i];
17.
18.     char  name[5];
19.     int  prec = 3;
20.
21.     cout << "Name: ";
22.     cin  >> setw(5) >> name;
23.
24.     printMatrix(cout, tab, DIM, prec, name);
25. }
26.
27. void printMatrix(ostream& strm, double* tab[], int size,
28.                                int prec, const char* name) {
```

```
29.     ios::fmtflags old =
30.         strm.setf(ios::fixed, ios::floatfield);
31.
32.     strm << setiosflags(strm.flags() | ios::showpoint)
33.          << setprecision(prec) << "\nMatrix: " << name
34.          << "\n\n";
35.
36.     for (int i = 0; i < size; i++) {
37.         strm << "ROW " << setfill('0') << setw(2)
38.              << (i+1)  <<  ":" << setfill(' ');
39.         for (int j = 0; j < size; j++)
40.             strm << setw(9) << tab[i][j];
41.         strm << endl;
42.     }
43.     strm << endl << setiosflags(old);
44. }
```

The program defines a function which prints a matrix in a readable form (the matrix is passed as an array of pointers to the beginnings of rows). Note that a stream to which the matrix is to be output is passed as an argument — in this way the same function can be used to output the matrix to a file.

Note also the way the program reads a name from the user (line 22). As the array which stores the name is only 5 characters long, we use **setw** to limit the length of a C-string which can be read: even if the user enters a longer name, it will be trucated (to four characters + ***NUL***), but no overflow of the array will occur:

```
Name: zanzibar

Matrix: zanz

ROW 01:     1.000     3.000     5.000    23.000    16.420
ROW 02:     4.567     4.000     6.000   234.345    98.000
ROW 03:   585.000    34.000     1.000    67.000    31.200
ROW 04:     1.000     0.000     1.000  2345.967   123.200
ROW 05:     1.200    10.000    34.100     5.900     0.200
```

Another important issue is avoiding side effects when calling a function. That is why our function, before returning, restores the format flag to the state it was on entry (lines 29 and 43).

It is possible to define our own manipulators with parameters, but it is more complicated than it was for parameterless manipulators; we will come back to this issue in section on function objects .

Subsections

- 16.3.1 Format flags
- 16.3.2 Manipulators
  - 16.3.2.1 Parameterless manipulators
  - 16.3.2.2 Manipulators with arguments

16.3 Formatting

The way in which data is formatted on output can be changed to make it more readable or better suited to some requirements. There are two tools providing flexibility of formatting data: format flags and manipulators.

16.3.1 Format flags

Formatting is defined by a **format flag** which is an attribute of every stream.

One can modify the format flag of a stream with the help of special predefined flags which are static members of **ios_base** class (they can also be accessed by referring to a derived class **ios**), for example: ' ***ios::left***', ' ***ios::scientific***' etc. These flags are constants of type **ios::fmtflags** (some older compilers do not support this type; usually type **long** can be used instead in such a case). Note that type name **fmtflags** is declared within **ios_base** class, so must be referred to by its qualified name **ios_base::fmtflags** (or just **ios::fmtflags**).

Bit representation of flags usually contains one bit set (1) and the remaining bits unset (0). One can then construct a format flag with desired properties by “ORing” several flags with bitwise OR operator ('|').

Before we present a few examples, let us list the flags which can be used to produce a format flag of a stream. Flags ***boolalpha***, ***showbase***, ***showpoint***, ***showpos***, ***skipws***, ***unitbuf*** and ***uppercase*** have “opposite” counterparts — their names are prefixed with '***no***', e.g., ***noboolalpha***, ***noshowbase*** etc.

ios::skipws — ignore leading white characters when reading (default: YES).

ios::left, ***ios::right***, ***ios::internal*** — output data will be left or right justified within the width of a character field. “Internal” justification means “sign (or other prefix, like 0x) to the left, number to the right”. For example, if we print the number -123 within an 8 character wide field, we get with the three justifications

```
|-123    |
|    -123|
|-    123|
```

The three flags constitute one format field ***ios::adjustfield***. At most one of the flags from

the field can be set. If none is set, text will be right justified.

ios::dec, ***ios::hex***, ***ios::oct*** — defines the base to which integer values are printed: decimal (default), hexadecimal and octal. Together they constitute format field ***ios::basefield***. At most one of the flags from the field can be set. If none is set, base 10 will be used.

ios::scientific, ***ios::fixed*** — define a format for floating point numbers. Together they constitute format field ***ios::floatfield***. At most one of the flags from the field can be set. If none is set, the default “general” format will be used. In scientific notation (flag ***scientific***), one significant digit is writen before the decimal point, then as many digits as determined by the current precision, and then the letter 'e' followed by a number indicating the exponent of 10 by which the number appearing to the left of 'e' should be multiplied. Before the number and before the exponential part, a minus sign is output, if appropriate. For example, 1.123456e2 means 112.3456, while -1.123456e-3 is -0.001123456. In fixed format (flag ***fixed***), numbers are output with as many digits after the decimal point as determined by the current value of precision. General format (the default) leaves the decision to the compiler — scientific or fixed format will be used depending on which one of them will fit smaller width with the same precision.

ios::boolalpha — if ***boolalpha*** is set, boolean values will be output as literals 'true' and 'false', instead of their numerical equivalents 1 and 0 (default: NO).

ios::showbase — if set, prefixes 0 and 0x will be added if an integer number is output in octal or hexadecimal representation (default: NO).

ios::showpoint — always show decimal point for floating point numbers, even if the fractional part is zero (default: NO).

ios::showpos — add plus sign before positive numbers (default: NO).

ios::uppercase — if set, letters 'e' in scientific notation and 'x' in hexadecimal notation are output in uppercase (default: NO).

ios::unitbuf — if set, the buffer of output stream is flushed after each insertion to the stream (default: NO).

As we have already mentioned, some flags form groups, called fields. This is because they are not independent: you cannot ask for hexadecimal and octal notation simultaneously. Therefore, flags ***ios::dec***, ***ios::hex*** and ***ios::oct*** belong to field ***ios::basefield***; flags ***ios::left***, ***ios::right*** and ***ios::internal*** to field ***ios::adjustfield*** and flags ***ios::scientific*** and ***ios::fixed*** to field ***ios::floatfield***. Flags which are members of fields have to be set in a special way, which will be described in the following.

A format flag is an attribute of every object representing a stream, so to access it, methods of object's class should be invoked for this object (e.g., for ***cin*** or ***cout***). The most important of these methods are:

ios::fmtflags flags( ) — returns the format flag associated with this stream;

ios::fmtflags flags(ios::fmtflags flg) — returns the format flag and sets its new value to ***flg***.

To set new format flag for a stream, one can first construct it by ORing predefined flags and

then call the methods described above, e.g.,

```
1.        // construct new flag
2.      ios::fmtflags n = ios::hex | ios::showbase
3.                                 | ios::uppercase;
4.        // set new flag and remember
5.        // its old value
6.      ios::fmtflags o = cout.flags(n);
7.      //
8.      // ... use new settings
9.      //
10.     cout.flags(o);  // restore old settings
```

Note that on line 6 we set a new value for the flag, but we remembered its old value; in this way we were able to restore old settings in the last line.

Sometimes we do not want to change the format flag entirely, but only add one flag to it. One can do it in the following way:

```
  // retrieve old flag
ios::fmtflags oldf = cout.flags();
  // create new one
ios::fmtflags newf  = oldf | ios::showpos;
  // set new one
cout.flags(newf);
// ... use new settings
// ... and restore old settings
cout.flags(oldf);
```

This way of setting the format flag can be somewhat troublesome, therefore there exist methods allowing to do it in a more direct way.

ios::fmtflags setf(ios::fmtflags flg) — modifies the format flag by ORing it with flag ***flg***; returns the original format flag;

ios::fmtflags setf(ios::fmtflags flg, ios::fmtflags field) modifies the format flag by ORing it with flag ***flg*** which belongs to field ***field***; unsets the other flags from the same field; returns the original format flag. Flags which are members of fields should be set in this way in order to avoid a situation when two mutually exclusive flags are set simultaneously. For example

```
cout.setf(ios::scientific, ios::floatfield);
```

ios::fmtflags unsetf(ios::fmtflags flg) — unsets flag ***flg*** in the format flag by ANDing the latter with bitwise negation of ***flg***.

The following example demonstrates how these function can be used:

P132: flags.cpp Format flags

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  typedef ios_base::fmtflags FFLAG;
5.
6.  int main() {
7.      int    m =  49;
8.      double x = 21.73;
9.
10.     cout << "1. m = " << m << ", x = " << x << endl;
11.
12.     FFLAG  newf = ios::hex | ios::showbase
```

```
13.                                 | ios::showpoint;
14.     FFLAG  oldf = cout.flags(newf);
15.     cout << "2. m = " << m << ", x = " << x << endl;
16.
17.     cout.setf(ios::scientific, ios::floatfield);
18.     cout.unsetf(ios::showbase);
19.     cout << "3. m = " << m << ", x = " << x << endl;
20.
21.     cout.setf(ios::fixed, ios::floatfield);
22.     cout.setf(ios::showbase | ios::uppercase);
23.     cout << "4. m = " << m << ", x = " << x << endl;
24.
25.     cout.flags(oldf);
26.     cout << "5. m = " << m << ", x = " << x << endl;
27. }
```

The output is:

```
1. m = 49, x = 21.73
2. m = 0x31, x = 21.7300
3. m = 31, x = 2.173000e+01
4. m = 0X31, x = 21.730000
5. m = 49, x = 21.73
```

Note that the default precision is 6, but the meaning of this fact depends on formatting: for scientific and fixed formats the 6 refers to the number of digits after the decimal point (see lines 3 and 4), while for general format it denotes the total number of significant digits (line 2). Additionally, if ***ios::showpos*** is not set, trailing zeros are omitted (lines 1 and 5). On line 4 of the program above, we have assigned a name to the type **ios_base::fmtflags** using the **typedef** — in this way we can use this alias instead of typing the, somewhat lengthy, true name of this type.

Class **ios** defines also methods which allow us to specify the field width (number of character) within which a given piece of data is to be output and methods for modifying the current precision. Being methods, they always have to be called for a specific object (stream), e.g., for ***cout*** or ***cin***:

streamsize width( ) — returns the current setting for the field width (length of the string within which a piece of data is to be output). The value 0 means “as many characters as necessary, but not more”. Type **streamsize** is a synonym of a signed integer type.

streamsize width(streamsize wid) — sets the value ***wid*** for the current field width; returns original setting. Note that field width is a *minimum* length of the output sequence of characters; if a given piece of data does not fit this size, it will *not* be truncated, but the field width actually used will be expanded as needed (as if field width were 0).

streamsize precision( ) — returns current setting of the floating-point numbers precision for this stream — see comments on precision after the program `flags.cpp`.

streamsize precision(streamsize prec) — sets new value, ***prec***, for precision, returns the previous value.

char fill( ) — returns the character which is currently used for padding if output field is longer than data to be printed (a space character by default).

char fill(char chr) — sets new value, ***chr***, for padding character, returns the previous value.

It should be kept in mind that

- setting the width of output field by invoking **width(int wid)** function affects *only* to the next stream insertion (write) operation; after that the default value (which is zero, meaning “as many as necessary”) will be restored;
- on the other hand, modifying precision of padding character is persistent — new settings will apply until they are explicitly changed.

The **width(int wid)** function (method) can also be applied to input streams. Its setting will be ignored if a number is read, but influences the way in which strings are read: it determines the maximum number of characters which will be read, including the '\0' terminator, which will always be added. Therefore,

```
char str[10];
cin.width(sizeof(str));
cin >> str;
```

will read at most 9 characters from keyboard, appending '\0' as the tenth, if necessary — this guarantees that even when the user typed more characters than expected, superfluous characters will not overflow the array ***str*** (corrupting the memory). These superfluous character will then remain in the keyboard buffer and will be input during the next stream extraction (read) operation (what can lead to confusion...). We will tell how one can clear the buffer in a moment.

16.3.2 Manipulators

As we could see, format flags are not particularly convenient. There is, however, a more “user friendly” mechanism of formatting data — the so called **manipulators**. These are basically functions defined in **ios** class and invoked by their names given as elements to be inserted to or extracted from a stream. As a result of their invocation, format flags can (although do not have to) be modified.

Actually, as we will see, manipulators do not have to be implemented as functions — they usually correspond to what is known as function objects, to be discussed in section on object functions .

There are tow kinds of manipulators: with and without parameters.

16.3.2.1 Parameterless manipulators

Parameterless manipulators are put into a stream without parentheses — just their names. There are quite a few such manipulators — their names correspond to the names of flags discussed in section on flags .

hex, ***oct***, ***dec*** — set the base of integer numbers to be output, as does an invocation of the method **setf(ios::hex,ios::basefield)** etc. Modification of format flag is persistent, to change it, one has to use another manipulator or call **setf** again.

left, ***right***, ***internal*** — set the way data is justified within its field, as does the method **setf(ios::left,ios::adjustfield)** etc.

fixed, ***scientific*** — they set the format for floating-point numbers, as does **setf(ios::fixed,ios::floatfield)** etc.

showbase, ***noshowbase*** — they act as invocation of the methods **setf(ios::showbase)** and **setf(ios::noshowbase)**.

showpoint, ***noshowpoint*** — as invoking methods **setf(ios::showpoint)** or **setf(ios::noshowpoint)**.

flush — flushes output stream.

endl — sends line termination character to output stream and flushes it, so all characters are immediately output to the destination of the stream (screen, file etc).

ends — sends C-string terminator, '\0', to output stream.

For example, the program

P133: mani.cpp Parameterless manipulators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int a = 0xdf, b = 0771, c = 123;
6.
7.      cout << "dec (default): "
8.           << dec << a << " " << b << " " << c << endl;
9.
10.     cout << "hex, no showbase:   "
11.          << hex << a << " " << b << " " << c << endl;
12.
13.     cout.setf(ios::showbase);
14.
15.     cout << "hex, with showbase: "
16.          << a << " " << b << " " << c << endl;
17.
18.     cout << "oct, with showbase: "
19.          << oct << a << " " << b << " " << c << endl;
20.
21.     cout.unsetf(ios::showbase);
22.
23.     cout << "oct, no showbase:   "
24.          << a << " " << b << " " << c << endl;
25. }
```

prints

```
dec (default): 223 505 123
hex, no showbase:   df 1f9 7b
hex, with showbase: 0xdf 0x1f9 0x7b
oct, with
```

We do not specify a base on line 15, as it has already been set as ***hex*** on line 11. We have to set it again only if we want to change it, e.g., for ***oct***, as we do on line 19.

It is relatively easy to define our own parameterless manipulator. In order to do it, we have to define a function with one parameter of type '*reference to a stream*' and returning by reference exactly the same stream; e.g.,

```
ostream& my_manip(ostream& stream) {
    // ...
    return stream;
}
```

When such a function is defined, we can use our manipulator just by inserting its name, without any arguments or parentheses, into a stream. The function will be invoked automatically with the current stream as an argument. It will return a reference to the stream, so the manipulator may be followed by another stream extraction or insertion operator, whichever is appropriate for the stream.

For example, the program

P134: wmani.cpp User defined manipulators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  ostream& scient(ostream&);
5.  ostream& normal(ostream&);
6.  ostream& acomma(ostream&);
7.
8.  int main() {
9.      double x = 123.456;
10.     cout << scient << x << acomma
11.          << normal << x << endl;
12. }
13.
14. ostream& scient(ostream& str) {
15.     str.setf(ios::showpos | ios::showpoint);
16.     str.setf(ios::scientific, ios::floatfield);
17.     str.precision(12);
18.     return str;
19. }
20.
21. ostream& normal(ostream& str) {
22.     str.flags((ios::fmtflags)0);
23.     return str;
24. }
25.
26. ostream& acomma(ostream& str) {
27.     return str << ", ";
28. }
```

prints

```
+1.234560000000e+02, 123.456
```

The first manipulator, ***scient***, defined on lines 14-19, using methods already known to us, modifies format flag of the stream. The second, ***normal***, restores default settings for the format flag; they correspond to the value of the format flag equal to zero of type **ios::fmtflags** — that is why we used casting on line 22. Finally, the third manipulator, ***acomma***, does not modify any flags — it just inserts a comma and a space into the stream.

The parameter of the function defining a manipulator is in all cases of type **ostream&**. This makes the construct quite flexible: an argument does not have to be ***cout***, it could be a reference to any object of type **ostream** or any type derived form **ostream**, e.g., an object of type **ofstream** representing an output stream connected to a file.

16.3.2.2 Manipulators with arguments

There are also manipulators *with* parameters. They are used in the same way as parameterless manipulators, but they require an argument (or arguments) to be specified (in parentheses, as in a function invocation). Usually, they are implemented as function objects

(see section on function objects).

Manipulators with parameters are accessible after including the header `iomanip`.

Several manipulators with parameters are already predefined; they perform the same tasks that the methods that we have already described, the main difference being the fact that they return, as all manipulators do, a reference to the stream they are inserted into.

setw(int wid) — sets the field width for the next I/O operation, as the *method* **width(int)**, that we have already described. The method returns old setting of the field width; the manipulator, as always, returns a reference to the stream. Both set the minimum value of field width — if this is not enough to represent a given piece of data, the field width will be expanded. Default value is 0, what means "as wide as needed, but no wider".

setfill(int padd) — sets padding character that is used to fill unused space when the field width is wider than the width of data to be output (space character by deault). Corresponds to the method **fill**.

setprecision(int prec) — sets precision, as the method **precision**.

setiosflags(ios::fmtflags flag) — modifies format flag, as one-argument method **setf**, i.e., "ORs" ***flag*** with the current format flag.

resetiosflags(ios::fmtflags flag) — corresponds to the method **unsetf**;

setbase(int base) — changes current value of base used when outputting integer numbers.

Some of these manipulators are demonstrated in the following program:

P135: primatr.cpp Formatting

```
1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4.
5. void printMatrix(ostream&,double**,int,int,const char*);
6.
7. int main() {
8.     const int DIM = 5;
9.     double t[][DIM] = { {    1,  3,    5,       23, 16.42},
10.                         {4.567,  4,    6,  234.345,    98},
11.                         {  585, 34,    1,       67,  31.2},
12.                         {    1,  0,    1, 2345.967, 123.2},
13.                         {  1.2, 10, 34.1,    5.900,   0.2}
14.                       };
15.     double* tab[DIM];
16.     for (int i = 0; i < 5; i++) tab[i] = t[i];
17.
18.     char  name[5];
19.     int  prec = 3;
20.
21.     cout << "Name: ";
22.     cin  >> setw(5) >> name;
23.
24.     printMatrix(cout, tab, DIM, prec, name);
25. }
26.
27. void printMatrix(ostream& strm, double* tab[], int size,
28.                                int prec, const char* name) {
```

```
29.     ios::fmtflags old =
30.         strm.setf(ios::fixed, ios::floatfield);
31.
32.     strm << setiosflags(strm.flags() | ios::showpoint)
33.          << setprecision(prec) << "\nMatrix: " << name
34.          << "\n\n";
35.
36.     for (int i = 0; i < size; i++) {
37.         strm << "ROW " << setfill('0') << setw(2)
38.              << (i+1)  <<  ":" << setfill(' ');
39.         for (int j = 0; j < size; j++)
40.             strm << setw(9) << tab[i][j];
41.         strm << endl;
42.     }
43.     strm << endl << setiosflags(old);
44. }
```

The program defines a function which prints a matrix in a readable form (the matrix is passed as an array of pointers to the beginnings of rows). Note that a stream to which the matrix is to be output is passed as an argument — in this way the same function can be used to output the matrix to a file.

Note also the way the program reads a name from the user (line 22). As the array which stores the name is only 5 characters long, we use **setw** to limit the length of a C-string which can be read: even if the user enters a longer name, it will be trucated (to four characters + ***NUL***), but no overflow of the array will occur:

```
Name: zanzibar

Matrix: zanz

ROW 01:     1.000     3.000     5.000    23.000    16.420
ROW 02:     4.567     4.000     6.000   234.345    98.000
ROW 03:   585.000    34.000     1.000    67.000    31.200
ROW 04:     1.000     0.000     1.000  2345.967   123.200
ROW 05:     1.200    10.000    34.100     5.900     0.200
```

Another important issue is avoiding side effects when calling a function. That is why our function, before returning, restores the format flag to the state it was on entry (lines 29 and 43).

It is possible to define our own manipulators with parameters, but it is more complicated than it was for parameterless manipulators; we will come back to this issue in section on function objects .

Subsections

- 16.3.1 Format flags
- 16.3.2 Manipulators
  - 16.3.2.1 Parameterless manipulators
  - 16.3.2.2 Manipulators with arguments

16.3 Formatting

The way in which data is formatted on output can be changed to make it more readable or better suited to some requirements. There are two tools providing flexibility of formatting data: format flags and manipulators.

16.3.1 Format flags

Formatting is defined by a **format flag** which is an attribute of every stream.

One can modify the format flag of a stream with the help of special predefined flags which are static members of **ios_base** class (they can also be accessed by referring to a derived class **ios**), for example: ' ***ios::left***', ' ***ios::scientific***' etc. These flags are constants of type **ios::fmtflags** (some older compilers do not support this type; usually type **long** can be used instead in such a case). Note that type name **fmtflags** is declared within **ios_base** class, so must be referred to by its qualified name **ios_base::fmtflags** (or just **ios::fmtflags**).

Bit representation of flags usually contains one bit set (1) and the remaining bits unset (0). One can then construct a format flag with desired properties by “ORing” several flags with bitwise OR operator ('|').

Before we present a few examples, let us list the flags which can be used to produce a format flag of a stream. Flags ***boolalpha***, ***showbase***, ***showpoint***, ***showpos***, ***skipws***, ***unitbuf*** and ***uppercase*** have “opposite” counterparts — their names are prefixed with '***no***', e.g., ***noboolalpha***, ***noshowbase*** etc.

ios::skipws — ignore leading white characters when reading (default: YES).

ios::left, ***ios::right***, ***ios::internal*** — output data will be left or right justified within the width of a character field. “Internal” justification means “sign (or other prefix, like 0x) to the left, number to the right”. For example, if we print the number -123 within an 8 character wide field, we get with the three justifications

```
|-123    |
|    -123|
|-    123|
```

The three flags constitute one format field ***ios::adjustfield***. At most one of the flags from

the field can be set. If none is set, text will be right justified.

ios::dec, ***ios::hex***, ***ios::oct*** — defines the base to which integer values are printed: decimal (default), hexadecimal and octal. Together they constitute format field ***ios::basefield***. At most one of the flags from the field can be set. If none is set, base 10 will be used.

ios::scientific, ***ios::fixed*** — define a format for floating point numbers. Together they constitute format field ***ios::floatfield***. At most one of the flags from the field can be set. If none is set, the default “general” format will be used. In scientific notation (flag ***scientific***), one significant digit is writen before the decimal point, then as many digits as determined by the current precision, and then the letter 'e' followed by a number indicating the exponent of 10 by which the number appearing to the left of 'e' should be multiplied. Before the number and before the exponential part, a minus sign is output, if appropriate. For example, 1.123456e2 means 112.3456, while -1.123456e-3 is -0.001123456. In fixed format (flag ***fixed***), numbers are output with as many digits after the decimal point as determined by the current value of precision. General format (the default) leaves the decision to the compiler — scientific or fixed format will be used depending on which one of them will fit smaller width with the same precision.

ios::boolalpha — if ***boolalpha*** is set, boolean values will be output as literals 'true' and 'false', instead of their numerical equivalents 1 and 0 (default: NO).

ios::showbase — if set, prefixes 0 and 0x will be added if an integer number is output in octal or hexadecimal representation (default: NO).

ios::showpoint — always show decimal point for floating point numbers, even if the fractional part is zero (default: NO).

ios::showpos — add plus sign before positive numbers (default: NO).

ios::uppercase — if set, letters 'e' in scientific notation and 'x' in hexadecimal notation are output in uppercase (default: NO).

ios::unitbuf — if set, the buffer of output stream is flushed after each insertion to the stream (default: NO).

As we have already mentioned, some flags form groups, called fields. This is because they are not independent: you cannot ask for hexadecimal and octal notation simultaneously. Therefore, flags ***ios::dec***, ***ios::hex*** and ***ios::oct*** belong to field ***ios::basefield***; flags ***ios::left***, ***ios::right*** and ***ios::internal*** to field ***ios::adjustfield*** and flags ***ios::scientific*** and ***ios::fixed*** to field ***ios::floatfield***. Flags which are members of fields have to be set in a special way, which will be described in the following.

A format flag is an attribute of every object representing a stream, so to access it, methods of object's class should be invoked for this object (e.g., for ***cin*** or ***cout***). The most important of these methods are:

ios::fmtflags flags( ) — returns the format flag associated with this stream;

ios::fmtflags flags(ios::fmtflags flg) — returns the format flag and sets its new value to ***flg***.

To set new format flag for a stream, one can first construct it by ORing predefined flags and

then call the methods described above, e.g.,

```
1.        // construct new flag
2.      ios::fmtflags n = ios::hex | ios::showbase
3.                                 | ios::uppercase;
4.        // set new flag and remember
5.        // its old value
6.      ios::fmtflags o = cout.flags(n);
7.      //
8.      // ... use new settings
9.      //
10.     cout.flags(o);  // restore old settings
```

Note that on line 6 we set a new value for the flag, but we remembered its old value; in this way we were able to restore old settings in the last line.

Sometimes we do not want to change the format flag entirely, but only add one flag to it. One can do it in the following way:

```
  // retrieve old flag
ios::fmtflags oldf = cout.flags();
  // create new one
ios::fmtflags newf  = oldf | ios::showpos;
  // set new one
cout.flags(newf);
// ... use new settings
// ... and restore old settings
cout.flags(oldf);
```

This way of setting the format flag can be somewhat troublesome, therefore there exist methods allowing to do it in a more direct way.

ios::fmtflags setf(ios::fmtflags flg) — modifies the format flag by ORing it with flag ***flg***; returns the original format flag;

ios::fmtflags setf(ios::fmtflags flg, ios::fmtflags field) modifies the format flag by ORing it with flag ***flg*** which belongs to field ***field***; unsets the other flags from the same field; returns the original format flag. Flags which are members of fields should be set in this way in order to avoid a situation when two mutually exclusive flags are set simultaneously. For example

```
cout.setf(ios::scientific, ios::floatfield);
```

ios::fmtflags unsetf(ios::fmtflags flg) — unsets flag ***flg*** in the format flag by ANDing the latter with bitwise negation of ***flg***.

The following example demonstrates how these function can be used:

P132: flags.cpp Format flags

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  typedef ios_base::fmtflags FFLAG;
5.
6.  int main() {
7.      int    m =  49;
8.      double x = 21.73;
9.
10.     cout << "1. m = " << m << ", x = " << x << endl;
11.
12.     FFLAG  newf = ios::hex | ios::showbase
```

```
13.                                 | ios::showpoint;
14.     FFLAG  oldf = cout.flags(newf);
15.     cout << "2. m = " << m << ", x = " << x << endl;
16.
17.     cout.setf(ios::scientific, ios::floatfield);
18.     cout.unsetf(ios::showbase);
19.     cout << "3. m = " << m << ", x = " << x << endl;
20.
21.     cout.setf(ios::fixed, ios::floatfield);
22.     cout.setf(ios::showbase | ios::uppercase);
23.     cout << "4. m = " << m << ", x = " << x << endl;
24.
25.     cout.flags(oldf);
26.     cout << "5. m = " << m << ", x = " << x << endl;
27. }
```

The output is:

```
1. m = 49, x = 21.73
2. m = 0x31, x = 21.7300
3. m = 31, x = 2.173000e+01
4. m = 0X31, x = 21.730000
5. m = 49, x = 21.73
```

Note that the default precision is 6, but the meaning of this fact depends on formatting: for scientific and fixed formats the 6 refers to the number of digits after the decimal point (see lines 3 and 4), while for general format it denotes the total number of significant digits (line 2). Additionally, if ***ios::showpos*** is not set, trailing zeros are omitted (lines 1 and 5). On line 4 of the program above, we have assigned a name to the type **ios_base::fmtflags** using the **typedef** — in this way we can use this alias instead of typing the, somewhat lengthy, true name of this type.

Class **ios** defines also methods which allow us to specify the field width (number of character) within which a given piece of data is to be output and methods for modifying the current precision. Being methods, they always have to be called for a specific object (stream), e.g., for ***cout*** or ***cin***:

streamsize width( ) — returns the current setting for the field width (length of the string within which a piece of data is to be output). The value 0 means “as many characters as necessary, but not more”. Type **streamsize** is a synonym of a signed integer type.

streamsize width(streamsize wid) — sets the value ***wid*** for the current field width; returns original setting. Note that field width is a *minimum* length of the output sequence of characters; if a given piece of data does not fit this size, it will *not* be truncated, but the field width actually used will be expanded as needed (as if field width were 0).

streamsize precision( ) — returns current setting of the floating-point numbers precision for this stream — see comments on precision after the program `flags.cpp`.

streamsize precision(streamsize prec) — sets new value, ***prec***, for precision, returns the previous value.

char fill( ) — returns the character which is currently used for padding if output field is longer than data to be printed (a space character by default).

char fill(char chr) — sets new value, ***chr***, for padding character, returns the previous value.

It should be kept in mind that

- setting the width of output field by invoking **width(int wid)** function affects *only* to the next stream insertion (write) operation; after that the default value (which is zero, meaning “as many as necessary”) will be restored;
- on the other hand, modifying precision of padding character is persistent — new settings will apply until they are explicitly changed.

The **width(int wid)** function (method) can also be applied to input streams. Its setting will be ignored if a number is read, but influences the way in which strings are read: it determines the maximum number of characters which will be read, including the '\0' terminator, which will always be added. Therefore,

```
char str[10];
cin.width(sizeof(str));
cin >> str;
```

will read at most 9 characters from keyboard, appending '\0' as the tenth, if necessary — this guarantees that even when the user typed more characters than expected, superfluous characters will not overflow the array ***str*** (corrupting the memory). These superfluous character will then remain in the keyboard buffer and will be input during the next stream extraction (read) operation (what can lead to confusion...). We will tell how one can clear the buffer in a moment.

16.3.2 Manipulators

As we could see, format flags are not particularly convenient. There is, however, a more “user friendly” mechanism of formatting data — the so called **manipulators**. These are basically functions defined in **ios** class and invoked by their names given as elements to be inserted to or extracted from a stream. As a result of their invocation, format flags can (although do not have to) be modified.

Actually, as we will see, manipulators do not have to be implemented as functions — they usually correspond to what is known as function objects, to be discussed in section on object functions .

There are tow kinds of manipulators: with and without parameters.

16.3.2.1 Parameterless manipulators

Parameterless manipulators are put into a stream without parentheses — just their names. There are quite a few such manipulators — their names correspond to the names of flags discussed in section on flags .

hex, ***oct***, ***dec*** — set the base of integer numbers to be output, as does an invocation of the method **setf(ios::hex,ios::basefield)** etc. Modification of format flag is persistent, to change it, one has to use another manipulator or call **setf** again.

left, ***right***, ***internal*** — set the way data is justified within its field, as does the method **setf(ios::left,ios::adjustfield)** etc.

fixed, ***scientific*** — they set the format for floating-point numbers, as does **setf(ios::fixed,ios::floatfield)** etc.

showbase, ***noshowbase*** — they act as invocation of the methods **setf(ios::showbase)** and **setf(ios::noshowbase)**.

showpoint, ***noshowpoint*** — as invoking methods **setf(ios::showpoint)** or **setf(ios::noshowpoint)**.

flush — flushes output stream.

endl — sends line termination character to output stream and flushes it, so all characters are immediately output to the destination of the stream (screen, file etc).

ends — sends C-string terminator, '\0', to output stream.

For example, the program

P133: mani.cpp Parameterless manipulators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      int a = 0xdf, b = 0771, c = 123;
6.
7.      cout << "dec (default): "
8.           << dec << a << " " << b << " " << c << endl;
9.
10.     cout << "hex, no showbase:   "
11.          << hex << a << " " << b << " " << c << endl;
12.
13.     cout.setf(ios::showbase);
14.
15.     cout << "hex, with showbase: "
16.          << a << " " << b << " " << c << endl;
17.
18.     cout << "oct, with showbase: "
19.          << oct << a << " " << b << " " << c << endl;
20.
21.     cout.unsetf(ios::showbase);
22.
23.     cout << "oct, no showbase:   "
24.          << a << " " << b << " " << c << endl;
25. }
```

prints

```
dec (default): 223 505 123
hex, no showbase:   df 1f9 7b
hex, with showbase: 0xdf 0x1f9 0x7b
oct, with
```

We do not specify a base on line 15, as it has already been set as ***hex*** on line 11. We have to set it again only if we want to change it, e.g., for ***oct***, as we do on line 19.

It is relatively easy to define our own parameterless manipulator. In order to do it, we have to define a function with one parameter of type '*reference to a stream*' and returning by reference exactly the same stream; e.g.,

```
ostream& my_manip(ostream& stream) {
    // ...
    return stream;
}
```

When such a function is defined, we can use our manipulator just by inserting its name, without any arguments or parentheses, into a stream. The function will be invoked automatically with the current stream as an argument. It will return a reference to the stream, so the manipulator may be followed by another stream extraction or insertion operator, whichever is appropriate for the stream.

For example, the program

P134: wmani.cpp User defined manipulators

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  ostream& scient(ostream&);
5.  ostream& normal(ostream&);
6.  ostream& acomma(ostream&);
7.
8.  int main() {
9.      double x = 123.456;
10.     cout << scient << x << acomma
11.          << normal << x << endl;
12. }
13.
14. ostream& scient(ostream& str) {
15.     str.setf(ios::showpos | ios::showpoint);
16.     str.setf(ios::scientific, ios::floatfield);
17.     str.precision(12);
18.     return str;
19. }
20.
21. ostream& normal(ostream& str) {
22.     str.flags((ios::fmtflags)0);
23.     return str;
24. }
25.
26. ostream& acomma(ostream& str) {
27.     return str << ", ";
28. }
```

prints

```
+1.234560000000e+02, 123.456
```

The first manipulator, ***scient***, defined on lines 14-19, using methods already known to us, modifies format flag of the stream. The second, ***normal***, restores default settings for the format flag; they correspond to the value of the format flag equal to zero of type **ios::fmtflags** — that is why we used casting on line 22. Finally, the third manipulator, ***acomma***, does not modify any flags — it just inserts a comma and a space into the stream.

The parameter of the function defining a manipulator is in all cases of type **ostream&**. This makes the construct quite flexible: an argument does not have to be ***cout***, it could be a reference to any object of type **ostream** or any type derived form **ostream**, e.g., an object of type **ofstream** representing an output stream connected to a file.

16.3.2.2 Manipulators with arguments

There are also manipulators *with* parameters. They are used in the same way as parameterless manipulators, but they require an argument (or arguments) to be specified (in parentheses, as in a function invocation). Usually, they are implemented as function objects

(see section on function objects).

Manipulators with parameters are accessible after including the header `iomanip`.

Several manipulators with parameters are already predefined; they perform the same tasks that the methods that we have already described, the main difference being the fact that they return, as all manipulators do, a reference to the stream they are inserted into.

setw(int wid) — sets the field width for the next I/O operation, as the *method* **width(int)**, that we have already described. The method returns old setting of the field width; the manipulator, as always, returns a reference to the stream. Both set the minimum value of field width — if this is not enough to represent a given piece of data, the field width will be expanded. Default value is 0, what means "as wide as needed, but no wider".

setfill(int padd) — sets padding character that is used to fill unused space when the field width is wider than the width of data to be output (space character by deault). Corresponds to the method **fill**.

setprecision(int prec) — sets precision, as the method **precision**.

setiosflags(ios::fmtflags flag) — modifies format flag, as one-argument method **setf**, i.e., "ORs" ***flag*** with the current format flag.

resetiosflags(ios::fmtflags flag) — corresponds to the method **unsetf**;

setbase(int base) — changes current value of base used when outputting integer numbers.

Some of these manipulators are demonstrated in the following program:

P135: primatr.cpp Formatting

```
1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4.
5. void printMatrix(ostream&,double**,int,int,const char*);
6.
7. int main() {
8.     const int DIM = 5;
9.     double t[][DIM] = { {    1,  3,    5,       23, 16.42},
10.                         {4.567,  4,    6,  234.345,    98},
11.                         {  585, 34,    1,       67,  31.2},
12.                         {    1,  0,    1, 2345.967, 123.2},
13.                         {  1.2, 10, 34.1,    5.900,   0.2}
14.                       };
15.     double* tab[DIM];
16.     for (int i = 0; i < 5; i++) tab[i] = t[i];
17.
18.     char  name[5];
19.     int  prec = 3;
20.
21.     cout << "Name: ";
22.     cin  >> setw(5) >> name;
23.
24.     printMatrix(cout, tab, DIM, prec, name);
25. }
26.
27. void printMatrix(ostream& strm, double* tab[], int size,
28.                                int prec, const char* name) {
```

```
29.     ios::fmtflags old =
30.         strm.setf(ios::fixed, ios::floatfield);
31.
32.     strm << setiosflags(strm.flags() | ios::showpoint)
33.          << setprecision(prec) << "\nMatrix: " << name
34.          << "\n\n";
35.
36.     for (int i = 0; i < size; i++) {
37.         strm << "ROW " << setfill('0') << setw(2)
38.              << (i+1)  <<  ":" << setfill(' ');
39.         for (int j = 0; j < size; j++)
40.             strm << setw(9) << tab[i][j];
41.         strm << endl;
42.     }
43.     strm << endl << setiosflags(old);
44. }
```

The program defines a function which prints a matrix in a readable form (the matrix is passed as an array of pointers to the beginnings of rows). Note that a stream to which the matrix is to be output is passed as an argument — in this way the same function can be used to output the matrix to a file.

Note also the way the program reads a name from the user (line 22). As the array which stores the name is only 5 characters long, we use **setw** to limit the length of a C-string which can be read: even if the user enters a longer name, it will be trucated (to four characters + ***NUL***), but no overflow of the array will occur:

```
Name: zanzibar

Matrix: zanz

ROW 01:     1.000     3.000     5.000    23.000    16.420
ROW 02:     4.567     4.000     6.000   234.345    98.000
ROW 03:   585.000    34.000     1.000    67.000    31.200
ROW 04:     1.000     0.000     1.000  2345.967   123.200
ROW 05:     1.200    10.000    34.100     5.900     0.200
```

Another important issue is avoiding side effects when calling a function. That is why our function, before returning, restores the format flag to the state it was on entry (lines 29 and 43).

It is possible to define our own manipulators with parameters, but it is more complicated than it was for parameterless manipulators; we will come back to this issue in section on function objects .

Subsections

- 16.4.1 Unformatted input
- 16.4.2 Unformatted output

16.4 Unformatted I/O operations

Input/output operations that we have dealt with so far were formatted — information was somehow interpreted: white space was ignored, numbers were converted to strings which only then were displayed or printed, etc. However, there are operations when what we want is to read or write just “raw bytes”, without interpreting them.

16.4.1 Unformatted input

There are a few methods which can be invoked for an input stream object in order to read a portion of information in the form of bytes, without any translation.

istream& get(char& c) — reads one byte and returns a reference to the stream, so it can be used in a cascade. The argument ***c*** is passed by reference and modified, so on return it contains the byte just read in (it can be *any* byte, also corresponding to a control character, white space or ***NUL*** character (' \0'). If reading failed, ***c*** will be equal to a special value ***EOF***,, i.e., a byte which, on a given platform, denotes end of file (Ctrl-Z in Windows, Ctrl-D in Unix/Linux). After a failure, the state of the stream will be ***bad*** (more details will be given shortly). We can recognize if this is the case by comparing the value of a stream with **nullptr** (or 0). The value will be converted to type **void*** and **nullptr** means that the stream is in a state of an error; any other value signals that the stream is “healthy”. Such a conversion is performed automatically in a context requiring a logical value. This means that this value may be used in conditional statements (**if**), and in loops **for**, **while** etc. For example, if ***strin*** is an input stream connected to a file, we can copy the file to the standard output by a simple loop:

```
char c;
while (strin.get(c)) cout << c;
```

The fact that **get** returns ****this***, i.e., the stream itself, can be used for reading in a cascade:

```
char a, b, c;
strm.get(a).get(b).get(c);
```

where three characters are read, without ignoring white characters or interpreting ***EOL*** (end of line) character in any way.

int get() — returns one character as an integer; if the end of file has been encountered, ***EOF*** will be returned. As the function does not return the stream, it cannot be used in a cascade.

istream& get(char* buf, streamsize length, char termin = '\n') — reads character into a buffer ***buf*** (which points to an array of characters). It reads at most ***length-1*** bytes: ***NUL*** character is always appended at the end. Reading terminates if end of stream is encountered, or the ***termin*** character is read, in which case it is put back to the stream (and will be the first character read by the next input operation — we can get rid of it by invoking **ignore**, described below). The default value of ***termin*** is the end of line character '\n'. The ***NUL*** character will be appended in any case. Of course, a sufficiently large buffer ***buf*** has to be allocated *before* calling the function! The second parameter is of type **streamsize**, which is an alias of an integer type. The function returns ****this***, i.e., a reference to the stream.

istream& getline(char* buf, streamsize length, char termin = '\n') — similar to the previous function, but the terminating character ***termin***, if encountered, *is* extracted form the stream, but not copied to the buffer ***buf***. This method is usually more practical than the previous one when reading data from a stream line by line.

istream& read(char* buf, streamsize length) — reads ***length*** bytes into buffer ***buf*** (which must exist and be large enough). Reading can terminate earlier only if the end of stream has been encountered. Raw bytes are read without any interpretation, the ***NUL*** character is *not* appended (the function is usually used for reading non-textual data). Function **gcount** (see below) can be used to retrieve the number of bytes actually read (which can be different than ***length*** if end of data has been encountered). The function returns ****this***, i.e., the stream.

istream& ignore(streamsize length = 1, int termin = EOF) — reads and ignores ***length*** bytes (one by default). If character ***termin*** has been encountered (end of file by default), it is extracted from the stream and reading terminates. The function returns ****this***.

int gcount( ) — returns number of bytes read by the last operation of unformatted reading.

int peek( ) — returns, as an **int**, one character from the stream, leaving it there, so it will be the first character read by the next input operation. It can be ***EOF***, if the end of stream has been encountered.

istream& putback(char c) — puts character ***c*** into the stream, so it will be the first character read by the next input operation. The operation is not always possible.

istream& unget() — puts back into the stream the last character extracted. The operation is not always possible. Returns ****this***.

Some of these methods have been used in the following program. It reads lines from the standard input and rewrites them into standard output, but ignoring comments, i.e., parts of lines which start with digraph '//':

P136: unfrd.cpp Unformatted reading

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      cout << "Enter lines of text. Terminate with EOF "
6.              "character\n(Ctrl-Z in Windows, Ctrl-D "
7.              "in Linux). Comments\nfrom  \'//\' to "
8.              "the end of line will be ignored.\n";
9.      char c;
10.     while ( cin.get(c) ) {
```

```
11.         if ( c == '/' )
12.             if ( cin.peek() == '/') {
13.                 cin.ignore(1024,'\n');
14.                 cout << endl;
15.                 continue;
16.             }
17.         cout << c;
18.     }
19. }
```

After reading a character, the program checks if this is a slash (line 11). If so, function **peek** is used to see if the next character is also a slash: in that case the rest of the line will be read off and ignored by **ignore** (line 13) and ***EOF*** will be output; otherwise the loop will continue. The program stops after reading an ***EOF*** character, i.e., when end of file has been encountered — this can be simulated by entering Ctrl-Z (Windows) or Ctrl-D (Linux). The conversion of the stream variable in line 10 yields **nullptr**, what terminates the loop. Running the program produces:

```
Enter lines of text. Terminate with EOF character
(Ctrl-Z in Windows, Ctrl-D in Linux). Comments
from  '//' to the end of line will be ignored.
int main(void) {
int main(void) {
    const int DIM = 15; // dimension of array
    const int DIM = 15;
    int tab[DIM];       // array of integers
    int tab[DIM];
    double x=1, y=2, z=x/y; // three doubles
    double x=1, y=2, z=x/y;
    // ...

}
}
^D
```

Note also lines 5-8 of the program. They illustrate a feature of C/C++ compilers: two literal C-strings (enclosed in double quotation marks) are automatically concatenated into one C-string if they are separated by a nonempty sequence of white characters (including the new-line character).

16.4.2 Unformatted output

Writing unformatted data is quite common. Of course, it is used for outputting data to a file or a socket, not to the computer screen. It allows us to output numbers with full machine precision in a form which saves space on disk (note that -1.234567E+19 takes 13 characters; the same **float** number with full machine precision takes only 4 bytes in memory, so only 4 bytes need to be output to a binary file). Binary (nontextual) files are not viewed or modified in editors, but can be easily read by other programs. Many files are “by definition” nontextual and all operations on them must be unformatted (sound, graphics etc.).

Two basic methods (therefore, called for a stream object) which output binary data are:

ostream& put(char c) — inserts single character (byte) ***c*** into the stream; returns ****this***, i.e., the stream.

ostream& write(const char* buf, streamsize length) — ouputs exactly ***length*** characters (bytes) from buffer (array) ***buf***; returns ****this***.

We will use these functions in examples of operations on files (next section).

Prev: 16.3 Formatting **Up:** 16. Input/Output **Next:** 16.5 Files

Subsections

- 16.4.1 Unformatted input
- 16.4.2 Unformatted output

16.4 Unformatted I/O operations

Input/output operations that we have dealt with so far were formatted — information was somehow interpreted: white space was ignored, numbers were converted to strings which only then were displayed or printed, etc. However, there are operations when what we want is to read or write just “raw bytes”, without interpreting them.

16.4.1 Unformatted input

There are a few methods which can be invoked for an input stream object in order to read a portion of information in the form of bytes, without any translation.

istream& get(char& c) — reads one byte and returns a reference to the stream, so it can be used in a cascade. The argument ***c*** is passed by reference and modified, so on return it contains the byte just read in (it can be *any* byte, also corresponding to a control character, white space or ***NUL*** character (' \0'). If reading failed, ***c*** will be equal to a special value ***EOF***,, i.e., a byte which, on a given platform, denotes end of file (Ctrl-Z in Windows, Ctrl-D in Unix/Linux). After a failure, the state of the stream will be ***bad*** (more details will be given shortly). We can recognize if this is the case by comparing the value of a stream with **nullptr** (or 0). The value will be converted to type **void*** and **nullptr** means that the stream is in a state of an error; any other value signals that the stream is “healthy”. Such a conversion is performed automatically in a context requiring a logical value. This means that this value may be used in conditional statements (**if**), and in loops **for**, **while** etc. For example, if ***strin*** is an input stream connected to a file, we can copy the file to the standard output by a simple loop:

```
char c;
while (strin.get(c)) cout << c;
```

The fact that **get** returns ***this**, i.e., the stream itself, can be used for reading in a cascade:

```
char a, b, c;
strm.get(a).get(b).get(c);
```

where three characters are read, without ignoring white characters or interpreting ***EOL*** (end of line) character in any way.

int get() — returns one character as an integer; if the end of file has been encountered, ***EOF*** will be returned. As the function does not return the stream, it cannot be used in a cascade.

istream& get(char* buf, streamsize length, char termin = '\n') — reads character into a buffer ***buf*** (which points to an array of characters). It reads at most ***length-1*** bytes: ***NUL*** character is always appended at the end. Reading terminates if end of stream is encountered, or the ***termin*** character is read, in which case it is put back to the stream (and will be the first character read by the next input operation — we can get rid of it by invoking **ignore**, described below). The default value of ***termin*** is the end of line character '\n'. The ***NUL*** character will be appended in any case. Of course, a sufficiently large buffer ***buf*** has to be allocated *before* calling the function! The second parameter is of type **streamsize**, which is an alias of an integer type. The function returns ****this***, i.e., a reference to the stream.

istream& getline(char* buf, streamsize length, char termin = '\n') — similar to the previous function, but the terminating character ***termin***, if encountered, *is* extracted form the stream, but not copied to the buffer ***buf***. This method is usually more practical than the previous one when reading data from a stream line by line.

istream& read(char* buf, streamsize length) — reads ***length*** bytes into buffer ***buf*** (which must exist and be large enough). Reading can terminate earlier only if the end of stream has been encountered. Raw bytes are read without any interpretation, the ***NUL*** character is *not* appended (the function is usually used for reading non-textual data). Function **gcount** (see below) can be used to retrieve the number of bytes actually read (which can be different than ***length*** if end of data has been encountered). The function returns ****this***, i.e., the stream.

istream& ignore(streamsize length = 1, int termin = EOF) — reads and ignores ***length*** bytes (one by default). If character ***termin*** has been encountered (end of file by default), it is extracted from the stream and reading terminates. The function returns ****this***.

int gcount( ) — returns number of bytes read by the last operation of unformatted reading.

int peek( ) — returns, as an **int**, one character from the stream, leaving it there, so it will be the first character read by the next input operation. It can be ***EOF***, if the end of stream has been encountered.

istream& putback(char c) — puts character ***c*** into the stream, so it will be the first character read by the next input operation. The operation is not always possible.

istream& unget() — puts back into the stream the last character extracted. The operation is not always possible. Returns ****this***.

Some of these methods have been used in the following program. It reads lines from the standard input and rewrites them into standard output, but ignoring comments, i.e., parts of lines which start with digraph '//':

P136: unfrd.cpp Unformatted reading

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      cout << "Enter lines of text. Terminate with EOF "
6.              "character\n(Ctrl-Z in Windows, Ctrl-D "
7.              "in Linux). Comments\nfrom  \'//\' to "
8.              "the end of line will be ignored.\n";
9.      char c;
10.     while ( cin.get(c) ) {
```

```
11.         if ( c == '/' )
12.             if ( cin.peek() == '/') {
13.                 cin.ignore(1024,'\n');
14.                 cout << endl;
15.                 continue;
16.             }
17.         cout << c;
18.     }
19. }
```

After reading a character, the program checks if this is a slash (line 11). If so, function **peek** is used to see if the next character is also a slash: in that case the rest of the line will be read off and ignored by **ignore** (line 13) and ***EOF*** will be output; otherwise the loop will continue. The program stops after reading an ***EOF*** character, i.e., when end of file has been encountered — this can be simulated by entering Ctrl-Z (Windows) or Ctrl-D (Linux). The conversion of the stream variable in line 10 yields **nullptr**, what terminates the loop. Running the program produces:

```
Enter lines of text. Terminate with EOF character
(Ctrl-Z in Windows, Ctrl-D in Linux). Comments
from  '//' to the end of line will be ignored.
int main(void) {
int main(void) {
    const int DIM = 15; // dimension of array
    const int DIM = 15;
    int tab[DIM];       // array of integers
    int tab[DIM];
    double x=1, y=2, z=x/y; // three doubles
    double x=1, y=2, z=x/y;
    // ...

}
}
^D
```

Note also lines 5-8 of the program. They illustrate a feature of C/C++ compilers: two literal C-strings (enclosed in double quotation marks) are automatically concatenated into one C-string if they are separated by a nonempty sequence of white characters (including the new-line character).

16.4.2 Unformatted output

Writing unformatted data is quite common. Of course, it is used for outputting data to a file or a socket, not to the computer screen. It allows us to output numbers with full machine precision in a form which saves space on disk (note that -1.234567E+19 takes 13 characters; the same **float** number with full machine precision takes only 4 bytes in memory, so only 4 bytes need to be output to a binary file). Binary (nontextual) files are not viewed or modified in editors, but can be easily read by other programs. Many files are “by definition” nontextual and all operations on them must be unformatted (sound, graphics etc.).

Two basic methods (therefore, called for a stream object) which output binary data are:

ostream& put(char c) — inserts single character (byte) ***c*** into the stream; returns ****this***, i.e., the stream.

ostream& write(const char* buf, streamsize length) — ouputs exactly ***length*** characters (bytes) from buffer (array) ***buf***; returns ****this***.

We will use these functions in examples of operations on files (next section).

Prev: 16.3 Formatting **Up:** 16. Input/Output **Next:** 16.5 Files

Subsections

- 16.4.1 Unformatted input
- 16.4.2 Unformatted output

16.4 Unformatted I/O operations

Input/output operations that we have dealt with so far were formatted — information was somehow interpreted: white space was ignored, numbers were converted to strings which only then were displayed or printed, etc. However, there are operations when what we want is to read or write just “raw bytes”, without interpreting them.

16.4.1 Unformatted input

There are a few methods which can be invoked for an input stream object in order to read a portion of information in the form of bytes, without any translation.

istream& get(char& c) — reads one byte and returns a reference to the stream, so it can be used in a cascade. The argument ***c*** is passed by reference and modified, so on return it contains the byte just read in (it can be *any* byte, also corresponding to a control character, white space or ***NUL*** character (' \0'). If reading failed, ***c*** will be equal to a special value ***EOF***,, i.e., a byte which, on a given platform, denotes end of file (Ctrl-Z in Windows, Ctrl-D in Unix/Linux). After a failure, the state of the stream will be ***bad*** (more details will be given shortly). We can recognize if this is the case by comparing the value of a stream with **nullptr** (or 0). The value will be converted to type **void*** and **nullptr** means that the stream is in a state of an error; any other value signals that the stream is “healthy”. Such a conversion is performed automatically in a context requiring a logical value. This means that this value may be used in conditional statements (**if**), and in loops **for**, **while** etc. For example, if ***strin*** is an input stream connected to a file, we can copy the file to the standard output by a simple loop:

```
char c;
while (strin.get(c)) cout << c;
```

The fact that **get** returns ****this***, i.e., the stream itself, can be used for reading in a cascade:

```
char a, b, c;
strm.get(a).get(b).get(c);
```

where three characters are read, without ignoring white characters or interpreting ***EOL*** (end of line) character in any way.

int get() — returns one character as an integer; if the end of file has been encountered, ***EOF*** will be returned. As the function does not return the stream, it cannot be used in a cascade.

istream& get(char* buf, streamsize length, char termin = '\n') — reads character into a buffer ***buf*** (which points to an array of characters). It reads at most ***length-1*** bytes: ***NUL*** character is always appended at the end. Reading terminates if end of stream is encountered, or the ***termin*** character is read, in which case it is put back to the stream (and will be the first character read by the next input operation — we can get rid of it by invoking **ignore**, described below). The default value of ***termin*** is the end of line character '\n'. The ***NUL*** character will be appended in any case. Of course, a sufficiently large buffer ***buf*** has to be allocated *before* calling the function! The second parameter is of type **streamsize**, which is an alias of an integer type. The function returns ****this***, i.e., a reference to the stream.

istream& getline(char* buf, streamsize length, char termin = '\n') — similar to the previous function, but the terminating character ***termin***, if encountered, *is* extracted form the stream, but not copied to the buffer ***buf***. This method is usually more practical than the previous one when reading data from a stream line by line.

istream& read(char* buf, streamsize length) — reads ***length*** bytes into buffer ***buf*** (which must exist and be large enough). Reading can terminate earlier only if the end of stream has been encountered. Raw bytes are read without any interpretation, the ***NUL*** character is *not* appended (the function is usually used for reading non-textual data). Function **gcount** (see below) can be used to retrieve the number of bytes actually read (which can be different than ***length*** if end of data has been encountered). The function returns ****this***, i.e., the stream.

istream& ignore(streamsize length = 1, int termin = EOF) — reads and ignores ***length*** bytes (one by default). If character ***termin*** has been encountered (end of file by default), it is extracted from the stream and reading terminates. The function returns ****this***.

int gcount( ) — returns number of bytes read by the last operation of unformatted reading.

int peek( ) — returns, as an **int**, one character from the stream, leaving it there, so it will be the first character read by the next input operation. It can be ***EOF***, if the end of stream has been encountered.

istream& putback(char c) — puts character ***c*** into the stream, so it will be the first character read by the next input operation. The operation is not always possible.

istream& unget() — puts back into the stream the last character extracted. The operation is not always possible. Returns ****this***.

Some of these methods have been used in the following program. It reads lines from the standard input and rewrites them into standard output, but ignoring comments, i.e., parts of lines which start with digraph '//':

P136: unfrd.cpp Unformatted reading

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main() {
5.      cout << "Enter lines of text. Terminate with EOF "
6.              "character\n(Ctrl-Z in Windows, Ctrl-D "
7.              "in Linux). Comments\nfrom  \'//\' to "
8.              "the end of line will be ignored.\n";
9.      char c;
10.     while ( cin.get(c) ) {
```

```
11.         if ( c == '/' )
12.             if ( cin.peek() == '/') {
13.                 cin.ignore(1024,'\n');
14.                 cout << endl;
15.                 continue;
16.             }
17.         cout << c;
18.     }
19. }
```

After reading a character, the program checks if this is a slash (line 11). If so, function **peek** is used to see if the next character is also a slash: in that case the rest of the line will be read off and ignored by **ignore** (line 13) and ***EOF*** will be output; otherwise the loop will continue. The program stops after reading an ***EOF*** character, i.e., when end of file has been encountered — this can be simulated by entering Ctrl-Z (Windows) or Ctrl-D (Linux). The conversion of the stream variable in line 10 yields **nullptr**, what terminates the loop. Running the program produces:

```
Enter lines of text. Terminate with EOF character
(Ctrl-Z in Windows, Ctrl-D in Linux). Comments
from  '//' to the end of line will be ignored.
int main(void) {
int main(void) {
    const int DIM = 15; // dimension of array
    const int DIM = 15;
    int tab[DIM];       // array of integers
    int tab[DIM];
    double x=1, y=2, z=x/y; // three doubles
    double x=1, y=2, z=x/y;
    // ...

}
}
^D
```

Note also lines 5-8 of the program. They illustrate a feature of C/C++ compilers: two literal C-strings (enclosed in double quotation marks) are automatically concatenated into one C-string if they are separated by a nonempty sequence of white characters (including the new-line character).

16.4.2 Unformatted output

Writing unformatted data is quite common. Of course, it is used for outputting data to a file or a socket, not to the computer screen. It allows us to output numbers with full machine precision in a form which saves space on disk (note that -1.234567E+19 takes 13 characters; the same **float** number with full machine precision takes only 4 bytes in memory, so only 4 bytes need to be output to a binary file). Binary (nontextual) files are not viewed or modified in editors, but can be easily read by other programs. Many files are “by definition” nontextual and all operations on them must be unformatted (sound, graphics etc.).

Two basic methods (therefore, called for a stream object) which output binary data are:

ostream& put(char c) — inserts single character (byte) ***c*** into the stream; returns ****this***, i.e., the stream.

ostream& write(const char* buf, streamsize length) — ouputs exactly ***length*** characters (bytes) from buffer (array) ***buf***; returns ****this***.

We will use these functions in examples of operations on files (next section).

Prev: 16.3 Formatting **Up:** 16. Input/Output **Next:** 16.5 Files

16.5 Files

Classes providing methods for I/O operations on files are:

- **ofstream** — derived from **ostream**;
- **ifstream** — derived from **istream**;
- **fstream** — derived from **iostream** and providing functionality for both input and output operations.

It follows from inheritance that the same methods which work for, say, ***cin*** and ***cout***, will work for file streams; in particular we can use '<<' and '>>' operators, manipulators, functions **get**, **put**, **getline** etc. All we have to do is to create a stream object connected to a file. We can do it by creating a stream object first and then opening a particular file:

```
1.    #include <fstream>
2.    // ...
3.    ofstream fil;
4.    fil.open("plik.txt");
5.    fil << "That will go to file \"fil.txt\"" << endl;
6.    fil.close();
```

On line 3 we create a stream object representing an output stream (that is why we used **ofstream** — *output file stream*). In the next line we tie the stream with a particular disk file and open it for writing. From now on, we can use the name ***fil*** as we did it with ***cout***, the difference being a destination: it was the standard output for ***cout*** while it will be a disk file for ***fil***. When we are done with the file, we have to close the stream (line 6). Note that predefined streams, like ***cin*** or ***cout***, should *not* be open or closed explicitly.

Instead of calling **open** explicitly, we can pass the name of a file directly to the constructor:

```
#include <fstream>
// ...
ofstream plik("fil.txt");
fil << "That will go to file \"fil.txt\"" << endl;
fil.close();
```

Both the constructor and **open** accept the second argument which determines the so called **opening mode** of a stream. For files which are open for reading and therefore correspond to streams of type **ifstream** (*input file stream*), this mode is by default ***ios::in*** — this is a static constant inherited from class **ios_base**. Analoguously, for files open for writing, corresponding to streams of type **ofstream** (*output file stream*), the default value of the opening mode is **ios::out**. The constants defining the opening mode can be ORed, exactly as it was for format flags. For example,

```
fstream strm("fil.txt", ios::in | ios::out);
```

creates a stream of type **fstream** in the mode *for reading and writing*.

Let us mention all predefined basic modes which can be combined when opening a file:

- ***ios::in*** — permits extraction from a stream (reading);
- ***ios::out*** — permits insertion to a stream (writing);
- ***ios::trunc*** — after creation, truncate a stream (so it becomes empty);
- ***ios::ate*** — (*at end*) after creation, go to the end of a stream;
- ***ios::app*** — go to the end of a stream before each insertion operation (writing) — appending mode;
- ***ios::binary*** — do not translate in any way the end of line character(s): treat the file as binary (in Unix/Linux this option is irrelevant, but can be significant in Windows where end of line corresponds to *two* characters);

Each stream object holds its current position within the stream (file). This is a number of type **streampos** (usually equivalent to **long**) indicating the offset (counting from 0) of a byte which will be written/read by the next I/O operation. When a stream is created, it is positioned at the beginning, i.e., the position is 0 — except, of course, when the opening mode is ***ios::ate*** or ***ios::app***, in which case the position corresponds to the byte just after the last (and its numerical value is equal to the legth of the file in bytes). After each insertion (writing) or extraction (reading), the position is modified to point to the byte which will be written/read next. Even if a stream (file) is open for both reading *and* writing, only one positions is remembered: the same for writing and reading.

We can manipulate with positioning of a stream using the following methods:

streampos tellg( ) — returns current position for reading (the letter 'g' stands for 'get'). The opening mode must include ***ios::in***.

streampos tellp( ) — returns current position for writing (the letter 'p' stands for 'put'). The opening mode must include ***ios::out***.

ostream& seekg(streampos pos) — sets the position for reading on ***pos***. Returns ****this***, i.e., the stream. The opening mode must include ***ios::in***. Positioning before the beginning or after the end puts the stream into the ***bad*** state (see below), what can be checked by '`if (strm) ...`'.

ostream& seekp(streampos pos) — as **seekg** but for writing position.

ostream& seekg(streamoff offset, ios::seek_dir pos) — sets the read position ***offset*** bytes counting from position ***pos***. Argument ***offset*** may be negative. Types **streamoff** and **ios::seek_dir** are aliases of integer types. Argument ***pos*** must be equal to one of the static constants from class **ios**:
ios::beg — beginning (of file);
ios::cur — current position;
ios::end — end (of file);
Returns ****this***, i.e., the stream. Positioning before the beginning or after the end puts the stream into the ***bad*** state.

ostream& seekp(streamoff offset, ios::seek_dir pos) — sets the write position — otherwise analogous to **seekg(streamoff offset, ios::seek_dir pos)**.

Here is an example:

P137: filerw.cpp Binary read/write operations on a file

```
1. #include <iostream>
```

```
2.  #include <fstream>
3.  using namespace std;
4.
5.  int main() {
6.      int tab[] = { 97, 105, 115, 255, 111 },k;
7.
8.      int size = sizeof(tab)/sizeof(tab[0]);
9.
10.     cout << "Array of size: " << size << endl;
11.     for (int i = 0; i < size; ++i)
12.         cout << tab[i] << " ";
13.     cout << endl;
14.
15.     ofstream file_out("file.dat",ios::out|ios::binary);
16.     if (! file_out ) {
17.         cout << "Can\'t open file_out" << endl;
18.         return -1;
19.     }
20.
21.     file_out.write((char*)tab, sizeof(tab));
22.     file_out.close();
23.
24.     fstream file("file.dat",ios::in|ios::out|ios::binary);
25.     if (! file ) {
26.         cout << "Can\'t open file" << endl;
27.         return -1;
28.     }
29.
30.     file.seekg(0,ios::end);
31.     streamsize len = file.tellg();
32.     cout << "File has length " << len << " bytes\n";
33.     file.seekg(0);
34.
35.     cout << "Bytes in file:" << endl;
36.     while ( (k = file.get()) != EOF )
37.         cout << k << " ";
38.     cout << endl;
39.
40.     file.clear(); // <-- NECESSARY !!!
41.
42.     file.seekg(4);
43.     file.read((char*)&k,4);
44.     cout << "Integer from position 4: " << k << endl;
45.
46.     file.seekp(12);
47.     file.write((char*)&k,4);
48.
49.     file.seekg(0);
50.     cout << "Bytes in file now:" << endl;
51.     while ( (k = file.get()) != EOF )
52.         cout << k << " ";
53.     cout << endl;
54.
55.     file.close();
56. }
```

A file, `file.dat`, is open for writing on line 15. In order to avoid problems in Windows, we open it in binary mode.

On line 21 we dump to the file the content of a five-element array of **int** s. This will store in `file.dat` exactly 20 bytes of the array with bit representation of five numbers. We then close the file and open it again, this time for reading and writing (line 24). After opening, the file is positioned at the end and current position is retrieved (lines 30-32). It should correspond to the “first after the last” byte; as the last one has index 19, the position is now 20, what is equal to the length of the file in bytes.

Next we rewind the file back to the beginning (line 33) and read it byte by byte outputting its content to the standard output stream (lines 36-38).

After that operation, the stream is in the state ***bad***, because at the end of the loop we

attempted to read something when there was nothing more to read. This means that from now on any I/O operation on this stream will be simply ignored *without* signaling an error! Therefore, we have to “repair” the stream — this is what **clear** from line 40 does (more details below).

We now go to the byte number 4 (i.e., the fifth, the first byte of the second number from the array). Function **seekg** is used, because we want to read from this position. On line 43 four bytes are read and copied to four bytes occupied by an integer variable ***k***. After that, the value of ***k*** should be 105. Now we do sometging opposite: we go to byte number 12, which is the first byte of the fourth number. This time, function **seekp** is used, because we want to write at this position. On line 47, four bytes representing the variable ***k*** are copied to the file erasing the previous content of four bytes, which represented the fourth number form the original array. Outputting again the content of the file we can check that the fourth number has indeed been modified:

```
Array of size: 5
97 105 115 255 111
File has length 20 bytes
Bytes in file:
97 0 0 0 105 0 0 0 115 0 0 0 255 0 0 0 111 0 0 0
Integer from position 4: 105
Bytes in file now:
97 0 0 0 105 0 0 0 115 0 0 0 105 0 0 0 111 0 0 0
```

Note that the order of bytes representing each number depends on our computer's architecture: in this case it was ***little-endian***. On a ***big-endian*** computer, the order would be different; e.g., bytes of the first number would be '0 0 0 97', and not, as in our example, '97 0 0 0'.

Prev: 16.4 Unformatted I/O operations **Up:** 16. Input/Output **Next:** 16.6 Handling stream errors

16.6 Handling stream errors

Each stream has associated with it **stream state information** (stream state word). It is an integer variable (of type **iostate**) which holds information on the state of a stream, in particular about errors caused by I/O operations on this stream.

There are several methods which can be used to access the state of a stream. It is also possible to check the state of a stream indirectly, in logical conditions of **if**, **for**, **while** statements. In such cases, the value of a stream variable will be converted automatically to **bool** or **void***, due to overloading of '!' and conversion operators in class **ios**.

In expressions like

```
if ( strm ) ...
```

the value of ***strm*** will be converted to empty pointer **nullptr** (what will correspond to **false**), if the state of the stream ***strm*** is ***bad***, i.e., if method **fail** would return **true** (see below). Otherwise, the value will be non-zero, what will correspond to **true**. Similarly, in expressions like

```
if ( !strm ) ...
```

the '!' operator applied to a stream will return the logical value **true** if, and only if, method **fail** would return **true**.

Stream state word contains, however, more information. As the format flag, it can be composed and decomposed by ORing it with several predefined constants of type **iostate**.

ios::badbit — Stream is corrupted. It is not known if the last I/O operation succeded; next one will fail.

ios::eofbit — There was an attempt to access data past the end of stream (file).

ios::failbit — Stream is corrupted. The last I/O operation succeded; next one will fail.

ios::goodbit — Stream is “healthy” (the value of ***goodbit*** jest 0).

The state of a stream can be checked by methods of class **ios**:

- **bool bad( )**
- **bool eof( )**
- **bool fail( )**
- **bool good( )**

which return, as logical value, settings of bits corresponding to ***ios::badbit***, ***ios::eofbit***, ***ios::failbit*** and ***ios::goodbit*** in the stream state word.

The state word of a stream can be read and modified with the help of methods of class **ios**:

iostate rdstate() — Returns the stream state word as a value of type **iostate**.

void clear(iostate state = ios::goodbit) — Clears the stream state word setting its value to 0, what corresponds to ***good***. Then it sets bit ***state***, which has to be one of ***ios::badbit***, ***ios::failbit*** etc. Default value of ***state*** is ***ios::goodbit***, therefore invoking **clear** without argument "repairs" a stream (see an example in `filerw.cpp`).

void setstate(iostate state) — Adds (by ORing) ***state*** (one of ***ios::badbit***, ***ios::failbit*** etc.) to stream state word. Equivalent to '`clear( rdstate() | ios::stte )`'.

Generally, state of a stream should be checked after every I/O operation. If it is "no good", i.e., when **fail**, **bad** or **eof** returns **true**, next I/O operation will be *ignored*, unless the state is first repaired. Let us consider the following important example:

P138: validat.cpp Validating input data

```
 1. #include <fstream>
 2. #include <iostream>
 3. using namespace std;
 4.
 5. int main() {
 6.     const int DIM = 80;
 7.     char      name[DIM];
 8.     ifstream  infile;
 9.     double    x;
10.
11.     do {
12.         cout << "Name of input file: ";
13.         cin.getline(name, DIM);
14.
15.         infile.clear();
16.         infile.open(name);
17.     } while (!infile);
18.
19.     cout << "File = " << name << endl;
20.     infile.close();
21.
22.     do {
23.         if (!cin) {
24.             // order important!
25.             cin.clear();
26.             cin.ignore(1024,'\n');
27.         };
28.         cout << "Enter a number: ";
29.         cin >> x;
30.     } while (!cin);
31.
32.     cout << "Number = " << x << endl;
33. }
```

We read (line 13) a name of a file. Then we try to open it (line 16). If the file does not exist, stream ***infile*** is in ***bad*** state and the condition in **do...while** loop is met; the program will proceed to next cycle of the loop, asking for a name again. However, the state of stream is still ***bad***, so we repair it before the next attempt (line 15). What is important is the fact that the program leaves the loop only if opening the file succeeded and ***infile*** is "healthy".

The loop on lines 22-30 is more subtle. Its task is to read a number on ***x***. The program attempts to read it on line 29. Suppose that the user made a mistake, entering, e.g., letters instead of digits. Then ***cin*** will become ***bad***. The program will proceed to the next cycle of the loop. But the state is ***bad***, so all I/O operations will be ignored. Therefore, we have to

repair the stream, what we do on line 25. However, this is not enough! Wrong data has not been extracted from the stream, because an error occurred. Therefore, next reading will read this “rubbish” again, even if this time the user entered a correct number. Therefore, the reading will fail again, and again, and again forever! Hence, we have to get rid of the “rubbish”. That is why we have to add invocation of **ignore** on line 26. Now the program behaves correctly (we assume that a file `val.dat` exists in the current catalog):

```
Name of input file: val.txt
Name of input file: val
Name of input file: val.dat
File = val.dat
Enter a number: s12
Enter a number: @
Enter a number: 12
Number = 12
```

Note that the order of repairing and removing the rubbish (clearing and cleaning) *is* important (lines 25 and 26). Suppose the order is reversed. Then **ignore** is invoked *before* the stream is repaired, and is ignored, because it is an I/O operation and the stream is ***bad***. The rubbish remains in the stream and we have an infinite loop again:

```
Name of input file: val.dat
File = val.dat
Enter a number: s12
Enter a number: Enter a number: Enter a number: En
ter a number: Enter a number: Enter a number: Ente
```

where the execution had to be terminated by pressing Ctrl-C.

Handling all possible I/O errors is an extremely difficult and time consuming task — it can happen that the necessary code constitutes more than a half of the whole program!

Subsections

- 16.7.1 C-strings as internal files
- 16.7.2 Internal files represented by C++ strings

16.7 Internal files

It is possible to define a “file” which does not exist on a disk, but corresponds to a region of memory to which and from which data can be transferred, exactly as to/from a disk file. Such “files” are called **internal files**. Technically, the rôle of an internal file can be played by C-strings (arrays of characters) or by objects of class **string**, specific to C++.

16.7.1 C-strings as internal files

A C-string, as we know, is a contiguous region of memory which is treated as an array of characters, but the same region can also be treated as an internal file.

In order to be able to use such files, one has to include the header `strstream` which provides classes **istrstream**, which represents an input stream, and **ostrstream**, representing an output stream.

This header has been removed from the standard, but is implemented by all C++ compilers for backward compatibility. You can get some warnings when compiling the program!

An object of class **ostrstream** can be created in two ways:

- `ostrstream strm;` — without any argument for the constructor. Streams created in this way can accept arbitrary large amount of data. As with “normal” files, one can use manipulators, formatting etc. When we are done with writing to the file, we can call parameterless method **str**, which returns a pointer of type **char*** pointing to the beginning of the resulting array of characters. If we want to treat it as a legal C-string (what is not always required), we have to ensure that the last character which has been output to the stream is '\0', e.g., by inserting it manually: `strm << ends`. The array has been allocated on the heap by **malloc**; therefore, if we do not need it any more, we should deallocate it by calling **free**, *not* **delete[]** (see section on memory management). There is no method **close**.
- `ostrstream strm(char* tab, size_t len);` — data will be transferred to character array ***tab*** of a fixed length, determined by the second argument ***len*** (of type **size_t**, which is an integer type). When a stream is open, we can use operations, like << operator, functions **write**, **put** etc., as with normal files. It can happen that we attempt to write past the end of the character array associated with the stream: in such case the last character (with index ***len-1***) will be set to ***NUL*** (i.e., '\0'), the state of the

stream will be set to ***bad***, and all subsequent output operations will be ignored (so no overflow should happen).

An example:

P139: intfilout.cpp C-string as an internal file

```
1.  #include <iostream>
2.  #include <strstream>
3.  #include <cstdlib>   // free
4.  using namespace std;
5.
6.  int main() {
7.      // "rubber" version
8.      ostrstream napis1;
9.      napis1 << "Beginning, " << "continuation, "
10.            << "end."        << ends;
11.     char* n = napis1.str();
12.     cout << "The string is: " << n << endl;
13.     free(n);
14.
15.     // version with array
16.     char tab[30];
17.     ostrstream napis2(tab,sizeof(tab));
18.     napis2 << "Maggie " << "Kathy " << "Mary" << ends;
19.     cout << tab << endl;
20. }
```

The program prints:

```
The string is: Beginning, continuation, end.
Maggie Kathy Mary
```

Note that we do not allocate memory for the C-string ***n*** — this will be done automatically. However, we have to deallocate it ourselves (line 13).

In a similar way, one can read from a character array by creating an object of class **istrstream**.

- `istrstream strm(char* t, size_t len);` — the file is a region of memory of length ***len*** starting from byte pointed to by ***t***.
- `istrstream strm(char* t);` — the "file" is really a region of memory starting from byte pointed to by ***t*** and ending at ***NUL*** character, which is treated as the end-of-file character.

Internal files based on C-strings are sometimes used to represent data from a disk file, especially when we want to "jump" between different parts of the file. This is a rather expensive operation, as it involves many accesses to disk. It may be then advategous to read the whole file into a character array and then treat it as an internal file.

16.7.2 Internal files represented by C++ strings

A safer, comparing to C-strings, implementation of strings is provided in C++ by the class **string** (described in the next chapter). Objects of this class can also play rôle of internal files.

In order to be able to use **string** s as internal files, one has to include the header `sstream`, which provides acces to classes **istringstream** and **ostringstream**.

An output stream can be created with the default constructor

```
ostringstream strm;
```

Then the object created can be trated as a file:

```
strm << "To be" << " or not to be" << ", etc.";
```

We do not have to care about overflowing: memory will be allocated automatically as needed. When we are done with the file, we can get the resulting string by calling the method **str**, which returns an object of class **string** with the content of the internal file:

```
string s = strm.str();
cout << s << endl;
```

It is also possible to create a file with some initial content

```
ostringstream ostr("Something ", ios::ate);
ostr << "another something";
```

and then insert additional data into it, as in the example above.

Any existing object of class **string** can be open as an internal input file; an example can be found in the program below:

P140: intstr.cpp C++ strings as internal files

```
1.  #include <iostream>
2.  #include <sstream>
3.  using namespace std;
4.
5.  void words(const string& s) {
6.      istringstream istr(s);
7.
8.      string word;
9.      while ( istr >> word )
10.         cout << word << endl;
11. }
12.
13. int main() {
14.     string s = "Bach Haydn";
15.     ostringstream ostr(s, ios::ate);
16.     ostr << " Chopin";
17.     string s1 = ostr.str();
18.     words(s1);
19. }
```

We open an output internal file initialized with the content of string ***s*** (line 15). Then we add some data and retrieve the resulting file to string ***s1***. This string is then passed to function **words**, which opens it as an input internal file (line 6). We read from this file space separated words according to normal rules of reading from an input stream. Note that after reaching the end of data, the state of the stream becomes ***bad***, as it should, what terminates the loop from lines 9-10. The program prints

```
Bach
Haydn
Chopin
```

We will show now how to conveniently read and write text files. Let us look at the example:

P141: RWfile.cpp Reading and writing text files

```
1.  #include <iostream>
2.  #include <fstream>   // ifstream, ofstream
3.  #include <string>
4.  #include <sstream>   // stringstream
5.  using namespace std;
6.
7.  int main() {
8.      string line{};
9.
10.     ofstream outf{"RWfile.out"};                          ❶
11.     for (ifstream in{"RWfile.dat"}; getline(in, line);) {  ❷
12.         cout << line  << "\n";
13.         string name;
14.         int height;
15.         double weight;
16.         istringstream str{line};                          ❸
17.         str >> name >> height >> weight;
18.         cout << name << ": height=" << height
19.              << ", weight=" << weight << '\n';
20.         outf << name << ": height=" << height             ❹
21.              << ", weight=" << weight << '\n';
22.     }
23.     outf.close();
24.
25.       // again but in a while loop
26.     ifstream in{"RWfile.dat"};
27.     while (getline(in, line)) {
28.         cout << line  << "\n";
29.     }
30. }
```

In line ❶, we create an object ***outf*** of type **ofstream** ('of' comes from *output stream*). This is an object representing an output stream, like ***cout*** but associated with file `RWfile.out` (which will be created, if it doesn't exist). As can be seen in line ❹, we use ***outf*** exactly like ***cout***, but the text goes into the file instead of the screen. In line ❷, we create object ***in*** of type **ifstream** — this is input stream, like ***cin*** but which takes data from a file instead of the keyboard. The function **getline** (from `string` header) takes an input stream and a string (***line***, in our case), reads one line from the stream and puts it into ***line***. It returns its first argument (an object representing our input stream) and this is converted to a logical value yielding **false** when the end of file has been encountered.
The object ***str*** (❸) is of type **istringstream** (from `sstream` header) and it represents an input stream taking data from the string passed to the constructor (***line***, in our case). As we can see, we use it like ***cin*** to read values from the ***line***.
With data file `RWfile.dat` containing

```
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

the output on the screen and in the file `RWfile.out` will be

```
Mary 167 56.5
Mary: height=167, weight=56.5
Jane 162 55.7
Jane: height=162, weight=55.7
Kate 170 59.1
Kate: height=170, weight=59.1
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

Subsections

- 16.7.1 C-strings as internal files
- 16.7.2 Internal files represented by C++ strings

16.7 Internal files

It is possible to define a “file” which does not exist on a disk, but corresponds to a region of memory to which and from which data can be transferred, exactly as to/from a disk file. Such “files” are called **internal files**. Technically, the rôle of an internal file can be played by C-strings (arrays of characters) or by objects of class **string**, specific to C++.

16.7.1 C-strings as internal files

A C-string, as we know, is a contiguous region of memory which is treated as an array of characters, but the same region can also be treated as an internal file.

In order to be able to use such files, one has to include the header `strstream` which provides classes **istrstream**, which represents an input stream, and **ostrstream**, representing an output stream.

This header has been removed from the standard, but is implemented by all C++ compilers for backward compatibility. You can get some warnings when compiling the program!

An object of class **ostrstream** can be created in two ways:

- `ostrstream strm;` — without any argument for the constructor. Streams created in this way can accept arbitrary large amount of data. As with “normal” files, one can use manipulators, formatting etc. When we are done with writing to the file, we can call parameterless method **str**, which returns a pointer of type **char*** pointing to the beginning of the resulting array of characters. If we want to treat it as a legal C-string (what is not always required), we have to ensure that the last character which has been output to the stream is '\0', e.g., by inserting it manually: `strm << ends`. The array has been allocated on the heap by **malloc**; therefore, if we do not need it any more, we should deallocate it by calling **free**, *not* **delete[]** (see section on memory management). There is no method **close**.
- `ostrstream strm(char* tab, size_t len);` — data will be transferred to character array ***tab*** of a fixed length, determined by the second argument ***len*** (of type **size_t**, which is an integer type). When a stream is open, we can use operations, like << operator, functions **write**, **put** etc., as with normal files. It can happen that we attempt to write past the end of the character array associated with the stream: in such case the last character (with index ***len-1***) will be set to ***NUL*** (i.e., '\0'), the state of the

stream will be set to ***bad***, and all subsequent output operations will be ignored (so no overflow should happen).

An example:

P139: intfilout.cpp C-string as an internal file

```
1.  #include <iostream>
2.  #include <strstream>
3.  #include <cstdlib>   // free
4.  using namespace std;
5.
6.  int main() {
7.      // "rubber" version
8.      ostrstream napis1;
9.      napis1 << "Beginning, " << "continuation, "
10.            << "end."        << ends;
11.     char* n = napis1.str();
12.     cout << "The string is: " << n << endl;
13.     free(n);
14.
15.     // version with array
16.     char tab[30];
17.     ostrstream napis2(tab,sizeof(tab));
18.     napis2 << "Maggie " << "Kathy " << "Mary" << ends;
19.     cout << tab << endl;
20. }
```

The program prints:

```
The string is: Beginning, continuation, end.
Maggie Kathy Mary
```

Note that we do not allocate memory for the C-string ***n*** — this will be done automatically. However, we have to deallocate it ourselves (line 13).

In a similar way, one can read from a character array by creating an object of class **istrstream**.

- `istrstream strm(char* t, size_t len);` — the file is a region of memory of length ***len*** starting from byte pointed to by ***t***.
- `istrstream strm(char* t);` — the "file" is really a region of memory starting from byte pointed to by ***t*** and ending at ***NUL*** character, which is treated as the end-of-file character.

Internal files based on C-strings are sometimes used to represent data from a disk file, especially when we want to "jump" between different parts of the file. This is a rather expensive operation, as it involves many accesses to disk. It may be then advategous to read the whole file into a character array and then treat it as an internal file.

16.7.2 Internal files represented by C++ strings

A safer, comparing to C-strings, implementation of strings is provided in C++ by the class **string** (described in the next chapter). Objects of this class can also play rôle of internal files.

In order to be able to use **string** s as internal files, one has to include the header `sstream`, which provides acces to classes **istringstream** and **ostringstream**.

An output stream can be created with the default constructor

```
ostringstream strm;
```

Then the object created can be trated as a file:

```
strm << "To be" << " or not to be" << ", etc.";
```

We do not have to care about overflowing: memory will be allocated automatically as needed. When we are done with the file, we can get the resulting string by calling the method **str**, which returns an object of class **string** with the content of the internal file:

```
string s = strm.str();
cout << s << endl;
```

It is also possible to create a file with some initial content

```
ostringstream ostr("Something ", ios::ate);
ostr << "another something";
```

and then insert additional data into it, as in the example above.

Any existing object of class **string** can be open as an internal input file; an example can be found in the program below:

P140: intstr.cpp C++ strings as internal files

```
1.  #include <iostream>
2.  #include <sstream>
3.  using namespace std;
4.
5.  void words(const string& s) {
6.      istringstream istr(s);
7.
8.      string word;
9.      while ( istr >> word )
10.         cout << word << endl;
11. }
12.
13. int main() {
14.     string s = "Bach Haydn";
15.     ostringstream ostr(s, ios::ate);
16.     ostr << " Chopin";
17.     string s1 = ostr.str();
18.     words(s1);
19. }
```

We open an output internal file initialized with the content of string ***s*** (line 15). Then we add some data and retrieve the resulting file to string ***s1***. This string is then passed to function **words**, which opens it as an input internal file (line 6). We read from this file space separated words according to normal rules of reading from an input stream. Note that after reaching the end of data, the state of the stream becomes ***bad***, as it should, what terminates the loop from lines 9-10. The program prints

```
Bach
Haydn
Chopin
```

We will show now how to conveniently read and write text files. Let us look at the example:

P141: RWfile.cpp Reading and writing text files

```
1.  #include <iostream>
2.  #include <fstream>   // ifstream, ofstream
3.  #include <string>
4.  #include <sstream>   // stringstream
5.  using namespace std;
6.
7.  int main() {
8.      string line{};
9.
10.     ofstream outf{"RWfile.out"};                          ❶
11.     for (ifstream in{"RWfile.dat"}; getline(in, line);) {  ❷
12.         cout << line  << "\n";
13.         string name;
14.         int height;
15.         double weight;
16.         istringstream str{line};                          ❸
17.         str >> name >> height >> weight;
18.         cout << name << ": height=" << height
19.              << ", weight=" << weight << '\n';
20.         outf << name << ": height=" << height             ❹
21.              << ", weight=" << weight << '\n';
22.     }
23.     outf.close();
24.
25.       // again but in a while loop
26.     ifstream in{"RWfile.dat"};
27.     while (getline(in, line)) {
28.         cout << line  << "\n";
29.     }
30. }
```

In line ❶, we create an object ***outf*** of type **ofstream** ('of' comes from *output stream*). This is an object representing an output stream, like ***cout*** but associated with file `RWfile.out` (which will be created, if it doesn't exist). As can be seen in line ❹, we use ***outf*** exactly like ***cout***, but the text goes into the file instead of the screen. In line ❷, we create object ***in*** of type **ifstream** — this is input stream, like ***cin*** but which takes data from a file instead of the keyboard. The function **getline** (from `string` header) takes an input stream and a string (***line***, in our case), reads one line from the stream and puts it into ***line***. It returns its first argument (an object representing our input stream) and this is converted to a logical value yielding **false** when the end of file has been encountered.
The object ***str*** (❸) is of type **istringstream** (from `sstream` header) and it represents an input stream taking data from the string passed to the constructor (***line***, in our case). As we can see, we use it like ***cin*** to read values from the ***line***.
With data file `RWfile.dat` containing

```
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

the output on the screen and in the file `RWfile.out` will be

```
Mary 167 56.5
Mary: height=167, weight=56.5
Jane 162 55.7
Jane: height=162, weight=55.7
Kate 170 59.1
Kate: height=170, weight=59.1
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

Subsections

- 16.7.1 C-strings as internal files
- 16.7.2 Internal files represented by C++ strings

16.7 Internal files

It is possible to define a “file” which does not exist on a disk, but corresponds to a region of memory to which and from which data can be transferred, exactly as to/from a disk file. Such “files” are called **internal files**. Technically, the rôle of an internal file can be played by C-strings (arrays of characters) or by objects of class **string**, specific to C++.

16.7.1 C-strings as internal files

A C-string, as we know, is a contiguous region of memory which is treated as an array of characters, but the same region can also be treated as an internal file.

In order to be able to use such files, one has to include the header `strstream` which provides classes **istrstream**, which represents an input stream, and **ostrstream**, representing an output stream.

This header has been removed from the standard, but is implemented by all C++ compilers for backward compatibility. You can get some warnings when compiling the program!

An object of class **ostrstream** can be created in two ways:

- `ostrstream strm;` — without any argument for the constructor. Streams created in this way can accept arbitrary large amount of data. As with “normal” files, one can use manipulators, formatting etc. When we are done with writing to the file, we can call parameterless method **str**, which returns a pointer of type **char*** pointing to the beginning of the resulting array of characters. If we want to treat it as a legal C-string (what is not always required), we have to ensure that the last character which has been output to the stream is '\0', e.g., by inserting it manually: `strm << ends`. The array has been allocated on the heap by **malloc**; therefore, if we do not need it any more, we should deallocate it by calling **free**, *not* **delete[]** (see section on memory management). There is no method **close**.
- `ostrstream strm(char* tab, size_t len);` — data will be transferred to character array ***tab*** of a fixed length, determined by the second argument ***len*** (of type **size_t**, which is an integer type). When a stream is open, we can use operations, like << operator, functions **write**, **put** etc., as with normal files. It can happen that we attempt to write past the end of the character array associated with the stream: in such case the last character (with index ***len-1***) will be set to ***NUL*** (i.e., '\0'), the state of the

stream will be set to ***bad***, and all subsequent output operations will be ignored (so no overflow should happen).

An example:

P139: intfilout.cpp C-string as an internal file

```
1.  #include <iostream>
2.  #include <strstream>
3.  #include <cstdlib>   // free
4.  using namespace std;
5.
6.  int main() {
7.      // "rubber" version
8.      ostrstream napis1;
9.      napis1 << "Beginning, " << "continuation, "
10.            << "end."        << ends;
11.     char* n = napis1.str();
12.     cout << "The string is: " << n << endl;
13.     free(n);
14.
15.     // version with array
16.     char tab[30];
17.     ostrstream napis2(tab,sizeof(tab));
18.     napis2 << "Maggie " << "Kathy " << "Mary" << ends;
19.     cout << tab << endl;
20. }
```

The program prints:

```
The string is: Beginning, continuation, end.
Maggie Kathy Mary
```

Note that we do not allocate memory for the C-string ***n*** — this will be done automatically. However, we have to deallocate it ourselves (line 13).

In a similar way, one can read from a character array by creating an object of class **istrstream**.

- `istrstream strm(char* t, size_t len);` — the file is a region of memory of length ***len*** starting from byte pointed to by ***t***.
- `istrstream strm(char* t);` — the "file" is really a region of memory starting from byte pointed to by ***t*** and ending at ***NUL*** character, which is treated as the end-of-file character.

Internal files based on C-strings are sometimes used to represent data from a disk file, especially when we want to "jump" between different parts of the file. This is a rather expensive operation, as it involves many accesses to disk. It may be then advategous to read the whole file into a character array and then treat it as an internal file.

16.7.2 Internal files represented by C++ strings

A safer, comparing to C-strings, implementation of strings is provided in C++ by the class **string** (described in the next chapter). Objects of this class can also play rôle of internal files.

In order to be able to use **string** s as internal files, one has to include the header `sstream`, which provides acces to classes **istringstream** and **ostringstream**.

An output stream can be created with the default constructor

```
ostringstream strm;
```

Then the object created can be trated as a file:

```
strm << "To be" << " or not to be" << ", etc.";
```

We do not have to care about overflowing: memory will be allocated automatically as needed. When we are done with the file, we can get the resulting string by calling the method **str**, which returns an object of class **string** with the content of the internal file:

```
string s = strm.str();
cout << s << endl;
```

It is also possible to create a file with some initial content

```
ostringstream ostr("Something ", ios::ate);
ostr << "another something";
```

and then insert additional data into it, as in the example above.

Any existing object of class **string** can be open as an internal input file; an example can be found in the program below:

P140: intstr.cpp C++ strings as internal files

```
1.  #include <iostream>
2.  #include <sstream>
3.  using namespace std;
4.
5.  void words(const string& s) {
6.      istringstream istr(s);
7.
8.      string word;
9.      while ( istr >> word )
10.         cout << word << endl;
11. }
12.
13. int main() {
14.     string s = "Bach Haydn";
15.     ostringstream ostr(s, ios::ate);
16.     ostr << " Chopin";
17.     string s1 = ostr.str();
18.     words(s1);
19. }
```

We open an output internal file initialized with the content of string ***s*** (line 15). Then we add some data and retrieve the resulting file to string ***s1***. This string is then passed to function **words**, which opens it as an input internal file (line 6). We read from this file space separated words according to normal rules of reading from an input stream. Note that after reaching the end of data, the state of the stream becomes ***bad***, as it should, what terminates the loop from lines 9-10. The program prints

```
Bach
Haydn
Chopin
```

We will show now how to conveniently read and write text files. Let us look at the example:

P141: RWfile.cpp Reading and writing text files

```
1.  #include <iostream>
2.  #include <fstream>   // ifstream, ofstream
3.  #include <string>
4.  #include <sstream>   // stringstream
5.  using namespace std;
6.
7.  int main() {
8.      string line{};
9.
10.     ofstream outf{"RWfile.out"};                          ❶
11.     for (ifstream in{"RWfile.dat"}; getline(in, line);) {  ❷
12.         cout << line  << "\n";
13.         string name;
14.         int height;
15.         double weight;
16.         istringstream str{line};                          ❸
17.         str >> name >> height >> weight;
18.         cout << name << ": height=" << height
19.              << ", weight=" << weight << '\n';
20.         outf << name << ": height=" << height             ❹
21.              << ", weight=" << weight << '\n';
22.     }
23.     outf.close();
24.
25.       // again but in a while loop
26.     ifstream in{"RWfile.dat"};
27.     while (getline(in, line)) {
28.         cout << line  << "\n";
29.     }
30. }
```

In line ❶, we create an object ***outf*** of type **ofstream** ('of' comes from *output stream*). This is an object representing an output stream, like ***cout*** but associated with file `RWfile.out` (which will be created, if it doesn't exist). As can be seen in line ❹, we use ***outf*** exactly like ***cout***, but the text goes into the file instead of the screen. In line ❷, we create object ***in*** of type **ifstream** — this is input stream, like ***cin*** but which takes data from a file instead of the keyboard. The function **getline** (from `string` header) takes an input stream and a string (***line***, in our case), reads one line from the stream and puts it into ***line***. It returns its first argument (an object representing our input stream) and this is converted to a logical value yielding **false** when the end of file has been encountered.
The object ***str*** (❸) is of type **istringstream** (from `sstream` header) and it represents an input stream taking data from the string passed to the constructor (***line***, in our case). As we can see, we use it like ***cin*** to read values from the ***line***.
With data file `RWfile.dat` containing

```
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

the output on the screen and in the file `RWfile.out` will be

```
Mary 167 56.5
Mary: height=167, weight=56.5
Jane 162 55.7
Jane: height=162, weight=55.7
Kate 170 59.1
Kate: height=170, weight=59.1
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

17. Strings

We have been using strings in our programs since the first chapters, but only in the simplest way, without delving into the details. In this chapter, we will have a closer look at strings in both C and C++.

Subsections

- 17.1 C-strings
  - 17.1.1 Functions on C-strings
  - 17.1.2 Functions operating on characters
  - 17.1.3 Conversion functions
- 17.2 Strings in C++
  - 17.2.1 Constructors
  - 17.2.2 Methods and operators

Subsections

- 17.1.1 Functions on C-strings
- 17.1.2 Functions operating on characters
- 17.1.3 Conversion functions

17.1 C-strings

In traditional C, types **char** and **char*** are treated as generic types often used for referring to raw memory. Therefore, it is not strange that many functions acting on C-strings are very similar to those acting on memory (see section on memory operations in C).

A C-string is just an array of characters; the only requirement is that it should end with a ***NUL*** character — the character with ASCII code equal to 0, whose literal is therefore '\0' — ***NUL*** is just a generally accepted name of this character.

One should remember that

> **a C-string of type char* initialized with a string literal (enclosed in double quotation marks) is *unmodifiable*.**

If we want a C-string to be modifiable, we have to define it explicitly as a character array:

```
1.    char* s1   = "Joe";  // Unmodifiable, dimension 4
2.    char  s2[] = "Joe";  // Modifiable,   dimension 4
3.    char  s3[] = {'J', 'o', 'e', '\0'}       // As s2
```

The two last definitions are equivalent, but the form used in the second line is definitly more convenient. Note that forms from the first two lines make the compiler add the terminating ***NUL*** character itself — in the third form, we have to do it explicilty. In all cases the array will contain four characters: three letters and the ***NUL***. Arrays created in last two lines are normal, local arrays allocated on the stack. However, the array from line 1 will be created elsewhere and *cannot* be modified. It is extremely important to remeber that, especially when passing such tables (C-strings) to functions. The type of ***s1*** is **char***, so the compiler will not object to pass it to a function which does *not* declare the corresponding parameter as **const char***. Nevertheless, an attempt to modify the array inside the function will cause the crash of the program! Using the form from the first line, it is better, although not required by the standard, to declare such pointers as **const char***. Then the compiler will check if we do not try to pass an unmodifiable C-string to a function that does not promise that it will not alter it (as we remember, a function can make such a promise by declaring the correponding parameter as **const char***).

The same applies to literals passed as arguments to functions with a parameter of type **char*** — like invocation `func("John")`. Although the parameter have been declared as modifiable C-string (**char***), any attempt to alter this string inside the function will crash the program!

There are several standard functions operating on C-strings. We can access them by including the header `cstring`. Functions operating on single characters come from the header `cctype`, while some useful conversion functions can be made available by including `cstdlib`.

17.1.1 Functions on C-strings

The functions mentioned below come from the header `cstring`(`string.h` in C) and can be used in both C and C++.

size_t strlen(const char* s) — returns the length of a C-string ***s*** in characters *excluding* the terminating ***NUL***. Type **size_t** is an integer type.

char* strcat(char* dest, const char* src) — appends string ***src*** to ***dest*** (concatenation); returns ***dest***. Both ***dest*** and ***src*** have to be ***NUL*** terminated C-strings. The user is responsible for allocating a segment of memory starting at address pointed to by ***dest*** that is sufficiently large to hold both strings after concatenation. The first ***NUL*** character in ***dest*** is overwritten by the first character of ***src*** followed by the remaining characters of ***src***, including the terminating ***NUL***. It is a common mistake to concatenate a string with an uninitialized string (which does not end with ***NUL***)

```
char s[100];
strcat(s,"Flower");
```

what is a mistake: the program will look for a ***NUL*** in ***s***, but will not find any, because array ***s*** has not been initialized.

char* strncat(char* dest, const char* src, size_t n) — is similar to the previous function, but copies at most ***n*** characters of ***src***. Copying terminates when a ***NUL*** character is encountered and copied. If ***n*** characters have been copied and ***NUL*** has not been found, it will be appended as the ***n+1***-th character. Note that in this case ***n+1*** characters will be added to ***dest***! For example

```
char t1[20] = {'\0'}; // NUL necessary!
strncat(t1,"123456789",5);
cout << t1 << " has " << strlen(t1) << " chars" << endl;
```

will print '12345 has 5 chars' (five characters have been copied, no ***NUL*** has been found, so it was appended as the sixth character).

char* strcpy(char* dest, const char* src) — copies characters from the segment of memory starting at address pointed to by ***src*** to the segment pointed to by ***dest***. The destination region is overwritten. Copying terminates when a ***NUL*** has been copied; the user is responsible for the presence of ***NUL*** in the string being copied and for allocating enough space at address ***dest***. The function returns ***dest***. For example, function **strcat** could be implemented in the following way

```
char* strcat(char* dest, const char* src) {
    char* s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

by using **strcpy** and **strlen**.

char* strncpy(char* dest, const char* src, size_t n) — copies exactly ***n*** characters from location starting at the address pointed to by ***src*** to the one pointed to by ***dest***. When ***NUL*** is encounterd, it is copied and then ***NUL*** characters are inserted into the destination string as padding until exactly ***n*** characters have been written. Note that if the length of ***src*** is equal or bigger than ***n***, the terminating ***NUL*** will *not* be copied.

int strcmp(const char* s1, const char* s2) — compares lexicographically (in dictionary order) two strings. Returns -1, if ***s1*** is less (earlier in dictionary order) than ***s2***, +1, if ***s2*** is less, and 0 if they are equal. A string ***s1*** is considered less than ***s2*** if one of the following condition is met:
a) both string are equal up to a certain position, and at the first differing position the character from ***s1*** has value which is less than the value of the character in ***s2*** at the same position;
b) the string ***s1*** is shorter than ***s2*** and has all characters equal to characters from ***s2*** at the corresponding positions.
For example

```
if ( ! strcmp(s1,s2) )
    cout << "Identical\n";
else
    cout << "Different\n";
```

will work, as any nonzero value returned by **strcmp** will be treated as **true** and, after negating with '!' operator, as **false**.
The example below illustrates the use of **strcmp** function. The function **first_last** (lines 20-26) takes a C-string and two pointers of type **char*** — by reference, so they can be modified. It then puts into pointer ***p*** the address of the beginning of a name (an element of the array of strings) which is lexicographically the least, and into ***q*** the address of the lexicographically last name.

P142: sorword.cpp Comparing C-strings

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  void  first_last(char**,char*&,char*&);
6.
7.  int main() {
8.
9.      char *nam[] = { "Cathy",      "Maggi",
10.                      "Alice",      "Wanda",
11.                      "Wendy", "Catharina", "" },
12.                    *p,*q;
13.
14.     first_last(nam,p,q);
15.
16.     cout << "First: " << p << endl
17.          << "Last : " << q << endl;
18. }
19.
20. void first_last(char** s, char*& p, char*& q) {
21.     p = q = *s;
22.     while ( **++s ) {
23.         if ( strcmp(*s, p) < 0 ) p = *s;
24.         if ( strcmp(*s, q) > 0 ) q = *s;
25.     }
26. }
```

Note that the dimension of the array has not been passed to the function. Instead, the last element of the array is an empty string, containing only ***NUL*** character, which will be supllied by the comopiler (line 11). This technique is very often used in C library (for arrays of C-strings). The code of the function **first_last** is rather compact: it could have been longer and easier to comprehend, but its analysis in the form presented is a good excersise on pointers. The program prints 'First: Alice Last : Wendy'.

int strcoll(const char* s1, const char* s2) — as **strcmp**, by takes locale information into account. For example, in Polish locale the letter 'a' is earlier than 'ą', while in French locale diacritic marks do no count in ordering, so 'â' is equivalent to plain 'a'.

int strncmp(const char* s1, const char* s2, size_t n) — as **strcmp**, but only first ***n*** characters are taken into account.

char* strchr(const char* s, int c) — searches for the first occurrence of character `(char)c` in a string ***s*** and returns a pointer to its position or **nullptr** if this character does dot occur in ***s***. For example,

```
cout << strchr("Daniel Defoe",'f') << endl;
```

will print 'foe'. A function counting the number of occurrences of `(char)c` in a string ***s*** could have the form

```
int count(const char* s, int c) {
    int n = 0;
    while (s = strchr(s,c)) ++s,++n;
    return n;
}
```

In particular, one *can* use **strchr** to look for ***NUL*** character.

char* strrchr(const char* s, int c) — as **strchr**, but returns a pointer to the last occurence of `(char)c` in ***s***, or **nullptr**.

size_t strspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that is *not* included in string ***set*** (which is regarded as a *set* of characters — the order of the characters does not matter) and returns the length of the initial segment of ***s*** composed of characters which *are* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strspn("sound and fury","os nudda") << endl;
```

will print 10.

size_t strcspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* included in string ***set*** (which is regarded as a *set* of characters) and returns the length of the initial segment of ***s*** composed of characters which are *not* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strcspn("sound and fury","xyztdv") << endl;
```

will print 4.

char* strpbrk(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* in ***set*** and returns a pointer to its location or **nullptr** if not found. For example,

```
cout << strpbrk("Daniel Defoe","wKlor") << endl;
```

will print 'l Defoe'.

char* strstr(const char* s, const char* sub) — returns a pointer to the first character in ***s*** beginning a substring of ***s*** which is identical to ***sub*** (or **nullptr** if substring ***sub*** does not occur in ***s***). For example,

```
cout << strstr("Daniel Defoe","De") << endl;
```

will print 'Defoe'.

char* strtok(char* str, const char* set) — This is a "tokenizer", i.e., a function performing decomposition of a given string into words ("tokens") which are separated one from another by characters from a set ***set*** (which is to be understood as a set of separators, e.g., space, comma, colon etc.). Inner working of the function is based on the existence of an internal pointer of type **char***. Basically, successive calls to **strtok** return successive tokens from the string. When calling it for the first time for a given input string, the string has to be passed as the first argument (***str***). After that, in all successive invocations, ***str*** should be **nullptr**, directing **strtok** to continue from the end of the last token returned. It is not permitted to modify ***str*** between calls, although it is allowable to alter ***set***. The string ***str*** *will* be modified by the function; therefore, it must not be an unmodifiable string, given, e.g., as a literal. The functions works in the following way:

If ***str*** is not **nullptr** (i.e., we start analysing a string), all leading separators from ***str*** are skipped (ignored). If nothing remains, i.e., there are but separators in the string, then the function returns **nullptr** and the internal pointer is set to **nullptr** as well; this terminates processing of a given string. If something remains, then the internal pointer is set to point to the first character encountered that is not a separator, and processing is continued as if ***str*** were **nullptr** (see below).

If ***str*** is **nullptr** and the internal pointer is also **nullptr**, then **nullptr** is returned and the internal pointer is left equal to **nullptr**. This concludes processing of the string.

If ***str*** is **nullptr** but the internal pointer is not **nullptr**, then the functions searches for the first separator, starting from the position currently pointed to by the internal pointer. If found, it is overwritten by an empty character '\0', the function returns current value of the internal pointer, and the pointer itself is set to point to the first character after the '\0' just written. If a separator is not found, the function returns the current value of the internal pointer, and the pointer itself is set to **nullptr**.

All this sounds complicated, but is quite simple to use. For example, the following program

P143: tok.cpp Tokenizer in C

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  int main() {
6.      char strin[] = "int* fun(char& c,double** wtab);";
7.      char separ[] = ")(,;";
8.      char* token;
9.
10.     token = strtok(strin,separ);
11.     while (token != 0) {
12.         cout << token << endl;
13.         token = strtok(0,separ);
14.     }
15. }
```

will print in successive lines

```
int* fun
char& c
double** wtab
```

Implementing functions which operate on C-strings (like those from the header `cstring`) can be a very good programming excercise. For example, some of them can be implemented as follows (and in many other ways):

P144: emulstr.cpp An implementation of some string functions

```
char* Strcpy(char* target, const char* source) {
    char* t = target;
    while ( *t++ = *source++ );
    return target;
}

char* Strcat(char* target, const char* source) {
    char* t = target-1;
    while ( *++t );
    while ( *t++ = *source++ );
    return target;
}

char* Strncat(char* target, const char* source, int n) {
    char* t = target-1;
    while ( *++t );
    while ( (*t++ = *source++) && n--);
    *(t-1) = '\0';
    return target;
}

int Strlen(const char* source) {
    const char* s = source;
    while ( *s++ );
    return s-source-1;
}

char* Strchr(const char* target, int c) {
    while ( *target && *target++ != c );
    return (char*)(*--target ? target : 0);
}
```

It is possible to make these functions even more compact, although not necessarily more comprehensible...

17.1.2 Functions operating on characters

The header `cctype` provides some useful functions operating on characters.

There is a large family of functions that check if a given character meets some criteria. The names of these functions start with `is`; their argument is formally of type **int**, but only the least significant byte, treated as a character, is analysed. They all return an **int**: *any* nonzero value (not necessarily 1) means **true**, zero means **false**. The prototype of these functions has always the form

```
int isSomeProperty(int c);
```

Let us mention the most important of these functions:

- **isalnum** — is `(char)c` a letter or digit?
- **isalpha** — is `(char)c` a letter?
- **isdigit** — is `(char)c` a digit?
- **isxdigit** — is `(char)c` a hexadecimal digit? Hexadecimal digits are digits 0-9 and letters, upper- and lowercase, from the range A-F.
- **iscntrl** — is `(char)c` a control character? Control characters are those with ASCII codes 0-31 inclusive and 127.
- **isprint** — is `(char)c` a printing character? Printing characters are those which are not control characters.
- **isgraph** — is `(char)c` a graphic character? Graphic characters are those which are printing characters (see above), except a space character.
- **ispunct** — is `(char)c` a punctuation character? Punctuation characters are those which are graphic (see above), but are not letters or digits.
- **isspace** — is `(char)c` a whitespace character? Whitespace characters are: '\t' (horizontal tab, symbol HT, ASCII 9), '\r' (carriage return, CR, 13), '\n' (new line, LF, 10), '\v' (vertical tab, VT, 11), '\f' (formfeed, FF, 12) and space (SPC, 32).
- **isupper** — is `(char)c` an uppercase letter?
- **islower** — is `(char)c` a lowercase letter?

Additionally, the header `cctype` provides two useful functions, also of type **int** → **int**:

- **tolower** — for a character which is an uppercase letter returns the corresponding lowercase letter (H → h). For other characters returns its argument.
- **toupper** — for a character which is a lowercase letter returns the corresponding uppercase letter (h → H). For other characters returns its argument.

For example, the function **uplow** in the following program

P145: upplowe.cpp Operations on characters

```
1. #include <iostream>
2. #include <cctype>
3. using namespace std;
4.
5. int uplow(char* s) {
6.     int cnt = 0;
7.     do {
8.         if (isalpha(*s))
9.             if ( cnt == 0 || !isalpha(*(s-1))) {
10.                 *s = (char)toupper(*s);
11.                 cnt++;
12.             } else
13.                 *s = (char)tolower(*s);
14.     } while (*s++);
15.     return cnt;
16. }
17.
18. int main() {
19.     char strn[] = "thiS Is lonG,lONG StRiNg!";
20.
21.     int c = uplow(strn);
22.     cout << c << " words, string = \'" << strn << "\'\n";
23. }
```

changes all first letters of words in a C-string into uppercase, all the remaining into lowercase, and returns the number of words. It prints:
5 words, string = 'This Is Long,Long String!'.

17.1.3 Conversion functions

The header `cstdlib` provides several useful conversion functions. Their usage is simple, but requires a careful error handling; in particular checking the value of ***errno*** which gives information on possible cause of a failure of the last function executed. One can treat ***errno*** as a global variable, although it is normally implemented as a macro (what is easier in multithreaded environment). In order to be able to use ***errno***, one should include the header `cerrno`.

double strtod(const char* str, char ptr)** — (*string to double*) returns a **double** represented by the initial part of a C-string ***str***. Leading whitespaces are skipped and processing of the string terminates when the first "wrong" character is encountered; i.e., a character which cannot be interpreted as a part of a representation of a number. Integer (like 127), scientific (like 1.2E-11) and floating-point with a dot (like 123.34) formats are recognized.

The pointer ***str*** points to a C-string which is supposed to start with a number; other characters after this number are allowed and remain in a stream (if we read from a string stream). The second argument should be the address of a pointer of type **char*** — that is why the type of the corresponding parameter is **char****: pointer to pointer to **char**. On entry, the value of this pointer can be **nullptr** (or just 0), but it is not recommended: it is better to pass as argument the address of an existing pointer variable of type **char***.

After successful conversion, the function returns as a **double** the number which has been read. If ***ptr*** was not **nullptr**, the value of the pointer pointed to by ***ptr*** is set to the address of the first character in the string after the number (this can be the terminating empty character '\0', if there was nothing but a number in the string). If ***ptr*** was **nullptr** on entry, it remains **nullptr** on return. After successful execution of the function, ***errno*** is set to 0.

If string ***str*** does not start with something which can be interpreted as representation of a number, the function returns 0 and the pointer pointed to by ***ptr*** is set to the address of the beginning of ***str***, which is the value of ***str*** itself — this of course happens only if ***ptr*** was not **nullptr**. Therefore, when the function returns 0, we should compare values of ***str*** and of a pointer whose address was passed as the second argument: if they are equal, it is not a "true" zero but a signal that the function failed.

It can happen that ***str*** contains a number, but too large to be represented by a **double** (i.e., it would cause **overflow**). The value ± ***HUGE_VAL***, with the correst sign, is then returned and ***errno*** is set to ***ERANGE***. The constant ***HUGE_VAL*** denotes ***inf*** — infinity. The pointer pointed to by ***ptr*** is set to point to the first character after the number, as if the conversion were successful.

If ***str*** contains a nonzero number but too small to be represented as different from zero (i.e., it causes **underflow**), the function returns zero and ***errno*** is set to ***ERANGE***. The pointer pointed to by ***ptr*** (if not **nullptr**) is set to point to the first character after the number, as if the conversion were successful.

The following program illustrates these definitions:

P146: strtod.cpp Function *strtod*

```
1. #include <iostream>
2. #include <iomanip>   // setw
3. #include <cstdlib>   // strtod
4. #include <cerrno>    // errno
5. using namespace std;
6.
7. int main() {
8.     char*  ptr;
9.     double x;
10.    char*  str;
11.
12.    cout << "ERANGE = " << ERANGE << endl;
13.
14.    // = 1 = OK
15.    str = "-1.2e+2xxx";
16.    x = strtod(str,&ptr);
17.    cout << "=1=  str = " << str << "; x = "
18.         << setw(4) << x << "; errno = " << setw(2)
19.         << errno << "; ptr = " << ptr << endl;
20.
21.    // = 2 =  Not a Number
22.    str = "abcdefghij";
23.    x = strtod(str,&ptr);
24.    cout << "=2=  str = " << str << "; x = "
25.         << setw(4) << x << "; errno = " << setw(2)
26.         << errno << "; ptr = " << ptr << endl;
27.
28.    // = 3 =  Overflow
29.    str = "-9e+9999xx";
30.    x = strtod(str,&ptr);
31.    cout << "=3=  str = " << str << "; x = "
32.         << setw(4) << x << "; errno = " << setw(2)
33.         << errno << "; ptr = " << ptr << endl;
34.
35.    // = 4 =  Underflow
36.    str = "-9e-9999xx";
37.    x = strtod(str,&ptr);
38.    cout << "=4=  str = " << str << "; x = "
39.         << setw(4) << x << "; errno = " << setw(2)
40.         << errno << "; ptr = " << ptr << endl;
41. }
```

Wydruk z tego programu:

```
ERANGE = 34
=1=  str = -1.2e+2xxx; x = -120; errno =  0; ptr = xxx
=2=  str = abcdefghij; x =    0; errno =  0; ptr = abcdefghij
=3=  str = -9e+9999xx; x = -inf; errno = 34; ptr = xx
=4=  str = -9e-9999xx; x =    0; errno = 34; ptr = xx
```

Similar rules apply to the following functions:

long strtol(const char* str, char ptr)** — (*string to long*) as **strtod**, but returns a **long** corresponding to the initial portion of the argument ***str***. In case of overflow returns ***LONG_MAX*** or ***LONG_MIN***. Underflow, of course, cannot happen.

long strtol(const char* str, char ptr, int base)** — as two-argument **strtol**, but expects that the number represented in the initial portion of ***str*** is written with radix ***base***, which can assume values from 2 to 36. The letters 'a' through 'z' (or 'A' to 'Z') represent digits 10 to 36. If ***base*** is 16, the string representation of a number can start with '0x' (or '0X'), which will be ignored. For example, `strtol("J23",&ptr,20)` returns 7643 (= $19 \cdot 400 + 2 \cdot 20 + 3$.

unsigned long strtoul(const char* str, char ptr)** — (*string to unsigned long*) as two-argument **strtol**, but returns an **unsigned long**. In case of overflow returns ***ULONG_MAX***.

unsigned long strtoul(const char* str, char ptr, int base)** — as three-argument **strtol**, but returns an **unsigned long**.

The following traditional conversion functions exist for compatibility, but are not recommended:

double atof(const char* str) — (*ascii to floating*) as **strtod**; in case of overflow returns ± ***HUGE_VAL***, in case of other errors returns zero. In both cases ***errno*** is set to ***ERANGE***.

int atoi(const char* str) — (*ascii to integer*) as **strtol**, but returns an **int**, not a **long**. Error handling similar to that of **atof**.

long atol(const char* str) — (*ascii to long*) as **strtol**; error handling similar to that of **atof**.

Prev: 17. Strings **Up:** 17. Strings **Next:** 17.2 Strings in C++

Subsections

- 17.1.1 Functions on C-strings
- 17.1.2 Functions operating on characters
- 17.1.3 Conversion functions

17.1 C-strings

In traditional C, types **char** and **char*** are treated as generic types often used for referring to raw memory. Therefore, it is not strange that many functions acting on C-strings are very similar to those acting on memory (see section on memory operations in C).

A C-string is just an array of characters; the only requirement is that it should end with a ***NUL*** character — the character with ASCII code equal to 0, whose literal is therefore '\0' — ***NUL*** is just a generally accepted name of this character.

One should remember that

a C-string of type char* initialized with a string literal (enclosed in double quotation marks) is *unmodifiable*.

If we want a C-string to be modifiable, we have to define it explicitly as a character array:

```
1.    char* s1   = "Joe";  // Unmodifiable, dimension 4
2.    char  s2[] = "Joe";  // Modifiable,   dimension 4
3.    char  s3[] = {'J', 'o', 'e', '\0'}       // As s2
```

The two last definitions are equivalent, but the form used in the second line is definitly more convenient. Note that forms from the first two lines make the compiler add the terminating ***NUL*** character itself — in the third form, we have to do it explicilty. In all cases the array will contain four characters: three letters and the ***NUL***. Arrays created in last two lines are normal, local arrays allocated on the stack. However, the array from line 1 will be created elsewhere and *cannot* be modified. It is extremely important to remeber that, especially when passing such tables (C-strings) to functions. The type of ***s1*** is **char***, so the compiler will not object to pass it to a function which does *not* declare the corresponding parameter as **const char***. Nevertheless, an attempt to modify the array inside the function will cause the crash of the program! Using the form from the first line, it is better, although not required by the standard, to declare such pointers as **const char***. Then the compiler will check if we do not try to pass an unmodifiable C-string to a function that does not promise that it will not alter it (as we remember, a function can make such a promise by declaring the correponding parameter as **const char***).

The same applies to literals passed as arguments to functions with a parameter of type **char*** — like invocation `func("John")`. Although the parameter have been declared as modifiable C-string (**char***), any attempt to alter this string inside the function will crash the program!

There are several standard functions operating on C-strings. We can access them by including the header `cstring`. Functions operating on single characters come from the header `cctype`, while some useful conversion functions can be made available by including `cstdlib`.

17.1.1 Functions on C-strings

The functions mentioned below come from the header `cstring`(`string.h` in C) and can be used in both C and C++.

size_t strlen(const char* s) — returns the length of a C-string ***s*** in characters *excluding* the terminating ***NUL***. Type **size_t** is an integer type.

char* strcat(char* dest, const char* src) — appends string ***src*** to ***dest*** (concatenation); returns ***dest***. Both ***dest*** and ***src*** have to be ***NUL*** terminated C-strings. The user is responsible for allocating a segment of memory starting at address pointed to by ***dest*** that is sufficiently large to hold both strings after concatenation. The first ***NUL*** character in ***dest*** is overwritten by the first character of ***src*** followed by the remaining characters of ***src***, including the terminating ***NUL***. It is a common mistake to concatenate a string with an uninitialized string (which does not end with ***NUL***)

```
char s[100];
strcat(s,"Flower");
```

what is a mistake: the program will look for a ***NUL*** in ***s***, but will not find any, because array ***s*** has not been initialized.

char* strncat(char* dest, const char* src, size_t n) — is similar to the previous function, but copies at most ***n*** characters of ***src***. Copying terminates when a ***NUL*** character is encountered and copied. If ***n*** characters have been copied and ***NUL*** has not been found, it will be appended as the ***n+1***-th character. Note that in this case ***n+1*** characters will be added to ***dest***! For example

```
char t1[20] = {'\0'}; // NUL necessary!
strncat(t1,"123456789",5);
cout << t1 << " has " << strlen(t1) << " chars" << endl;
```

will print '12345 has 5 chars' (five characters have been copied, no ***NUL*** has been found, so it was appended as the sixth character).

char* strcpy(char* dest, const char* src) — copies characters from the segment of memory starting at address pointed to by ***src*** to the segment pointed to by ***dest***. The destination region is overwritten. Copying terminates when a ***NUL*** has been copied; the user is responsible for the presence of ***NUL*** in the string being copied and for allocating enough space at address ***dest***. The function returns ***dest***. For example, function **strcat** could be implemented in the following way

```
char* strcat(char* dest, const char* src) {
    char* s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

by using **strcpy** and **strlen**.

char* strncpy(char* dest, const char* src, size_t n) — copies exactly ***n*** characters from location starting at the address pointed to by ***src*** to the one pointed to by ***dest***. When ***NUL*** is encounterd, it is copied and then ***NUL*** characters are inserted into the destination string as padding until exactly ***n*** characters have been written. Note that if the length of ***src*** is equal or bigger than ***n***, the terminating ***NUL*** will *not* be copied.

int strcmp(const char* s1, const char* s2) — compares lexicographically (in dictionary order) two strings. Returns -1, if ***s1*** is less (earlier in dictionary order) than ***s2***, +1, if ***s2*** is less, and 0 if they are equal. A string ***s1*** is considered less than ***s2*** if one of the following condition is met:
a) both string are equal up to a certain position, and at the first differing position the character from ***s1*** has value which is less than the value of the character in ***s2*** at the same position;
b) the string ***s1*** is shorter than ***s2*** and has all characters equal to characters from ***s2*** at the corresponding positions.
For example

```
if ( ! strcmp(s1,s2) )
    cout << "Identical\n";
else
    cout << "Different\n";
```

will work, as any nonzero value returned by **strcmp** will be treated as **true** and, after negating with '!' operator, as **false**.
The example below illustrates the use of **strcmp** function. The function **first_last** (lines 20-26) takes a C-string and two pointers of type **char*** — by reference, so they can be modified. It then puts into pointer ***p*** the address of the beginning of a name (an element of the array of strings) which is lexicographically the least, and into ***q*** the address of the lexicographically last name.

P142: sorword.cpp Comparing C-strings

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  void  first_last(char**,char*&,char*&);
6.
7.  int main() {
8.
9.      char *nam[] = { "Cathy",      "Maggi",
10.                      "Alice",      "Wanda",
11.                      "Wendy", "Catharina", "" },
12.                    *p,*q;
13.
14.     first_last(nam,p,q);
15.
16.     cout << "First: " << p << endl
17.          << "Last : " << q << endl;
18. }
19.
20. void first_last(char** s, char*& p, char*& q) {
21.     p = q = *s;
22.     while ( **++s ) {
23.         if ( strcmp(*s, p) < 0 ) p = *s;
24.         if ( strcmp(*s, q) > 0 ) q = *s;
25.     }
26. }
```

Note that the dimension of the array has not been passed to the function. Instead, the last element of the array is an empty string, containing only ***NUL*** character, which will be supllied by the comopiler (line 11). This technique is very often used in C library (for arrays of C-strings). The code of the function **first_last** is rather compact: it could have been longer and easier to comprehend, but its analysis in the form presented is a good excersise on pointers. The program prints 'First: Alice Last : Wendy'.

int strcoll(const char* s1, const char* s2) — as **strcmp**, by takes locale information into account. For example, in Polish locale the letter 'a' is earlier than 'ą', while in French locale diacritic marks do no count in ordering, so 'â' is equivalent to plain 'a'.

int strncmp(const char* s1, const char* s2, size_t n) — as **strcmp**, but only first ***n*** characters are taken into account.

char* strchr(const char* s, int c) — searches for the first occurrence of character `(char)c` in a string ***s*** and returns a pointer to its position or **nullptr** if this character does dot occur in ***s***. For example,

```
cout << strchr("Daniel Defoe",'f') << endl;
```

will print 'foe'. A function counting the number of occurrences of `(char)c` in a string ***s*** could have the form

```
int count(const char* s, int c) {
    int n = 0;
    while (s = strchr(s,c)) ++s,++n;
    return n;
}
```

In particular, one *can* use **strchr** to look for ***NUL*** character.

char* strrchr(const char* s, int c) — as **strchr**, but returns a pointer to the last occurence of `(char)c` in ***s***, or **nullptr**.

size_t strspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that is *not* included in string ***set*** (which is regarded as a *set* of characters — the order of the characters does not matter) and returns the length of the initial segment of ***s*** composed of characters which *are* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strspn("sound and fury","os nudda") << endl;
```

will print 10.

size_t strcspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* included in string ***set*** (which is regarded as a *set* of characters) and returns the length of the initial segment of ***s*** composed of characters which are *not* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strcspn("sound and fury","xyztdv") << endl;
```

will print 4.

char* strpbrk(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* in ***set*** and returns a pointer to its location or **nullptr** if not found. For example,

```
cout << strpbrk("Daniel Defoe","wKlor") << endl;
```

will print 'l Defoe'.

char* strstr(const char* s, const char* sub) — returns a pointer to the first character in ***s*** beginning a substring of ***s*** which is identical to ***sub*** (or **nullptr** if substring ***sub*** does not occur in ***s***). For example,

```
cout << strstr("Daniel Defoe","De") << endl;
```

will print 'Defoe'.

char* strtok(char* str, const char* set) — This is a "tokenizer", i.e., a function performing decomposition of a given string into words ("tokens") which are separated one from another by characters from a set ***set*** (which is to be understood as a set of separators, e.g., space, comma, colon etc.). Inner working of the function is based on the existence of an internal pointer of type **char***. Basically, successive calls to **strtok** return successive tokens from the string. When calling it for the first time for a given input string, the string has to be passed as the first argument (***str***). After that, in all successive invocations, ***str*** should be **nullptr**, directing **strtok** to continue from the end of the last token returned. It is not permitted to modify ***str*** between calls, although it is allowable to alter ***set***. The string ***str*** *will* be modified by the function; therefore, it must not be an unmodifiable string, given, e.g., as a literal. The functions works in the following way:

If ***str*** is not **nullptr** (i.e., we start analysing a string), all leading separators from ***str*** are skipped (ignored). If nothing remains, i.e., there are but separators in the string, then the function returns **nullptr** and the internal pointer is set to **nullptr** as well; this terminates processing of a given string. If something remains, then the internal pointer is set to point to the first character encountered that is not a separator, and processing is continued as if ***str*** were **nullptr** (see below).

If ***str*** is **nullptr** and the internal pointer is also **nullptr**, then **nullptr** is returned and the internal pointer is left equal to **nullptr**. This concludes processing of the string.

If ***str*** is **nullptr** but the internal pointer is not **nullptr**, then the functions searches for the first separator, starting from the position currently pointed to by the internal pointer. If found, it is overwritten by an empty character '\0', the function returns current value of the internal pointer, and the pointer itself is set to point to the first character after the '\0' just written. If a separator is not found, the function returns the current value of the internal pointer, and the pointer itself is set to **nullptr**.

All this sounds complicated, but is quite simple to use. For example, the following program

P143: tok.cpp Tokenizer in C

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  int main() {
6.      char strin[] = "int* fun(char& c,double** wtab);";
7.      char separ[] = ")(,;";
8.      char* token;
9.
10.     token = strtok(strin,separ);
11.     while (token != 0) {
12.         cout << token << endl;
13.         token = strtok(0,separ);
14.     }
15. }
```

will print in successive lines

```
int* fun
char& c
double** wtab
```

Implementing functions which operate on C-strings (like those from the header `cstring`) can be a very good programming excercise. For example, some of them can be implemented as follows (and in many other ways):

P144: emulstr.cpp An implementation of some string functions

```
char* Strcpy(char* target, const char* source) {
    char* t = target;
    while ( *t++ = *source++ );
    return target;
}

char* Strcat(char* target, const char* source) {
    char* t = target-1;
    while ( *++t );
    while ( *t++ = *source++ );
    return target;
}

char* Strncat(char* target, const char* source, int n) {
    char* t = target-1;
    while ( *++t );
    while ( (*t++ = *source++) && n--);
    *(t-1) = '\0';
    return target;
}

int Strlen(const char* source) {
    const char* s = source;
    while ( *s++ );
    return s-source-1;
}

char* Strchr(const char* target, int c) {
    while ( *target && *target++ != c );
    return (char*)(*--target ? target : 0);
}
```

It is possible to make these functions even more compact, although not necessarily more comprehensible...

17.1.2 Functions operating on characters

The header `cctype` provides some useful functions operating on characters.

There is a large family of functions that check if a given character meets some criteria. The names of these functions start with `is`; their argument is formally of type **int**, but only the least significant byte, treated as a character, is analysed. They all return an **int**: *any* nonzero value (not necessarily 1) means **true**, zero means **false**. The prototype of these functions has always the form

```
int isSomeProperty(int c);
```

Let us mention the most important of these functions:

- **isalnum** — is `(char)c` a letter or digit?
- **isalpha** — is `(char)c` a letter?
- **isdigit** — is `(char)c` a digit?
- **isxdigit** — is `(char)c` a hexadecimal digit? Hexadecimal digits are digits 0-9 and letters, upper- and lowercase, from the range A-F.
- **iscntrl** — is `(char)c` a control character? Control characters are those with ASCII codes 0-31 inclusive and 127.
- **isprint** — is `(char)c` a printing character? Printing characters are those which are not control characters.
- **isgraph** — is `(char)c` a graphic character? Graphic characters are those which are printing characters (see above), except a space character.
- **ispunct** — is `(char)c` a punctuation character? Punctuation characters are those which are graphic (see above), but are not letters or digits.
- **isspace** — is `(char)c` a whitespace character? Whitespace characters are: '\t' (horizontal tab, symbol HT, ASCII 9), '\r' (carriage return, CR, 13), '\n' (new line, LF, 10), '\v' (vertical tab, VT, 11), '\f' (formfeed, FF, 12) and space (SPC, 32).
- **isupper** — is `(char)c` an uppercase letter?
- **islower** — is `(char)c` a lowercase letter?

Additionally, the header `cctype` provides two useful functions, also of type **int** → **int**:

- **tolower** — for a character which is an uppercase letter returns the corresponding lowercase letter (H → h). For other characters returns its argument.
- **toupper** — for a character which is a lowercase letter returns the corresponding uppercase letter (h → H). For other characters returns its argument.

For example, the function **uplow** in the following program

P145: upplowe.cpp Operations on characters

```
1. #include <iostream>
2. #include <cctype>
3. using namespace std;
4.
5. int uplow(char* s) {
6.     int cnt = 0;
7.     do {
8.         if (isalpha(*s))
9.             if ( cnt == 0 || !isalpha(*(s-1))) {
10.                 *s = (char)toupper(*s);
11.                 cnt++;
12.             } else
13.                 *s = (char)tolower(*s);
14.     } while (*s++);
15.     return cnt;
16. }
17.
18. int main() {
19.     char strn[] = "thiS Is lonG,lONG StRiNg!";
20.
21.     int c = uplow(strn);
22.     cout << c << " words, string = \'" << strn << "\'\n";
23. }
```

changes all first letters of words in a C-string into uppercase, all the remaining into lowercase, and returns the number of words. It prints:
5 words, string = 'This Is Long,Long String!'.

17.1.3 Conversion functions

The header `cstdlib` provides several useful conversion functions. Their usage is simple, but requires a careful error handling; in particular checking the value of ***errno*** which gives information on possible cause of a failure of the last function executed. One can treat ***errno*** as a global variable, although it is normally implemented as a macro (what is easier in multithreaded environment). In order to be able to use ***errno***, one should include the header `cerrno`.

double strtod(const char* str, char ptr)** — (*string to double*) returns a **double** represented by the initial part of a C-string ***str***. Leading whitespaces are skipped and processing of the string terminates when the first "wrong" character is encountered; i.e., a character which cannot be interpreted as a part of a representation of a number. Integer (like 127), scientific (like 1.2E-11) and floating-point with a dot (like 123.34) formats are recognized.

The pointer ***str*** points to a C-string which is supposed to start with a number; other characters after this number are allowed and remain in a stream (if we read from a string stream). The second argument should be the address of a pointer of type **char*** — that is why the type of the corresponding parameter is **char****: pointer to pointer to **char**. On entry, the value of this pointer can be **nullptr** (or just 0), but it is not recommended: it is better to pass as argument the address of an existing pointer variable of type **char***.

After successful conversion, the function returns as a **double** the number which has been read. If ***ptr*** was not **nullptr**, the value of the pointer pointed to by ***ptr*** is set to the address of the first character in the string after the number (this can be the terminating empty character '\0', if there was nothing but a number in the string). If ***ptr*** was **nullptr** on entry, it remains **nullptr** on return. After successful execution of the function, ***errno*** is set to 0.

If string ***str*** does not start with something which can be interpreted as representation of a number, the function returns 0 and the pointer pointed to by ***ptr*** is set to the address of the beginning of ***str***, which is the value of ***str*** itself — this of course happens only if ***ptr*** was not **nullptr**. Therefore, when the function returns 0, we should compare values of ***str*** and of a pointer whose address was passed as the second argument: if they are equal, it is not a "true" zero but a signal that the function failed.

It can happen that ***str*** contains a number, but too large to be represented by a **double** (i.e., it would cause **overflow**). The value ± ***HUGE_VAL***, with the correst sign, is then returned and ***errno*** is set to ***ERANGE***. The constant ***HUGE_VAL*** denotes ***inf*** — infinity. The pointer pointed to by ***ptr*** is set to point to the first character after the number, as if the conversion were successful.

If ***str*** contains a nonzero number but too small to be represented as different from zero (i.e., it causes **underflow**), the function returns zero and ***errno*** is set to ***ERANGE***. The pointer pointed to by ***ptr*** (if not **nullptr**) is set to point to the first character after the number, as if the conversion were successful.

The following program illustrates these definitions:

P146: strtod.cpp Function *strtod*

```
1. #include <iostream>
2. #include <iomanip>   // setw
3. #include <cstdlib>   // strtod
4. #include <cerrno>    // errno
5. using namespace std;
6.
7. int main() {
8.     char*  ptr;
9.     double x;
10.    char*  str;
11.
12.    cout << "ERANGE = " << ERANGE << endl;
13.
14.    // = 1 = OK
15.    str = "-1.2e+2xxx";
16.    x = strtod(str,&ptr);
17.    cout << "=1=  str = " << str << "; x = "
18.         << setw(4) << x << "; errno = " << setw(2)
19.         << errno << "; ptr = " << ptr << endl;
20.
21.    // = 2 =  Not a Number
22.    str = "abcdefghij";
23.    x = strtod(str,&ptr);
24.    cout << "=2=  str = " << str << "; x = "
25.         << setw(4) << x << "; errno = " << setw(2)
26.         << errno << "; ptr = " << ptr << endl;
27.
28.    // = 3 =  Overflow
29.    str = "-9e+9999xx";
30.    x = strtod(str,&ptr);
31.    cout << "=3=  str = " << str << "; x = "
32.         << setw(4) << x << "; errno = " << setw(2)
33.         << errno << "; ptr = " << ptr << endl;
34.
35.    // = 4 =  Underflow
36.    str = "-9e-9999xx";
37.    x = strtod(str,&ptr);
38.    cout << "=4=  str = " << str << "; x = "
39.         << setw(4) << x << "; errno = " << setw(2)
40.         << errno << "; ptr = " << ptr << endl;
41. }
```

Wydruk z tego programu:

```
ERANGE = 34
=1=  str = -1.2e+2xxx; x = -120; errno =  0; ptr = xxx
=2=  str = abcdefghij; x =    0; errno =  0; ptr = abcdefghij
=3=  str = -9e+9999xx; x = -inf; errno = 34; ptr = xx
=4=  str = -9e-9999xx; x =    0; errno = 34; ptr = xx
```

Similar rules apply to the following functions:

long strtol(const char* str, char ptr)** — (*string to long*) as **strtod**, but returns a **long** corresponding to the initial portion of the argument ***str***. In case of overflow returns ***LONG_MAX*** or ***LONG_MIN***. Underflow, of course, cannot happen.

long strtol(const char* str, char ptr, int base)** — as two-argument **strtol**, but expects that the number represented in the initial portion of ***str*** is written with radix ***base***, which can assume values from 2 to 36. The letters 'a' through 'z' (or 'A' to 'Z') represent digits 10 to 36. If ***base*** is 16, the string representation of a number can start with '0x' (or '0X'), which will be ignored. For example, `strtol("J23",&ptr,20)` returns 7643 (= $19 \cdot 400 + 2 \cdot 20 + 3$.

unsigned long strtoul(const char* str, char ptr)** — (*string to unsigned long*) as two-argument **strtol**, but returns an **unsigned long**. In case of overflow returns ***ULONG_MAX***.

unsigned long strtoul(const char* str, char ptr, int base)** — as three-argument **strtol**, but returns an **unsigned long**.

The following traditional conversion functions exist for compatibility, but are not recommended:

double atof(const char* str) — (*ascii to floating*) as **strtod**; in case of overflow returns ± ***HUGE_VAL***, in case of other errors returns zero. In both cases ***errno*** is set to ***ERANGE***.

int atoi(const char* str) — (*ascii to integer*) as **strtol**, but returns an **int**, not a **long**. Error handling similar to that of **atof**.

long atol(const char* str) — (*ascii to long*) as **strtol**; error handling similar to that of **atof**.

Prev: 17. Strings **Up:** 17. Strings **Next:** 17.2 Strings in C++

Subsections

- 17.1.1 Functions on C-strings
- 17.1.2 Functions operating on characters
- 17.1.3 Conversion functions

17.1 C-strings

In traditional C, types **char** and **char*** are treated as generic types often used for referring to raw memory. Therefore, it is not strange that many functions acting on C-strings are very similar to those acting on memory (see section on memory operations in C).

A C-string is just an array of characters; the only requirement is that it should end with a ***NUL*** character — the character with ASCII code equal to 0, whose literal is therefore '\0' — ***NUL*** is just a generally accepted name of this character.

One should remember that

a C-string of type char* initialized with a string literal (enclosed in double quotation marks) is *unmodifiable*.

If we want a C-string to be modifiable, we have to define it explicitly as a character array:

```
1.    char* s1   = "Joe";  // Unmodifiable, dimension 4
2.    char  s2[] = "Joe";  // Modifiable,   dimension 4
3.    char  s3[] = {'J', 'o', 'e', '\0'}       // As s2
```

The two last definitions are equivalent, but the form used in the second line is definitly more convenient. Note that forms from the first two lines make the compiler add the terminating ***NUL*** character itself — in the third form, we have to do it explicilty. In all cases the array will contain four characters: three letters and the ***NUL***. Arrays created in last two lines are normal, local arrays allocated on the stack. However, the array from line 1 will be created elsewhere and *cannot* be modified. It is extremely important to remeber that, especially when passing such tables (C-strings) to functions. The type of ***s1*** is **char***, so the compiler will not object to pass it to a function which does *not* declare the corresponding parameter as **const char***. Nevertheless, an attempt to modify the array inside the function will cause the crash of the program! Using the form from the first line, it is better, although not required by the standard, to declare such pointers as **const char***. Then the compiler will check if we do not try to pass an unmodifiable C-string to a function that does not promise that it will not alter it (as we remember, a function can make such a promise by declaring the correponding parameter as **const char***).

The same applies to literals passed as arguments to functions with a parameter of type **char*** — like invocation `func("John")`. Although the parameter have been declared as modifiable C-string (**char***), any attempt to alter this string inside the function will crash the program!

There are several standard functions operating on C-strings. We can access them by including the header `cstring`. Functions operating on single characters come from the header `cctype`, while some useful conversion functions can be made available by including `cstdlib`.

17.1.1 Functions on C-strings

The functions mentioned below come from the header `cstring`(`string.h` in C) and can be used in both C and C++.

size_t strlen(const char* s) — returns the length of a C-string ***s*** in characters *excluding* the terminating ***NUL***. Type **size_t** is an integer type.

char* strcat(char* dest, const char* src) — appends string ***src*** to ***dest*** (concatenation); returns ***dest***. Both ***dest*** and ***src*** have to be ***NUL*** terminated C-strings. The user is responsible for allocating a segment of memory starting at address pointed to by ***dest*** that is sufficiently large to hold both strings after concatenation. The first ***NUL*** character in ***dest*** is overwritten by the first character of ***src*** followed by the remaining characters of ***src***, including the terminating ***NUL***. It is a common mistake to concatenate a string with an uninitialized string (which does not end with ***NUL***)

```
char s[100];
strcat(s,"Flower");
```

what is a mistake: the program will look for a ***NUL*** in ***s***, but will not find any, because array ***s*** has not been initialized.

char* strncat(char* dest, const char* src, size_t n) — is similar to the previous function, but copies at most ***n*** characters of ***src***. Copying terminates when a ***NUL*** character is encountered and copied. If ***n*** characters have been copied and ***NUL*** has not been found, it will be appended as the ***n+1***-th character. Note that in this case ***n+1*** characters will be added to ***dest***! For example

```
char t1[20] = {'\0'}; // NUL necessary!
strncat(t1,"123456789",5);
cout << t1 << " has " << strlen(t1) << " chars" << endl;
```

will print '12345 has 5 chars' (five characters have been copied, no ***NUL*** has been found, so it was appended as the sixth character).

char* strcpy(char* dest, const char* src) — copies characters from the segment of memory starting at address pointed to by ***src*** to the segment pointed to by ***dest***. The destination region is overwritten. Copying terminates when a ***NUL*** has been copied; the user is responsible for the presence of ***NUL*** in the string being copied and for allocating enough space at address ***dest***. The function returns ***dest***. For example, function **strcat** could be implemented in the following way

```
char* strcat(char* dest, const char* src) {
    char* s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

by using **strcpy** and **strlen**.

char* strncpy(char* dest, const char* src, size_t n) — copies exactly ***n*** characters from location starting at the address pointed to by ***src*** to the one pointed to by ***dest***. When ***NUL*** is encounterd, it is copied and then ***NUL*** characters are inserted into the destination string as padding until exactly ***n*** characters have been written. Note that if the length of ***src*** is equal or bigger than ***n***, the terminating ***NUL*** will *not* be copied.

int strcmp(const char* s1, const char* s2) — compares lexicographically (in dictionary order) two strings. Returns -1, if ***s1*** is less (earlier in dictionary order) than ***s2***, +1, if ***s2*** is less, and 0 if they are equal. A string ***s1*** is considered less than ***s2*** if one of the following condition is met:
a) both string are equal up to a certain position, and at the first differing position the character from ***s1*** has value which is less than the value of the character in ***s2*** at the same position;
b) the string ***s1*** is shorter than ***s2*** and has all characters equal to characters from ***s2*** at the corresponding positions.
For example

```
if ( ! strcmp(s1,s2) )
    cout << "Identical\n";
else
    cout << "Different\n";
```

will work, as any nonzero value returned by **strcmp** will be treated as **true** and, after negating with '!' operator, as **false**.
The example below illustrates the use of **strcmp** function. The function **first_last** (lines 20-26) takes a C-string and two pointers of type **char*** — by reference, so they can be modified. It then puts into pointer ***p*** the address of the beginning of a name (an element of the array of strings) which is lexicographically the least, and into ***q*** the address of the lexicographically last name.

P142: sorword.cpp Comparing C-strings

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  void  first_last(char**,char*&,char*&);
6.
7.  int main() {
8.
9.      char *nam[] = { "Cathy",      "Maggi",
10.                      "Alice",      "Wanda",
11.                      "Wendy", "Catharina", "" },
12.                    *p,*q;
13.
14.     first_last(nam,p,q);
15.
16.     cout << "First: " << p << endl
17.          << "Last : " << q << endl;
18. }
19.
20. void first_last(char** s, char*& p, char*& q) {
21.     p = q = *s;
22.     while ( **++s ) {
23.         if ( strcmp(*s, p) < 0 ) p = *s;
24.         if ( strcmp(*s, q) > 0 ) q = *s;
25.     }
26. }
```

Note that the dimension of the array has not been passed to the function. Instead, the last element of the array is an empty string, containing only ***NUL*** character, which will be supllied by the comopiler (line 11). This technique is very often used in C library (for arrays of C-strings). The code of the function **first_last** is rather compact: it could have been longer and easier to comprehend, but its analysis in the form presented is a good excersise on pointers. The program prints 'First: Alice Last : Wendy'.

int strcoll(const char* s1, const char* s2) — as **strcmp**, by takes locale information into account. For example, in Polish locale the letter 'a' is earlier than 'ą', while in French locale diacritic marks do no count in ordering, so 'â' is equivalent to plain 'a'.

int strncmp(const char* s1, const char* s2, size_t n) — as **strcmp**, but only first ***n*** characters are taken into account.

char* strchr(const char* s, int c) — searches for the first occurrence of character `(char)c` in a string ***s*** and returns a pointer to its position or **nullptr** if this character does dot occur in ***s***. For example,

```
cout << strchr("Daniel Defoe",'f') << endl;
```

will print 'foe'. A function counting the number of occurrences of `(char)c` in a string ***s*** could have the form

```
int count(const char* s, int c) {
    int n = 0;
    while (s = strchr(s,c)) ++s,++n;
    return n;
}
```

In particular, one *can* use **strchr** to look for ***NUL*** character.

char* strrchr(const char* s, int c) — as **strchr**, but returns a pointer to the last occurence of `(char)c` in ***s***, or **nullptr**.

size_t strspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that is *not* included in string ***set*** (which is regarded as a *set* of characters — the order of the characters does not matter) and returns the length of the initial segment of ***s*** composed of characters which *are* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strspn("sound and fury","os nudda") << endl;
```

will print 10.

size_t strcspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* included in string ***set*** (which is regarded as a *set* of characters) and returns the length of the initial segment of ***s*** composed of characters which are *not* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strcspn("sound and fury","xyztdv") << endl;
```

will print 4.

char* strpbrk(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* in ***set*** and returns a pointer to its location or **nullptr** if not found. For example,

```
cout << strpbrk("Daniel Defoe","wKlor") << endl;
```

will print 'l Defoe'.

char* strstr(const char* s, const char* sub) — returns a pointer to the first character in ***s*** beginning a substring of ***s*** which is identical to ***sub*** (or **nullptr** if substring ***sub*** does not occur in ***s***). For example,

```
cout << strstr("Daniel Defoe","De") << endl;
```

will print 'Defoe'.

char* strtok(char* str, const char* set) — This is a "tokenizer", i.e., a function performing decomposition of a given string into words ("tokens") which are separated one from another by characters from a set ***set*** (which is to be understood as a set of separators, e.g., space, comma, colon etc.). Inner working of the function is based on the existence of an internal pointer of type **char***. Basically, successive calls to **strtok** return successive tokens from the string. When calling it for the first time for a given input string, the string has to be passed as the first argument (***str***). After that, in all successive invocations, ***str*** should be **nullptr**, directing **strtok** to continue from the end of the last token returned. It is not permitted to modify ***str*** between calls, although it is allowable to alter ***set***. The string ***str*** *will* be modified by the function; therefore, it must not be an unmodifiable string, given, e.g., as a literal. The functions works in the following way:

If ***str*** is not **nullptr** (i.e., we start analysing a string), all leading separators from ***str*** are skipped (ignored). If nothing remains, i.e., there are but separators in the string, then the function returns **nullptr** and the internal pointer is set to **nullptr** as well; this terminates processing of a given string. If something remains, then the internal pointer is set to point to the first character encountered that is not a separator, and processing is continued as if ***str*** were **nullptr** (see below).

If ***str*** is **nullptr** and the internal pointer is also **nullptr**, then **nullptr** is returned and the internal pointer is left equal to **nullptr**. This concludes processing of the string.

If ***str*** is **nullptr** but the internal pointer is not **nullptr**, then the functions searches for the first separator, starting from the position currently pointed to by the internal pointer. If found, it is overwritten by an empty character '\0', the function returns current value of the internal pointer, and the pointer itself is set to point to the first character after the '\0' just written. If a separator is not found, the function returns the current value of the internal pointer, and the pointer itself is set to **nullptr**.

All this sounds complicated, but is quite simple to use. For example, the following program

P143: tok.cpp Tokenizer in C

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  int main() {
6.      char strin[] = "int* fun(char& c,double** wtab);";
7.      char separ[] = ")(,;";
8.      char* token;
9.
10.     token = strtok(strin,separ);
11.     while (token != 0) {
12.         cout << token << endl;
13.         token = strtok(0,separ);
14.     }
15. }
```

will print in successive lines

```
int* fun
char& c
double** wtab
```

Implementing functions which operate on C-strings (like those from the header `cstring`) can be a very good programming excercise. For example, some of them can be implemented as follows (and in many other ways):

P144: emulstr.cpp An implementation of some string functions

```
char* Strcpy(char* target, const char* source) {
    char* t = target;
    while ( *t++ = *source++ );
    return target;
}

char* Strcat(char* target, const char* source) {
    char* t = target-1;
    while ( *++t );
    while ( *t++ = *source++ );
    return target;
}

char* Strncat(char* target, const char* source, int n) {
    char* t = target-1;
    while ( *++t );
    while ( (*t++ = *source++) && n--);
    *(t-1) = '\0';
    return target;
}

int Strlen(const char* source) {
    const char* s = source;
    while ( *s++ );
    return s-source-1;
}

char* Strchr(const char* target, int c) {
    while ( *target && *target++ != c );
    return (char*)(*--target ? target : 0);
}
```

It is possible to make these functions even more compact, although not necessarily more comprehensible...

17.1.2 Functions operating on characters

The header `cctype` provides some useful functions operating on characters.

There is a large family of functions that check if a given character meets some criteria. The names of these functions start with `is`; their argument is formally of type **int**, but only the least significant byte, treated as a character, is analysed. They all return an **int**: *any* nonzero value (not necessarily 1) means **true**, zero means **false**. The prototype of these functions has always the form

```
int isSomeProperty(int c);
```

Let us mention the most important of these functions:

- **isalnum** — is `(char)c` a letter or digit?
- **isalpha** — is `(char)c` a letter?
- **isdigit** — is `(char)c` a digit?
- **isxdigit** — is `(char)c` a hexadecimal digit? Hexadecimal digits are digits 0-9 and letters, upper- and lowercase, from the range A-F.
- **iscntrl** — is `(char)c` a control character? Control characters are those with ASCII codes 0-31 inclusive and 127.
- **isprint** — is `(char)c` a printing character? Printing characters are those which are not control characters.
- **isgraph** — is `(char)c` a graphic character? Graphic characters are those which are printing characters (see above), except a space character.
- **ispunct** — is `(char)c` a punctuation character? Punctuation characters are those which are graphic (see above), but are not letters or digits.
- **isspace** — is `(char)c` a whitespace character? Whitespace characters are: '\t' (horizontal tab, symbol HT, ASCII 9), '\r' (carriage return, CR, 13), '\n' (new line, LF, 10), '\v' (vertical tab, VT, 11), '\f' (formfeed, FF, 12) and space (SPC, 32).
- **isupper** — is `(char)c` an uppercase letter?
- **islower** — is `(char)c` a lowercase letter?

Additionally, the header `cctype` provides two useful functions, also of type **int** → **int**:

- **tolower** — for a character which is an uppercase letter returns the corresponding lowercase letter (H → h). For other characters returns its argument.
- **toupper** — for a character which is a lowercase letter returns the corresponding uppercase letter (h → H). For other characters returns its argument.

For example, the function **uplow** in the following program

P145: upplowe.cpp Operations on characters

```
1. #include <iostream>
2. #include <cctype>
3. using namespace std;
4.
5. int uplow(char* s) {
6.     int cnt = 0;
7.     do {
8.         if (isalpha(*s))
9.             if ( cnt == 0 || !isalpha(*(s-1))) {
10.                 *s = (char)toupper(*s);
11.                 cnt++;
12.             } else
13.                 *s = (char)tolower(*s);
14.     } while (*s++);
15.     return cnt;
16. }
17.
18. int main() {
19.     char strn[] = "thiS Is lonG,lONG StRiNg!";
20.
21.     int c = uplow(strn);
22.     cout << c << " words, string = \'" << strn << "\'\n";
23. }
```

changes all first letters of words in a C-string into uppercase, all the remaining into lowercase, and returns the number of words. It prints:
5 words, string = 'This Is Long,Long String!'.

17.1.3 Conversion functions

The header `cstdlib` provides several useful conversion functions. Their usage is simple, but requires a careful error handling; in particular checking the value of ***errno*** which gives information on possible cause of a failure of the last function executed. One can treat ***errno*** as a global variable, although it is normally implemented as a macro (what is easier in multithreaded environment). In order to be able to use ***errno***, one should include the header `cerrno`.

double strtod(const char* str, char ptr)** — (*string to double*) returns a **double** represented by the initial part of a C-string ***str***. Leading whitespaces are skipped and processing of the string terminates when the first "wrong" character is encountered; i.e., a character which cannot be interpreted as a part of a representation of a number. Integer (like 127), scientific (like 1.2E-11) and floating-point with a dot (like 123.34) formats are recognized.

The pointer ***str*** points to a C-string which is supposed to start with a number; other characters after this number are allowed and remain in a stream (if we read from a string stream). The second argument should be the address of a pointer of type **char*** — that is why the type of the corresponding parameter is **char****: pointer to pointer to **char**. On entry, the value of this pointer can be **nullptr** (or just 0), but it is not recommended: it is better to pass as argument the address of an existing pointer variable of type **char***.

After successful conversion, the function returns as a **double** the number which has been read. If ***ptr*** was not **nullptr**, the value of the pointer pointed to by ***ptr*** is set to the address of the first character in the string after the number (this can be the terminating empty character '\0', if there was nothing but a number in the string). If ***ptr*** was **nullptr** on entry, it remains **nullptr** on return. After successful execution of the function, ***errno*** is set to 0.

If string ***str*** does not start with something which can be interpreted as representation of a number, the function returns 0 and the pointer pointed to by ***ptr*** is set to the address of the beginning of ***str***, which is the value of ***str*** itself — this of course happens only if ***ptr*** was not **nullptr**. Therefore, when the function returns 0, we should compare values of ***str*** and of a pointer whose address was passed as the second argument: if they are equal, it is not a "true" zero but a signal that the function failed.

It can happen that ***str*** contains a number, but too large to be represented by a **double** (i.e., it would cause **overflow**). The value ± ***HUGE_VAL***, with the correst sign, is then returned and ***errno*** is set to ***ERANGE***. The constant ***HUGE_VAL*** denotes ***inf*** — infinity. The pointer pointed to by ***ptr*** is set to point to the first character after the number, as if the conversion were successful.

If ***str*** contains a nonzero number but too small to be represented as different from zero (i.e., it causes **underflow**), the function returns zero and ***errno*** is set to ***ERANGE***. The pointer pointed to by ***ptr*** (if not **nullptr**) is set to point to the first character after the number, as if the conversion were successful.

The following program illustrates these definitions:

P146: strtod.cpp Function *strtod*

```
1. #include <iostream>
2. #include <iomanip>   // setw
3. #include <cstdlib>   // strtod
4. #include <cerrno>    // errno
5. using namespace std;
6.
7. int main() {
8.     char*  ptr;
9.     double x;
10.    char*  str;
11.
12.    cout << "ERANGE = " << ERANGE << endl;
13.
14.    // = 1 = OK
15.    str = "-1.2e+2xxx";
16.    x = strtod(str,&ptr);
17.    cout << "=1=  str = " << str << "; x = "
18.         << setw(4) << x << "; errno = " << setw(2)
19.         << errno << "; ptr = " << ptr << endl;
20.
21.    // = 2 =  Not a Number
22.    str = "abcdefghij";
23.    x = strtod(str,&ptr);
24.    cout << "=2=  str = " << str << "; x = "
25.         << setw(4) << x << "; errno = " << setw(2)
26.         << errno << "; ptr = " << ptr << endl;
27.
28.    // = 3 =  Overflow
29.    str = "-9e+9999xx";
30.    x = strtod(str,&ptr);
31.    cout << "=3=  str = " << str << "; x = "
32.         << setw(4) << x << "; errno = " << setw(2)
33.         << errno << "; ptr = " << ptr << endl;
34.
35.    // = 4 =  Underflow
36.    str = "-9e-9999xx";
37.    x = strtod(str,&ptr);
38.    cout << "=4=  str = " << str << "; x = "
39.         << setw(4) << x << "; errno = " << setw(2)
40.         << errno << "; ptr = " << ptr << endl;
41. }
```

Wydruk z tego programu:

```
ERANGE = 34
=1=  str = -1.2e+2xxx; x = -120; errno =  0; ptr = xxx
=2=  str = abcdefghij; x =    0; errno =  0; ptr = abcdefghij
=3=  str = -9e+9999xx; x = -inf; errno = 34; ptr = xx
=4=  str = -9e-9999xx; x =    0; errno = 34; ptr = xx
```

Similar rules apply to the following functions:

long strtol(const char* str, char ptr)** — (*string to long*) as **strtod**, but returns a **long** corresponding to the initial portion of the argument ***str***. In case of overflow returns ***LONG_MAX*** or ***LONG_MIN***. Underflow, of course, cannot happen.

long strtol(const char* str, char ptr, int base)** — as two-argument **strtol**, but expects that the number represented in the initial portion of ***str*** is written with radix ***base***, which can assume values from 2 to 36. The letters 'a' through 'z' (or 'A' to 'Z') represent digits 10 to 36. If ***base*** is 16, the string representation of a number can start with '0x' (or '0X'), which will be ignored. For example, `strtol("J23",&ptr,20)` returns 7643 (= $19 \cdot 400 + 2 \cdot 20 + 3$.

unsigned long strtoul(const char* str, char ptr)** — (*string to unsigned long*) as two-argument **strtol**, but returns an **unsigned long**. In case of overflow returns ***ULONG_MAX***.

unsigned long strtoul(const char* str, char ptr, int base)** — as three-argument **strtol**, but returns an **unsigned long**.

The following traditional conversion functions exist for compatibility, but are not recommended:

double atof(const char* str) — (*ascii to floating*) as **strtod**; in case of overflow returns ± ***HUGE_VAL***, in case of other errors returns zero. In both cases ***errno*** is set to ***ERANGE***.

int atoi(const char* str) — (*ascii to integer*) as **strtol**, but returns an **int**, not a **long**. Error handling similar to that of **atof**.

long atol(const char* str) — (*ascii to long*) as **strtol**; error handling similar to that of **atof**.

Prev: 17. Strings **Up:** 17. Strings **Next:** 17.2 Strings in C++

Subsections

- 17.1.1 Functions on C-strings
- 17.1.2 Functions operating on characters
- 17.1.3 Conversion functions

17.1 C-strings

In traditional C, types **char** and **char*** are treated as generic types often used for referring to raw memory. Therefore, it is not strange that many functions acting on C-strings are very similar to those acting on memory (see section on memory operations in C).

A C-string is just an array of characters; the only requirement is that it should end with a ***NUL*** character — the character with ASCII code equal to 0, whose literal is therefore '\0' — ***NUL*** is just a generally accepted name of this character.

One should remember that

> **a C-string of type char* initialized with a string literal (enclosed in double quotation marks) is *unmodifiable*.**

If we want a C-string to be modifiable, we have to define it explicitly as a character array:

```
1.    char* s1   = "Joe";  // Unmodifiable, dimension 4
2.    char  s2[] = "Joe";  // Modifiable,   dimension 4
3.    char  s3[] = {'J', 'o', 'e', '\0'}       // As s2
```

The two last definitions are equivalent, but the form used in the second line is definitly more convenient. Note that forms from the first two lines make the compiler add the terminating ***NUL*** character itself — in the third form, we have to do it explicilty. In all cases the array will contain four characters: three letters and the ***NUL***. Arrays created in last two lines are normal, local arrays allocated on the stack. However, the array from line 1 will be created elsewhere and *cannot* be modified. It is extremely important to remeber that, especially when passing such tables (C-strings) to functions. The type of ***s1*** is **char***, so the compiler will not object to pass it to a function which does *not* declare the corresponding parameter as **const char***. Nevertheless, an attempt to modify the array inside the function will cause the crash of the program! Using the form from the first line, it is better, although not required by the standard, to declare such pointers as **const char***. Then the compiler will check if we do not try to pass an unmodifiable C-string to a function that does not promise that it will not alter it (as we remember, a function can make such a promise by declaring the correponding parameter as **const char***).

The same applies to literals passed as arguments to functions with a parameter of type **char*** — like invocation `func("John")`. Although the parameter have been declared as modifiable C-string (**char***), any attempt to alter this string inside the function will crash the program!

There are several standard functions operating on C-strings. We can access them by including the header `cstring`. Functions operating on single characters come from the header `cctype`, while some useful conversion functions can be made available by including `cstdlib`.

17.1.1 Functions on C-strings

The functions mentioned below come from the header `cstring`(`string.h` in C) and can be used in both C and C++.

size_t strlen(const char* s) — returns the length of a C-string ***s*** in characters *excluding* the terminating ***NUL***. Type **size_t** is an integer type.

char* strcat(char* dest, const char* src) — appends string ***src*** to ***dest*** (concatenation); returns ***dest***. Both ***dest*** and ***src*** have to be ***NUL*** terminated C-strings. The user is responsible for allocating a segment of memory starting at address pointed to by ***dest*** that is sufficiently large to hold both strings after concatenation. The first ***NUL*** character in ***dest*** is overwritten by the first character of ***src*** followed by the remaining characters of ***src***, including the terminating ***NUL***. It is a common mistake to concatenate a string with an uninitialized string (which does not end with ***NUL***)

```
char s[100];
strcat(s,"Flower");
```

what is a mistake: the program will look for a ***NUL*** in ***s***, but will not find any, because array ***s*** has not been initialized.

char* strncat(char* dest, const char* src, size_t n) — is similar to the previous function, but copies at most ***n*** characters of ***src***. Copying terminates when a ***NUL*** character is encountered and copied. If ***n*** characters have been copied and ***NUL*** has not been found, it will be appended as the ***n+1***-th character. Note that in this case ***n+1*** characters will be added to ***dest***! For example

```
char t1[20] = {'\0'}; // NUL necessary!
strncat(t1,"123456789",5);
cout << t1 << " has " << strlen(t1) << " chars" << endl;
```

will print '12345 has 5 chars' (five characters have been copied, no ***NUL*** has been found, so it was appended as the sixth character).

char* strcpy(char* dest, const char* src) — copies characters from the segment of memory starting at address pointed to by ***src*** to the segment pointed to by ***dest***. The destination region is overwritten. Copying terminates when a ***NUL*** has been copied; the user is responsible for the presence of ***NUL*** in the string being copied and for allocating enough space at address ***dest***. The function returns ***dest***. For example, function **strcat** could be implemented in the following way

```
char* strcat(char* dest, const char* src) {
    char* s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

by using **strcpy** and **strlen**.

char* strncpy(char* dest, const char* src, size_t n) — copies exactly ***n*** characters from location starting at the address pointed to by ***src*** to the one pointed to by ***dest***. When ***NUL*** is encounterd, it is copied and then ***NUL*** characters are inserted into the destination string as padding until exactly ***n*** characters have been written. Note that if the length of ***src*** is equal or bigger than ***n***, the terminating ***NUL*** will *not* be copied.

int strcmp(const char* s1, const char* s2) — compares lexicographically (in dictionary order) two strings. Returns -1, if ***s1*** is less (earlier in dictionary order) than ***s2***, +1, if ***s2*** is less, and 0 if they are equal. A string ***s1*** is considered less than ***s2*** if one of the following condition is met:
a) both string are equal up to a certain position, and at the first differing position the character from ***s1*** has value which is less than the value of the character in ***s2*** at the same position;
b) the string ***s1*** is shorter than ***s2*** and has all characters equal to characters from ***s2*** at the corresponding positions.
For example

```
if ( ! strcmp(s1,s2) )
    cout << "Identical\n";
else
    cout << "Different\n";
```

will work, as any nonzero value returned by **strcmp** will be treated as **true** and, after negating with '!' operator, as **false**.
The example below illustrates the use of **strcmp** function. The function **first_last** (lines 20-26) takes a C-string and two pointers of type **char*** — by reference, so they can be modified. It then puts into pointer ***p*** the address of the beginning of a name (an element of the array of strings) which is lexicographically the least, and into ***q*** the address of the lexicographically last name.

P142: sorword.cpp Comparing C-strings

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  void  first_last(char**,char*&,char*&);
6.
7.  int main() {
8.
9.      char *nam[] = { "Cathy",      "Maggi",
10.                      "Alice",      "Wanda",
11.                      "Wendy", "Catharina", "" },
12.                    *p,*q;
13.
14.     first_last(nam,p,q);
15.
16.     cout << "First: " << p << endl
17.          << "Last : " << q << endl;
18. }
19.
20. void first_last(char** s, char*& p, char*& q) {
21.     p = q = *s;
22.     while ( **++s ) {
23.         if ( strcmp(*s, p) < 0 ) p = *s;
24.         if ( strcmp(*s, q) > 0 ) q = *s;
25.     }
26. }
```

Note that the dimension of the array has not been passed to the function. Instead, the last element of the array is an empty string, containing only ***NUL*** character, which will be supllied by the comopiler (line 11). This technique is very often used in C library (for arrays of C-strings). The code of the function **first_last** is rather compact: it could have been longer and easier to comprehend, but its analysis in the form presented is a good excersise on pointers. The program prints 'First: Alice Last : Wendy'.

int strcoll(const char* s1, const char* s2) — as **strcmp**, by takes locale information into account. For example, in Polish locale the letter 'a' is earlier than 'ą', while in French locale diacritic marks do no count in ordering, so 'â' is equivalent to plain 'a'.

int strncmp(const char* s1, const char* s2, size_t n) — as **strcmp**, but only first ***n*** characters are taken into account.

char* strchr(const char* s, int c) — searches for the first occurrence of character `(char)c` in a string ***s*** and returns a pointer to its position or **nullptr** if this character does dot occur in ***s***. For example,

```
cout << strchr("Daniel Defoe",'f') << endl;
```

will print 'foe'. A function counting the number of occurrences of `(char)c` in a string ***s*** could have the form

```
int count(const char* s, int c) {
    int n = 0;
    while (s = strchr(s,c)) ++s,++n;
    return n;
}
```

In particular, one *can* use **strchr** to look for ***NUL*** character.

char* strrchr(const char* s, int c) — as **strchr**, but returns a pointer to the last occurence of `(char)c` in ***s***, or **nullptr**.

size_t strspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that is *not* included in string ***set*** (which is regarded as a *set* of characters — the order of the characters does not matter) and returns the length of the initial segment of ***s*** composed of characters which *are* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strspn("sound and fury","os nudda") << endl;
```

will print 10.

size_t strcspn(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* included in string ***set*** (which is regarded as a *set* of characters) and returns the length of the initial segment of ***s*** composed of characters which are *not* in ***set***. In particular, the result can be equal to 0 or to the full length of ***s***. For example,

```
cout << strcspn("sound and fury","xyztdv") << endl;
```

will print 4.

char* strpbrk(const char* s, const char* set) — searches for the first occurence in ***s*** of a character that *is* in ***set*** and returns a pointer to its location or **nullptr** if not found. For example,

```
cout << strpbrk("Daniel Defoe","wKlor") << endl;
```

will print 'l Defoe'.

char* strstr(const char* s, const char* sub) — returns a pointer to the first character in ***s*** beginning a substring of ***s*** which is identical to ***sub*** (or **nullptr** if substring ***sub*** does not occur in ***s***). For example,

```
cout << strstr("Daniel Defoe","De") << endl;
```

will print 'Defoe'.

char* strtok(char* str, const char* set) — This is a "tokenizer", i.e., a function performing decomposition of a given string into words ("tokens") which are separated one from another by characters from a set ***set*** (which is to be understood as a set of separators, e.g., space, comma, colon etc.). Inner working of the function is based on the existence of an internal pointer of type **char***. Basically, successive calls to **strtok** return successive tokens from the string. When calling it for the first time for a given input string, the string has to be passed as the first argument (***str***). After that, in all successive invocations, ***str*** should be **nullptr**, directing **strtok** to continue from the end of the last token returned. It is not permitted to modify ***str*** between calls, although it is allowable to alter ***set***. The string ***str*** *will* be modified by the function; therefore, it must not be an unmodifiable string, given, e.g., as a literal. The functions works in the following way:

If ***str*** is not **nullptr** (i.e., we start analysing a string), all leading separators from ***str*** are skipped (ignored). If nothing remains, i.e., there are but separators in the string, then the function returns **nullptr** and the internal pointer is set to **nullptr** as well; this terminates processing of a given string. If something remains, then the internal pointer is set to point to the first character encountered that is not a separator, and processing is continued as if ***str*** were **nullptr** (see below).

If ***str*** is **nullptr** and the internal pointer is also **nullptr**, then **nullptr** is returned and the internal pointer is left equal to **nullptr**. This concludes processing of the string.

If ***str*** is **nullptr** but the internal pointer is not **nullptr**, then the functions searches for the first separator, starting from the position currently pointed to by the internal pointer. If found, it is overwritten by an empty character '\0', the function returns current value of the internal pointer, and the pointer itself is set to point to the first character after the '\0' just written. If a separator is not found, the function returns the current value of the internal pointer, and the pointer itself is set to **nullptr**.

All this sounds complicated, but is quite simple to use. For example, the following program

P143: tok.cpp Tokenizer in C

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  int main() {
6.      char strin[] = "int* fun(char& c,double** wtab);";
7.      char separ[] = ")(,;";
8.      char* token;
9.
10.     token = strtok(strin,separ);
11.     while (token != 0) {
12.         cout << token << endl;
13.         token = strtok(0,separ);
14.     }
15. }
```

will print in successive lines

```
int* fun
char& c
double** wtab
```

Implementing functions which operate on C-strings (like those from the header `cstring`) can be a very good programming excercise. For example, some of them can be implemented as follows (and in many other ways):

P144: emulstr.cpp An implementation of some string functions

```
char* Strcpy(char* target, const char* source) {
    char* t = target;
    while ( *t++ = *source++ );
    return target;
}

char* Strcat(char* target, const char* source) {
    char* t = target-1;
    while ( *++t );
    while ( *t++ = *source++ );
    return target;
}

char* Strncat(char* target, const char* source, int n) {
    char* t = target-1;
    while ( *++t );
    while ( (*t++ = *source++) && n--);
    *(t-1) = '\0';
    return target;
}

int Strlen(const char* source) {
    const char* s = source;
    while ( *s++ );
    return s-source-1;
}

char* Strchr(const char* target, int c) {
    while ( *target && *target++ != c );
    return (char*)(*--target ? target : 0);
}
```

It is possible to make these functions even more compact, although not necessarily more comprehensible...

17.1.2 Functions operating on characters

The header `cctype` provides some useful functions operating on characters.

There is a large family of functions that check if a given character meets some criteria. The names of these functions start with `is`; their argument is formally of type **int**, but only the least significant byte, treated as a character, is analysed. They all return an **int**: *any* nonzero value (not necessarily 1) means **true**, zero means **false**. The prototype of these functions has always the form

```
int isSomeProperty(int c);
```

Let us mention the most important of these functions:

- **isalnum** — is `(char)c` a letter or digit?
- **isalpha** — is `(char)c` a letter?
- **isdigit** — is `(char)c` a digit?
- **isxdigit** — is `(char)c` a hexadecimal digit? Hexadecimal digits are digits 0-9 and letters, upper- and lowercase, from the range A-F.
- **iscntrl** — is `(char)c` a control character? Control characters are those with ASCII codes 0-31 inclusive and 127.
- **isprint** — is `(char)c` a printing character? Printing characters are those which are not control characters.
- **isgraph** — is `(char)c` a graphic character? Graphic characters are those which are printing characters (see above), except a space character.
- **ispunct** — is `(char)c` a punctuation character? Punctuation characters are those which are graphic (see above), but are not letters or digits.
- **isspace** — is `(char)c` a whitespace character? Whitespace characters are: '\t' (horizontal tab, symbol HT, ASCII 9), '\r' (carriage return, CR, 13), '\n' (new line, LF, 10), '\v' (vertical tab, VT, 11), '\f' (formfeed, FF, 12) and space (SPC, 32).
- **isupper** — is `(char)c` an uppercase letter?
- **islower** — is `(char)c` a lowercase letter?

Additionally, the header `cctype` provides two useful functions, also of type **int** → **int**:

- **tolower** — for a character which is an uppercase letter returns the corresponding lowercase letter (H → h). For other characters returns its argument.
- **toupper** — for a character which is a lowercase letter returns the corresponding uppercase letter (h → H). For other characters returns its argument.

For example, the function **uplow** in the following program

P145: upplowe.cpp Operations on characters

```
1. #include <iostream>
2. #include <cctype>
3. using namespace std;
4.
5. int uplow(char* s) {
6.     int cnt = 0;
7.     do {
8.         if (isalpha(*s))
9.             if ( cnt == 0 || !isalpha(*(s-1))) {
10.                 *s = (char)toupper(*s);
11.                 cnt++;
12.             } else
13.                 *s = (char)tolower(*s);
14.     } while (*s++);
15.     return cnt;
16. }
17.
18. int main() {
19.     char strn[] = "thiS Is lonG,lONG StRiNg!";
20.
21.     int c = uplow(strn);
22.     cout << c << " words, string = \'" << strn << "\'\n";
23. }
```

changes all first letters of words in a C-string into uppercase, all the remaining into lowercase, and returns the number of words. It prints:
5 words, string = 'This Is Long,Long String!'.

17.1.3 Conversion functions

The header `cstdlib` provides several useful conversion functions. Their usage is simple, but requires a careful error handling; in particular checking the value of ***errno*** which gives information on possible cause of a failure of the last function executed. One can treat ***errno*** as a global variable, although it is normally implemented as a macro (what is easier in multithreaded environment). In order to be able to use ***errno***, one should include the header `cerrno`.

double strtod(const char* str, char ptr)** — (*string to double*) returns a **double** represented by the initial part of a C-string ***str***. Leading whitespaces are skipped and processing of the string terminates when the first "wrong" character is encountered; i.e., a character which cannot be interpreted as a part of a representation of a number. Integer (like 127), scientific (like 1.2E-11) and floating-point with a dot (like 123.34) formats are recognized.

The pointer ***str*** points to a C-string which is supposed to start with a number; other characters after this number are allowed and remain in a stream (if we read from a string stream). The second argument should be the address of a pointer of type **char*** — that is why the type of the corresponding parameter is **char****: pointer to pointer to **char**. On entry, the value of this pointer can be **nullptr** (or just 0), but it is not recommended: it is better to pass as argument the address of an existing pointer variable of type **char***.

After successful conversion, the function returns as a **double** the number which has been read. If ***ptr*** was not **nullptr**, the value of the pointer pointed to by ***ptr*** is set to the address of the first character in the string after the number (this can be the terminating empty character '\0', if there was nothing but a number in the string). If ***ptr*** was **nullptr** on entry, it remains **nullptr** on return. After successful execution of the function, ***errno*** is set to 0.

If string ***str*** does not start with something which can be interpreted as representation of a number, the function returns 0 and the pointer pointed to by ***ptr*** is set to the address of the beginning of ***str***, which is the value of ***str*** itself — this of course happens only if ***ptr*** was not **nullptr**. Therefore, when the function returns 0, we should compare values of ***str*** and of a pointer whose address was passed as the second argument: if they are equal, it is not a "true" zero but a signal that the function failed.

It can happen that ***str*** contains a number, but too large to be represented by a **double** (i.e., it would cause **overflow**). The value ± ***HUGE_VAL***, with the correst sign, is then returned and ***errno*** is set to ***ERANGE***. The constant ***HUGE_VAL*** denotes ***inf*** — infinity. The pointer pointed to by ***ptr*** is set to point to the first character after the number, as if the conversion were successful.

If ***str*** contains a nonzero number but too small to be represented as different from zero (i.e., it causes **underflow**), the function returns zero and ***errno*** is set to ***ERANGE***. The pointer pointed to by ***ptr*** (if not **nullptr**) is set to point to the first character after the number, as if the conversion were successful.

The following program illustrates these definitions:

P146: strtod.cpp Function *strtod*

```
1. #include <iostream>
2. #include <iomanip>   // setw
3. #include <cstdlib>   // strtod
4. #include <cerrno>    // errno
5. using namespace std;
6.
7. int main() {
8.     char*  ptr;
9.     double x;
10.    char*  str;
11.
12.    cout << "ERANGE = " << ERANGE << endl;
13.
14.    // = 1 = OK
15.    str = "-1.2e+2xxx";
16.    x = strtod(str,&ptr);
17.    cout << "=1=  str = " << str << "; x = "
18.         << setw(4) << x << "; errno = " << setw(2)
19.         << errno << "; ptr = " << ptr << endl;
20.
21.    // = 2 =  Not a Number
22.    str = "abcdefghij";
23.    x = strtod(str,&ptr);
24.    cout << "=2=  str = " << str << "; x = "
25.         << setw(4) << x << "; errno = " << setw(2)
26.         << errno << "; ptr = " << ptr << endl;
27.
28.    // = 3 =  Overflow
29.    str = "-9e+9999xx";
30.    x = strtod(str,&ptr);
31.    cout << "=3=  str = " << str << "; x = "
32.         << setw(4) << x << "; errno = " << setw(2)
33.         << errno << "; ptr = " << ptr << endl;
34.
35.    // = 4 =  Underflow
36.    str = "-9e-9999xx";
37.    x = strtod(str,&ptr);
38.    cout << "=4=  str = " << str << "; x = "
39.         << setw(4) << x << "; errno = " << setw(2)
40.         << errno << "; ptr = " << ptr << endl;
41. }
```

Wydruk z tego programu:

```
ERANGE = 34
=1=  str = -1.2e+2xxx; x = -120; errno =  0; ptr = xxx
=2=  str = abcdefghij; x =    0; errno =  0; ptr = abcdefghij
=3=  str = -9e+9999xx; x = -inf; errno = 34; ptr = xx
=4=  str = -9e-9999xx; x =    0; errno = 34; ptr = xx
```

Similar rules apply to the following functions:

long strtol(const char* str, char ptr)** — (*string to long*) as **strtod**, but returns a **long** corresponding to the initial portion of the argument ***str***. In case of overflow returns ***LONG_MAX*** or ***LONG_MIN***. Underflow, of course, cannot happen.

long strtol(const char* str, char ptr, int base)** — as two-argument **strtol**, but expects that the number represented in the initial portion of ***str*** is written with radix ***base***, which can assume values from 2 to 36. The letters 'a' through 'z' (or 'A' to 'Z') represent digits 10 to 36. If ***base*** is 16, the string representation of a number can start with '0x' (or '0X'), which will be ignored. For example, `strtol("J23",&ptr,20)` returns 7643 (= $19 \cdot 400 + 2 \cdot 20 + 3$.

unsigned long strtoul(const char* str, char ptr)** — (*string to unsigned long*) as two-argument **strtol**, but returns an **unsigned long**. In case of overflow returns ***ULONG_MAX***.

unsigned long strtoul(const char* str, char ptr, int base)** — as three-argument **strtol**, but returns an **unsigned long**.

The following traditional conversion functions exist for compatibility, but are not recommended:

double atof(const char* str) — (*ascii to floating*) as **strtod**; in case of overflow returns ± ***HUGE_VAL***, in case of other errors returns zero. In both cases ***errno*** is set to ***ERANGE***.

int atoi(const char* str) — (*ascii to integer*) as **strtol**, but returns an **int**, not a **long**. Error handling similar to that of **atof**.

long atol(const char* str) — (*ascii to long*) as **strtol**; error handling similar to that of **atof**.

Prev: 17. Strings **Up:** 17. Strings **Next:** 17.2 Strings in C++

Subsections

- 17.2.1 Constructors
- 17.2.2 Methods and operators

17.2 Strings in C++

The C++ language supports, of course, C-strings, as they have been implemented in the C language. However, in C++ we have additional tools which provide even more functionality in handling strigs. The standard C++ library defines a class, named **string**, whose objects represent strings equipped with many useful methods to operate on them (in fact, it is a concretization of a more general class template, but this is of no importance for us at the moment).

Unlike C-strings, strings in C++ (objects of the class **string**) can contain any characters; even an empty character '\0' can be used inside a string, not necessarily at the end of it. Strings can also be viewed as collections (of characters) with all consequences of this fact; tools from the standard library which operate on collections can be applied to strings as well. We will mention this possibility in this chapter, but more details will be given in chapter on standard library .

In order to be able to use strings, one has to include the header `string`. The class defines, among other constructs:

- type **string::size_type**, an integer unsigned type which is used as the type for lengths of strings and substrings. The full, qualified name of this type is **string::size_type**, as it is defined in the class **string**.
- a constant ***string::npos***, which is equal to the maximum possible value of a variable of type **string::size_type**. No string can be of this length — the value is used as a signal of errors or as a default value interpreted as *as many as necessary*.

17.2.1 Constructors

An object of class **string** can be created in many ways:

string( )
string(const string& pat, size_type start = 0, size_type cnt = npos)
string(const char* pat)
string(const char* pat, size_type cnt)
string(size_type cnt, char c)
string(const char* start, const char* fin) — are overloaded constructors of class **string**. In the last case, arguments could have been iterators pointing to characters; in the simplest

case these are simply pointers to characters.
For example

```
string s;
```

creates an empty string;

```
string s1 = "Acapulco";
string s2("Acapulco");
```

creates a string initialized with a copy of a C-string;

```
string s("Acapulco",n);
```

creates a string initialized with the first ***n*** characters of C-string passed as the first argument — ***n*** is of type **size_type**;

```
string s(n,'x');
```

creates a string initialized with ***n*** (which is of type **size_type**) repetitions of character passed as the second argument. For ***n***= ***npos***, an exception ***length_error*** is thrown. Note that there is no constructor taking just a single character; one can use `s(1,'x')` instead. If ***s*** is an object of class **string**, then

```
string s1(s);
```

will create a string initialized with a copy of a string ***s*** (the copy-constructor);

```
string s2(s,n);
```

creates a string initialized with a copy of a substring of string ***s*** which starts at its ***n***-th character (counting, as usually, from zero). Argument ***n*** is of type **size_type**. If it is larger or equal than the length of ***s***, exception ***out_of_range*** is thrown. Note that if ***s*** were a C-string, then ***n*** would be interpreted as the number of characters from the beginning of ***s*** which are to be taken as the initializer!

```
string s2(s,n,k);
```

creates a string initialized with a copy of a substring of string ***s*** which starts at its ***n***-th character and has at most ***k*** characters. If the value of ***n+k*** is larger than the length of ***s***, no error occurs: it is interpreted as *all characters starting from position* ***n***. In particular ***k*** ca be ***npos***. Al three constructors above are special cases of one constructor with two default arguments:

```
string(const string& s, size_type start = 0,
                        size_type   ile = npos);
```

There is also a constructor using manifestly the fact that **string** objects are collections of characters; its special case is the constructor taking two pointers to characters: the string will be initialized with characters from the segment of memory which starts at position pointed to by the first pointer (inclusive) up to, but *exclusive*, the character pointed to by the second pointer; for example

```
const char* cstr = "0123456789";
string s(cstr+1,cstr+7);
cout << s << endl;
```

will print '123456'. This is an example of a more general mechanism of iterators, which will be descrobed in section on iterators . For the moment, we can treat iterators as a generalization of pointers. For example

```
string  s("Barcelona");
```

```
string  s1(s.begin()+3, s.end()-2);
cout << s1 << endl;
```

will print 'celo'. Methods **begin** and **end** return iterators (pointers) pointing to the first and the “first after the last” characters of a string. Adding and subtracting integer values for iterators works here as it does for pointers. As usual, the character pointed to by the first iterator is the first included, while that pointed to by the second is the first *excluded*.

17.2.2 Methods and operators

One can use a variety of operators with objects of class **string**. For example, it is possible to assign to a string another string, a character or a C-string:

```
const char* cstr = "strin";
string s1, s2, s3, s(" C++");
s1 = cstr;
s2 = 'g';
s3 = s;
cout << s1 << s2 << s3 << endl;
```

will print 'string C++'. Assignment is deep, i.e., after `s1=s2` the two objects are identical but completely independent; future modifications of ***s1*** will not have any impact on ***s2*** and *vice versa*.

Like in Java, strings can be concatenated with overloaded '+' operator. C-strings, single characters and other strings can be “added” to a string (of course, one of the arguments must be a string). For example:

```
string s1 = "C";
const char* cn = "string";

string s = s1 + '-' + cn;
cout << s << endl;
```

will create and print 'C-string', as the result of the first addition will be a string'C-', which will be then concatenated with a C-string 'string' yielding a string. However, concatenation like this

```
const char* cn = "C";
string s1 = "string";

string s = cn + '-' + s1;
cout << s << endl;
```

will not work, as the first “addition” here is applied to a C-string and a character, but there is no C++ string argument.

The ' `+=`' operator appends its right argument, which can be a string, a C-string or a single character, to the string on the left-hand side.

A string can also be viewed as an array of characters. Indexing works as one could expect:

```
string s("Benny");
s[0] = 'J';
for (int i = 0; i < 5; i++) cout << s[i];
```

will print 'Jenny'; the expression `s[i]` is a reference to the ***i***-th character of the string (counting from zero, as usual). It is not checked if an index falls into legal range; if it does not, behavior of the program is undefined (on the other hand, indexing works much faster

this way).

Operators ' ==', ' !=', ' >', ' >=', ' <', ' <=' work as expected. One can use a C-string on either side (but not both) of an operator; strings are then compared lexicographically. Therefore, in the following program

P147: writer.cpp Sorting strings

```
1. #include <iostream>
2. #include <string>
3. #include <iomanip>
4. using namespace std;
5.
6. void insertionSort(string[],int);
7.
8. int main() {
9.     int i;
10.     string writers[] = {
11.         string("Lampedusa"), string("Shakespeare"),
12.         string("Babel"),     string("Goethe"),
13.         string("Kafka"),     string("Schulz")
14.                         };
15.
16.     const int ile = sizeof(writers)/sizeof(string);
17.
18.     insertionSort(writers, ile);
19.
20.     for ( i = 0; i < ile; i++ )
21.         cout << setw(11) << writers[i] << endl;
22. }
23.
24. void insertionSort(string a[], int size) {
25.     if ( size <= 1 ) return;
26.
27.     for ( int i = 1 ; i < size ; ++i ) {
28.         int j = i;
29.         string v = a[i];
30.         while ( j >= 1 && v < a[j-1] ) {
31.             a[j] = a[j-1];
32.             j--;
33.         }
34.         a[j] = v;
35.     }
36. }
```

string can be treated like numerical values in a sorting procedure; compare this program with **writers.cpp**. The program prints:

```
      Babel
     Goethe
      Kafka
  Lampedusa
     Schulz
Shakespeare
```

Standard stream insertion and extraction operators (' **<<**' and ' **>>**') are overloaded for the class **string**. As usual, the ' **>>**' operator skips leading white characters and terminates reading when a white character is encountered after a string — it is therefore not well suited for reading strings composed of many words.

As a full-fledged class, **string** defines many useful *methods* allowing programmers to manipulate with strings easily. Being methods, they are always called for a specified object of the class:

size_type size( )
size_type length( ) — returns the length of string; e.g., if `s="Joe"`, then `s.size()` is 3.

bool empty( ) — returns a **bool** value answering the question *is this string empty?*

char& at(size_type n) — returns a reference to the ***n***-th character (starting from 0) of the string. The function checks if the value of ***n*** is legal and, if it is not, throws ***out_of_range*** exception. It is therefore safer (but less efficient) than indexing operator. For example, `string("Joe").at(2)` will return a reference to the letter 'e'.

void resize(size_type n, char c = '\0') — changes the size (length) of the string to ***n***. If ***n*** is smaller than the current value of the length, superfluous characters will be lost; if it is larger, then the string is padded with characters ***c*** — '\0' by default.

void clear( ) — makes the string empty; equivalent to invoking **resize(0)**.

string substr(size_type start = 0, size_type cnt = npos) — returns a string which is identical to a substring of the original string starting from position ***start*** and consisting of ***cnt*** characters (or all remaining characters if ***start+cnt*** is larger than the length of the string). For example

```
string s("Pernambuco");
cout << s.substr(5,3) << endl;
```

will print 'mbu'.

size_type copy(char cn[], size_type cnt, size_type start = 0) — copies *into* a C-string ***cn*** a substring of the string starting at position ***start*** (0 by default) and containing at most ***cnt*** characters. Returns the number of characters copied. Does *not* append '\0'. Note the order of arguments ***start*** and ***cnt*** — different than in **substr**. For example

```
char nap[] = "xxxxxx";
string s("Barbara");
string::size_type siz = s.copy(nap,3,2);
cout << "Copied " << siz << " characters: \n"
     <<    nap    << endl;
```

will print 'Copied 3 characters: rbaxxx'.

void swap(string s1) — swaps the contents of *this* string and ***s1***. For example

```
string s("Arles"), s1("Berlin");
s.swap(s1);
cout << s << " " << s1 << endl;
```

prints 'Berlin Arles'.

string& assign(const string& pat)
string& assign(const char* pat)
string& assign(string pat, size_type start, size_type cnt)
string& assign(const char* pat, size_type cnt)
string& assign(size_type cnt, char c)
string& assign(const char* start, const char* fin) — sets the contents of the string and returns a reference to *this* string (acts like an assignment). Arguments are similar to those used in constructors. In the last method, the arguments can be iterators pointing to characters in a collection; in the simplest case they are just pointers to characters in a C-string. Usually, the argument of **assign** is another string or a C-string; e.g., after

```
string s1("xxx"), s2("Betty");
s1.assign(s2);
s2.assign("Cathy");
```

the value of ***s1*** will be 'Betty' and ***s2*** will contain 'Cathy'.
As for constructors, the argument can be a substring of a string or a C-string which starts at a given position and contains a given number of characters (in case of a string, if it is too large, all its characters from the starting position will be taken and there will be no error); e.g.,

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(s1,2,5);
s2.assign(p, 5);
cout << s1 << " " << s2 << endl;
```

prints '23456 01234'.
Using **assign** one can create a string containing ***cnt*** repetitions of a character (the fifth form from those listed above). One can also use pointers to characters in a C-string as iterators defining a substring — as usual, the substring will *not* contain a character specifying the upper limit of the substring:

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(5,'x');
s2.assign(p+3,p+5);
cout << s1 << " " << s2 << endl;
```

will print 'xxxxx 34'.

string& insert(size_type where, const string* pat)
string& insert(size_type where, const char* pat)
string& insert(size_type where, string pat, size_type start, size_type cnt)
string& insert(size_type where, const char* pat, size_type cnt)
string& insert(size_type where, size_type cnt, char c) — modifies *this* string by inserting characters at position ***where***. Returns a reference to *this* string. Characters to be inserted are taken from another string or C-string: the rôle of arguments is similar as for **assign**. Characters starting from the position ***where*** are shifted to the right of the inserted substring; for example

```
string s1("watcher"), s2("likes");
const char* p = "mary";
s1.insert(5,p,2).insert(7,s2,2,1);
cout << s1 << endl;
```

will print 'watchmaker'.
There are three more forms of this method which use iterators:

iterator insert(iterator where, char c)
void insert(iterator where, size_type cnt, char c)
void insert(iterator where, const char* start, const char* fin) — where the first argument is an iterator (generalized pointer) pointing to character before which a substring, specified by all the other arguments, is to be inserted. In the third of these forms, ***start*** and ***fin*** can also be of iterator type (e.g., **string::iterator**). For example

```
string s1("abbccdd");
s1.insert(s1.insert(s1.begin()+5,'c')+1,2,'d');
cout << s1 << endl;
```

prints 'abbcccdddd'.

string& append(const string& pat)
string& append(const string& pat, size_type start, size_type cnt)
string& append(const char* pat)
string& append(const char* pat, size_type cnt)
string& append(size_type cnt, char c)
string& append(const char* start, const char* fin) — modifies string by appending a sequence of characters specified by arguments. Returns a reference to *this* string. Meaning of arguments — as for the previous method. In the last form, iterators can be used instead of character pointers.

string& erase(size_type start = 0, size_type cnt = npos)
iterator erase(iterator start)
iterator erase(iterator start, iterator fin) — erases a fragment of *this* string. Returns a reference to *this* string or an iterator pointing to the first character *after* the fragment which has been removed. The first form removes ***cnt*** characters (***npos*** by default, i.e., all of them) starting from position ***start*** (0 by default). The second form removes all characters from position pointed to by ***start***, and the third form from ***start*** to the character *preceding* that pointed to by ***fin***. For example

```
string s("0123456789");
string::iterator it = s.erase(s.begin()+3,s.end()-3);
cout << s << " " << *it << endl;
```

will print '012789 7'.

string& replace(size_type start, size_type cnt, const string& pat)
string& replace(size_type start, size_type cnt, const string& pat, size_type s, size_type i)
string& replace(size_type start, size_type cnt, const char* pat, size_type i)
string& replace(size_type start, size_type cnt, const char* pat)
string& replace(size_type start, size_type cnt, size_type i, char c)
string& replace(iterator start, iterator fin, const string& pat)
string& replace(iterator start, iterator fin, const char* pat)
string& replace(iterator start, iterator fin, const char* pat, size_type i)
string& replace(iterator start, iterator fin, size_type i, char c)
string& replace(iterator start, iterator fin, const char* st1, const char* kn1) — erases a substring specified by the firts two arguments and substitutes it with a sequence of characters given by all the other arguments (in the last form, iterators can be used instead of pointers). The meaning of arguments is similar as in **insert**. Methods **replace** return a reference to *this* string. For example

```
string s("0123456789");
const char* p("abcdef");
s.replace(0,2,p,2).replace(s.end()-2,s.end(),p+4,p+6);
cout << s << endl;
```

will print 'ab234567ef'.

size_type find(const string* s, size_type start = 0)
size_type find(const char* p, size_type start = 0)
size_type find(const char* p, size_type start, size_type cnt)
size_type find(char c, size_type start = 0)
size_type rfind(const string* s, size_type start = npos)
size_type rfind(const char* p, size_type start = npos)
size_type rfind(const char* p, size_type start, size_type cnt)

size_type rfind(char c, size_type start = npos) — find, starting from position ***start*** in *this* string, the first occurence of a sequence specified by the remaining arguments. Functions from **rfind** family act in a similar way, but searching proceeds backwards. The position (index) of the beginning of the substring found is returned, or ***npos*** if the search failed. In the example below, **find** scans the string 'abc345abcAB', starting from the position 3 (i.e., digit '3'), looking for the first occurence of a sequence composed of first two characters of C-string ***p*** (i.e., 'ab'):

```
string s("abc345abcAB");
const char* p("abcdef");
string::size_type i = s.find(p,3,2);
cout << s.substr(i-1,5) << endl;
```

The fragment would print '5abcA'.

size_type find_first_of(/* args */)
size_type find_last_of(/* args */)
size_type find_first_not_of(/* args */)
size_type find_last_not_of(/* args */) — have types of return value and arguments as the methods from **find**/ **rfind** family. First two methods look, starting from position ***start***, for the first occurence of *any* character belonging to character sequence specified by the remaining arguments. The direction of searching is forward for methods having **_first_** in their name and backwards for those with **_last_**. The methods with **_not_** in their names act in the same way, but look for any character *not* belonging to character sequence specified by the remaining arguments. In all cases the position of the character found is returned, or ***npos*** if the search failed. For example

```
1.  string s("abc123.,!");
2.  const char* p = "!.,?:1234";
3.  string::size_type i = s.find_first_of(p);
4.  string::size_type k = s.find_last_not_of(p,s.size()-1,5);
5.  string s1(s.begin()+i,s.begin()+k+1);
6.  cout << i << " " << k << " " << s1 << endl;
```

will print '3 5 123'. On line 4 we set the second argument to `s.size()-1`, as searching will be backwards, so we have to start from the last character. We take into account only five first characters of the pattern ***p***, i.e., characters '`!.,?:`'. on line 5 we create a new object of class **string** initialized with a slice of string ***s***. We add 1 in the second argument (`s.begin()+k`), as we want the (`s.begin()+k`)-th character to be included.

int compare(const string& pat)
int compare(size_type start, size_type cnt, const string& pat)
int compare(size_type start, size_type cnt, const string& pat, size_type s, size_type cnt1)
int compare(const char* p)
int compare(size_type start, size_type cnt, const char* p, size_type i = npos) — compares *this* string, or its substring specified by ***start*** and ***cnt***, with a sequence of characters defined by the remaining arguments. Returns `-1`, if *this* string is lexicographically earlier, zero if it is equal, or `+1` if it lexicographically follows the sequence.

void push_back(char c) — appends character ***c*** to *this* string. Invocation `s.push_back(c)` acts as `s.insert(s.end(),c)` but does not return any value (**insert** would return an iterator). Same effect can be obtained by invoking **append** or '`+=`' operator.

const char* c_str( ) — returns a pointer to a constant C-string with the same contents as

this string (with terminating ' \0'). The C-string returned cannot be modified.

iterator begin( ) — returns an iterator (see section on iterators) pointing to the first character of the string.

iterator end( ) — returns an iterator pointing to the first after the last character of the string.

reverse_iterator rbegin( )
reverse_iterator rend( ) — as **begin** and **end** but returns reverse iterator; e.g., after

```
string s1("ABCD");
string s2(s1.rbegin(),s1.rend());
```

s2 will be 'DCBA'.

After including the header file `string`, we also have a very useful function **getline** (which is *not* a method of class **string**):

istream& getline(istream& str, string& s)
istream& getline(istream& str, string& s, char eol) — reads one line of text from input stream ***str*** into string ***s***. The line *can* contain white characters other than end-of-line character. The end-of-line character itself is extracted from the stream but is *not* put into ***s***. If a character is given as the third argument, it will play the rôle of the end-of-line character. The method returns a reference to the stream ***str***, so a code like

```
string s1,s2;
getline(cin,s1) >> s2;
```

will work and will put the first line into ***s1*** and the first word of the next line into ***s2***.

Subsections

- 17.2.1 Constructors
- 17.2.2 Methods and operators

17.2 Strings in C++

The C++ language supports, of course, C-strings, as they have been implemented in the C language. However, in C++ we have additional tools which provide even more functionality in handling strigs. The standard C++ library defines a class, named **string**, whose objects represent strings equipped with many useful methods to operate on them (in fact, it is a concretization of a more general class template, but this is of no importance for us at the moment).

Unlike C-strings, strings in C++ (objects of the class **string**) can contain any characters; even an empty character '\0' can be used inside a string, not necessarily at the end of it. Strings can also be viewed as collections (of characters) with all consequences of this fact; tools from the standard library which operate on collections can be applied to strings as well. We will mention this possibility in this chapter, but more details will be given in chapter on standard library .

In order to be able to use strings, one has to include the header `string`. The class defines, among other constructs:

- type **string::size_type**, an integer unsigned type which is used as the type for lengths of strings and substrings. The full, qualified name of this type is **string::size_type**, as it is defined in the class **string**.
- a constant ***string::npos***, which is equal to the maximum possible value of a variable of type **string::size_type**. No string can be of this length — the value is used as a signal of errors or as a default value interpreted as *as many as necessary*.

17.2.1 Constructors

An object of class **string** can be created in many ways:

string( )
string(const string& pat, size_type start = 0, size_type cnt = npos)
string(const char* pat)
string(const char* pat, size_type cnt)
string(size_type cnt, char c)
string(const char* start, const char* fin) — are overloaded constructors of class **string**. In the last case, arguments could have been iterators pointing to characters; in the simplest

case these are simply pointers to characters.
For example

```
string s;
```

creates an empty string;

```
string s1 = "Acapulco";
string s2("Acapulco");
```

creates a string initialized with a copy of a C-string;

```
string s("Acapulco",n);
```

creates a string initialized with the first ***n*** characters of C-string passed as the first argument — ***n*** is of type **size_type**;

```
string s(n,'x');
```

creates a string initialized with ***n*** (which is of type **size_type**) repetitions of character passed as the second argument. For ***n***= ***npos***, an exception ***length_error*** is thrown. Note that there is no constructor taking just a single character; one can use `s(1,'x')` instead. If ***s*** is an object of class **string**, then

```
string s1(s);
```

will create a string initialized with a copy of a string ***s*** (the copy-constructor);

```
string s2(s,n);
```

creates a string initialized with a copy of a substring of string ***s*** which starts at its ***n***-th character (counting, as usually, from zero). Argument ***n*** is of type **size_type**. If it is larger or equal than the length of ***s***, exception ***out_of_range*** is thrown. Note that if ***s*** were a C-string, then ***n*** would be interpreted as the number of characters from the beginning of ***s*** which are to be taken as the initializer!

```
string s2(s,n,k);
```

creates a string initialized with a copy of a substring of string ***s*** which starts at its ***n***-th character and has at most ***k*** characters. If the value of ***n+k*** is larger than the length of ***s***, no error occurs: it is interpreted as *all characters starting from position* ***n***. In particular ***k*** ca be ***npos***. Al three constructors above are special cases of one constructor with two default arguments:

```
string(const string& s, size_type start = 0,
                        size_type   ile = npos);
```

There is also a constructor using manifestly the fact that **string** objects are collections of characters; its special case is the constructor taking two pointers to characters: the string will be initialized with characters from the segment of memory which starts at position pointed to by the first pointer (inclusive) up to, but *exclusive*, the character pointed to by the second pointer; for example

```
const char* cstr = "0123456789";
string s(cstr+1,cstr+7);
cout << s << endl;
```

will print '123456'. This is an example of a more general mechanism of iterators, which will be descrobed in section on iterators . For the moment, we can treat iterators as a generalization of pointers. For example

```
string  s("Barcelona");
```

```
string  s1(s.begin()+3, s.end()-2);
cout << s1 << endl;
```

will print 'celo'. Methods **begin** and **end** return iterators (pointers) pointing to the first and the “first after the last” characters of a string. Adding and subtracting integer values for iterators works here as it does for pointers. As usual, the character pointed to by the first iterator is the first included, while that pointed to by the second is the first *excluded*.

17.2.2 Methods and operators

One can use a variety of operators with objects of class **string**. For example, it is possible to assign to a string another string, a character or a C-string:

```
const char* cstr = "strin";
string s1, s2, s3, s(" C++");
s1 = cstr;
s2 = 'g';
s3 = s;
cout << s1 << s2 << s3 << endl;
```

will print 'string C++'. Assignment is deep, i.e., after `s1=s2` the two objects are identical but completely independent; future modifications of ***s1*** will not have any impact on ***s2*** and *vice versa*.

Like in Java, strings can be concatenated with overloaded '+' operator. C-strings, single characters and other strings can be “added” to a string (of course, one of the arguments must be a string). For example:

```
string s1 = "C";
const char* cn = "string";

string s = s1 + '-' + cn;
cout << s << endl;
```

will create and print 'C-string', as the result of the first addition will be a string'C-', which will be then concatenated with a C-string 'string' yielding a string. However, concatenation like this

```
const char* cn = "C";
string s1 = "string";

string s = cn + '-' + s1;
cout << s << endl;
```

will not work, as the first “addition” here is applied to a C-string and a character, but there is no C++ string argument.

The ' `+=`' operator appends its right argument, which can be a string, a C-string or a single character, to the string on the left-hand side.

A string can also be viewed as an array of characters. Indexing works as one could expect:

```
string s("Benny");
s[0] = 'J';
for (int i = 0; i < 5; i++) cout << s[i];
```

will print 'Jenny'; the expression `s[i]` is a reference to the ***i***-th character of the string (counting from zero, as usual). It is not checked if an index falls into legal range; if it does not, behavior of the program is undefined (on the other hand, indexing works much faster

this way).

Operators ' ==', ' !=', ' >', ' >=', ' <', ' <=' work as expected. One can use a C-string on either side (but not both) of an operator; strings are then compared lexicographically. Therefore, in the following program

P147: writer.cpp Sorting strings

```
1. #include <iostream>
2. #include <string>
3. #include <iomanip>
4. using namespace std;
5.
6. void insertionSort(string[],int);
7.
8. int main() {
9.     int i;
10.     string writers[] = {
11.         string("Lampedusa"), string("Shakespeare"),
12.         string("Babel"),     string("Goethe"),
13.         string("Kafka"),     string("Schulz")
14.                         };
15.
16.     const int ile = sizeof(writers)/sizeof(string);
17.
18.     insertionSort(writers, ile);
19.
20.     for ( i = 0; i < ile; i++ )
21.         cout << setw(11) << writers[i] << endl;
22. }
23.
24. void insertionSort(string a[], int size) {
25.     if ( size <= 1 ) return;
26.
27.     for ( int i = 1 ; i < size ; ++i ) {
28.         int j = i;
29.         string v = a[i];
30.         while ( j >= 1 && v < a[j-1] ) {
31.             a[j] = a[j-1];
32.             j--;
33.         }
34.         a[j] = v;
35.     }
36. }
```

string can be treated like numerical values in a sorting procedure; compare this program with **writers.cpp**. The program prints:

```
      Babel
     Goethe
      Kafka
  Lampedusa
     Schulz
Shakespeare
```

Standard stream insertion and extraction operators (' **<<**' and ' **>>**') are overloaded for the class **string**. As usual, the ' **>>**' operator skips leading white characters and terminates reading when a white character is encountered after a string — it is therefore not well suited for reading strings composed of many words.

As a full-fledged class, **string** defines many useful *methods* allowing programmers to manipulate with strings easily. Being methods, they are always called for a specified object of the class:

size_type size( )
size_type length( ) — returns the length of string; e.g., if `s="Joe"`, then `s.size()` is 3.

bool empty( ) — returns a **bool** value answering the question *is this string empty?*

char& at(size_type n) — returns a reference to the ***n***-th character (starting from 0) of the string. The function checks if the value of ***n*** is legal and, if it is not, throws ***out_of_range*** exception. It is therefore safer (but less efficient) than indexing operator. For example, `string("Joe").at(2)` will return a reference to the letter 'e'.

void resize(size_type n, char c = '\0') — changes the size (length) of the string to ***n***. If ***n*** is smaller than the current value of the length, superfluous characters will be lost; if it is larger, then the string is padded with characters ***c*** — '\0' by default.

void clear( ) — makes the string empty; equivalent to invoking **resize(0)**.

string substr(size_type start = 0, size_type cnt = npos) — returns a string which is identical to a substring of the original string starting from position ***start*** and consisting of ***cnt*** characters (or all remaining characters if ***start+cnt*** is larger than the length of the string). For example

```
string s("Pernambuco");
cout << s.substr(5,3) << endl;
```

will print 'mbu'.

size_type copy(char cn[], size_type cnt, size_type start = 0) — copies *into* a C-string ***cn*** a substring of the string starting at position ***start*** (0 by default) and containing at most ***cnt*** characters. Returns the number of characters copied. Does *not* append '\0'. Note the order of arguments ***start*** and ***cnt*** — different than in **substr**. For example

```
char nap[] = "xxxxxx";
string s("Barbara");
string::size_type siz = s.copy(nap,3,2);
cout << "Copied " << siz << " characters: \n"
     <<    nap    << endl;
```

will print 'Copied 3 characters: rbaxxx'.

void swap(string s1) — swaps the contents of *this* string and ***s1***. For example

```
string s("Arles"), s1("Berlin");
s.swap(s1);
cout << s << " " << s1 << endl;
```

prints 'Berlin Arles'.

string& assign(const string& pat)
string& assign(const char* pat)
string& assign(string pat, size_type start, size_type cnt)
string& assign(const char* pat, size_type cnt)
string& assign(size_type cnt, char c)
string& assign(const char* start, const char* fin) — sets the contents of the string and returns a reference to *this* string (acts like an assignment). Arguments are similar to those used in constructors. In the last method, the arguments can be iterators pointing to characters in a collection; in the simplest case they are just pointers to characters in a C-string. Usually, the argument of **assign** is another string or a C-string; e.g., after

```
string s1("xxx"), s2("Betty");
s1.assign(s2);
s2.assign("Cathy");
```

the value of ***s1*** will be 'Betty' and ***s2*** will contain 'Cathy'.
As for constructors, the argument can be a substring of a string or a C-string which starts at a given position and contains a given number of characters (in case of a string, if it is too large, all its characters from the starting position will be taken and there will be no error); e.g.,

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(s1,2,5);
s2.assign(p, 5);
cout << s1 << " " << s2 << endl;
```

prints '23456 01234'.
Using **assign** one can create a string containing ***cnt*** repetitions of a character (the fifth form from those listed above). One can also use pointers to characters in a C-string as iterators defining a substring — as usual, the substring will *not* contain a character specifying the upper limit of the substring:

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(5,'x');
s2.assign(p+3,p+5);
cout << s1 << " " << s2 << endl;
```

will print 'xxxxx 34'.

string& insert(size_type where, const string* pat)
string& insert(size_type where, const char* pat)
string& insert(size_type where, string pat, size_type start, size_type cnt)
string& insert(size_type where, const char* pat, size_type cnt)
string& insert(size_type where, size_type cnt, char c) — modifies *this* string by inserting characters at position ***where***. Returns a reference to *this* string. Characters to be inserted are taken from another string or C-string: the rôle of arguments is similar as for **assign**. Characters starting from the position ***where*** are shifted to the right of the inserted substring; for example

```
string s1("watcher"), s2("likes");
const char* p = "mary";
s1.insert(5,p,2).insert(7,s2,2,1);
cout << s1 << endl;
```

will print 'watchmaker'.
There are three more forms of this method which use iterators:

iterator insert(iterator where, char c)
void insert(iterator where, size_type cnt, char c)
void insert(iterator where, const char* start, const char* fin) — where the first argument is an iterator (generalized pointer) pointing to character before which a substring, specified by all the other arguments, is to be inserted. In the third of these forms, ***start*** and ***fin*** can also be of iterator type (e.g., **string::iterator**). For example

```
string s1("abbccdd");
s1.insert(s1.insert(s1.begin()+5,'c')+1,2,'d');
cout << s1 << endl;
```

prints 'abbcccdddd'.

string& append(const string& pat)
string& append(const string& pat, size_type start, size_type cnt)
string& append(const char* pat)
string& append(const char* pat, size_type cnt)
string& append(size_type cnt, char c)
string& append(const char* start, const char* fin) — modifies string by appending a sequence of characters specified by arguments. Returns a reference to *this* string. Meaning of arguments — as for the previous method. In the last form, iterators can be used instead of character pointers.

string& erase(size_type start = 0, size_type cnt = npos)
iterator erase(iterator start)
iterator erase(iterator start, iterator fin) — erases a fragment of *this* string. Returns a reference to *this* string or an iterator pointing to the first character *after* the fragment which has been removed. The first form removes ***cnt*** characters (***npos*** by default, i.e., all of them) starting from position ***start*** (0 by default). The second form removes all characters from position pointed to by ***start***, and the third form from ***start*** to the character *preceding* that pointed to by ***fin***. For example

```
string s("0123456789");
string::iterator it = s.erase(s.begin()+3,s.end()-3);
cout << s << " " << *it << endl;
```

will print '012789 7'.

string& replace(size_type start, size_type cnt, const string& pat)
string& replace(size_type start, size_type cnt, const string& pat, size_type s, size_type i)
string& replace(size_type start, size_type cnt, const char* pat, size_type i)
string& replace(size_type start, size_type cnt, const char* pat)
string& replace(size_type start, size_type cnt, size_type i, char c)
string& replace(iterator start, iterator fin, const string& pat)
string& replace(iterator start, iterator fin, const char* pat)
string& replace(iterator start, iterator fin, const char* pat, size_type i)
string& replace(iterator start, iterator fin, size_type i, char c)
string& replace(iterator start, iterator fin, const char* st1, const char* kn1) — erases a substring specified by the firts two arguments and substitutes it with a sequence of characters given by all the other arguments (in the last form, iterators can be used instead of pointers). The meaning of arguments is similar as in **insert**. Methods **replace** return a reference to *this* string. For example

```
string s("0123456789");
const char* p("abcdef");
s.replace(0,2,p,2).replace(s.end()-2,s.end(),p+4,p+6);
cout << s << endl;
```

will print 'ab234567ef'.

size_type find(const string* s, size_type start = 0)
size_type find(const char* p, size_type start = 0)
size_type find(const char* p, size_type start, size_type cnt)
size_type find(char c, size_type start = 0)
size_type rfind(const string* s, size_type start = npos)
size_type rfind(const char* p, size_type start = npos)
size_type rfind(const char* p, size_type start, size_type cnt)

size_type rfind(char c, size_type start = npos) — find, starting from position ***start*** in *this* string, the first occurence of a sequence specified by the remaining arguments. Functions from **rfind** family act in a similar way, but searching proceeds backwards. The position (index) of the beginning of the substring found is returned, or ***npos*** if the search failed. In the example below, **find** scans the string 'abc345abcAB', starting from the position 3 (i.e., digit '3'), looking for the first occurence of a sequence composed of first two characters of C-string ***p*** (i.e., 'ab'):

```
string s("abc345abcAB");
const char* p("abcdef");
string::size_type i = s.find(p,3,2);
cout << s.substr(i-1,5) << endl;
```

The fragment would print '5abcA'.

size_type find_first_of(/* args */)
size_type find_last_of(/* args */)
size_type find_first_not_of(/* args */)
size_type find_last_not_of(/* args */) — have types of return value and arguments as the methods from **find**/ **rfind** family. First two methods look, starting from position ***start***, for the first occurence of *any* character belonging to character sequence specified by the remaining arguments. The direction of searching is forward for methods having **_first_** in their name and backwards for those with **_last_**. The methods with **_not_** in their names act in the same way, but look for any character *not* belonging to character sequence specified by the remaining arguments. In all cases the position of the character found is returned, or ***npos*** if the search failed. For example

```
1.  string s("abc123.,!");
2.  const char* p = "!.,?:1234";
3.  string::size_type i = s.find_first_of(p);
4.  string::size_type k = s.find_last_not_of(p,s.size()-1,5);
5.  string s1(s.begin()+i,s.begin()+k+1);
6.  cout << i << " " << k << " " << s1 << endl;
```

will print '3 5 123'. On line 4 we set the second argument to `s.size()-1`, as searching will be backwards, so we have to start from the last character. We take into account only five first characters of the pattern ***p***, i.e., characters '`!.,?:`'. on line 5 we create a new object of class **string** initialized with a slice of string ***s***. We add 1 in the second argument (`s.begin()+k`), as we want the (`s.begin()+k`)-th character to be included.

int compare(const string& pat)
int compare(size_type start, size_type cnt, const string& pat)
int compare(size_type start, size_type cnt, const string& pat, size_type s, size_type cnt1)
int compare(const char* p)
int compare(size_type start, size_type cnt, const char* p, size_type i = npos) — compares *this* string, or its substring specified by ***start*** and ***cnt***, with a sequence of characters defined by the remaining arguments. Returns `-1`, if *this* string is lexicographically earlier, zero if it is equal, or `+1` if it lexicographically follows the sequence.

void push_back(char c) — appends character ***c*** to *this* string. Invocation `s.push_back(c)` acts as `s.insert(s.end(),c)` but does not return any value (**insert** would return an iterator). Same effect can be obtained by invoking **append** or '`+=`' operator.

const char* c_str( ) — returns a pointer to a constant C-string with the same contents as

this string (with terminating ' \0'). The C-string returned cannot be modified.

iterator begin( ) — returns an iterator (see section on iterators) pointing to the first character of the string.

iterator end( ) — returns an iterator pointing to the first after the last character of the string.

reverse_iterator rbegin( )
reverse_iterator rend( ) — as **begin** and **end** but returns reverse iterator; e.g., after

```
string s1("ABCD");
string s2(s1.rbegin(),s1.rend());
```

s2 will be 'DCBA'.

After including the header file `string`, we also have a very useful function **getline** (which is *not* a method of class **string**):

istream& getline(istream& str, string& s)
istream& getline(istream& str, string& s, char eol) — reads one line of text from input stream ***str*** into string ***s***. The line *can* contain white characters other than end-of-line character. The end-of-line character itself is extracted from the stream but is *not* put into ***s***. If a character is given as the third argument, it will play the rôle of the end-of-line character. The method returns a reference to the stream ***str***, so a code like

```
string s1,s2;
getline(cin,s1) >> s2;
```

will work and will put the first line into ***s1*** and the first word of the next line into ***s2***.

Subsections

- 17.2.1 Constructors
- 17.2.2 Methods and operators

17.2 Strings in C++

The C++ language supports, of course, C-strings, as they have been implemented in the C language. However, in C++ we have additional tools which provide even more functionality in handling strigs. The standard C++ library defines a class, named **string**, whose objects represent strings equipped with many useful methods to operate on them (in fact, it is a concretization of a more general class template, but this is of no importance for us at the moment).

Unlike C-strings, strings in C++ (objects of the class **string**) can contain any characters; even an empty character '\0' can be used inside a string, not necessarily at the end of it. Strings can also be viewed as collections (of characters) with all consequences of this fact; tools from the standard library which operate on collections can be applied to strings as well. We will mention this possibility in this chapter, but more details will be given in chapter on standard library .

In order to be able to use strings, one has to include the header `string`. The class defines, among other constructs:

- type **string::size_type**, an integer unsigned type which is used as the type for lengths of strings and substrings. The full, qualified name of this type is **string::size_type**, as it is defined in the class **string**.
- a constant ***string::npos***, which is equal to the maximum possible value of a variable of type **string::size_type**. No string can be of this length — the value is used as a signal of errors or as a default value interpreted as *as many as necessary*.

17.2.1 Constructors

An object of class **string** can be created in many ways:

string( )
string(const string& pat, size_type start = 0, size_type cnt = npos)
string(const char* pat)
string(const char* pat, size_type cnt)
string(size_type cnt, char c)
string(const char* start, const char* fin) — are overloaded constructors of class **string**. In the last case, arguments could have been iterators pointing to characters; in the simplest

case these are simply pointers to characters.
For example

```
string s;
```

creates an empty string;

```
string s1 = "Acapulco";
string s2("Acapulco");
```

creates a string initialized with a copy of a C-string;

```
string s("Acapulco",n);
```

creates a string initialized with the first ***n*** characters of C-string passed as the first argument — ***n*** is of type **size_type**;

```
string s(n,'x');
```

creates a string initialized with ***n*** (which is of type **size_type**) repetitions of character passed as the second argument. For ***n***= ***npos***, an exception ***length_error*** is thrown. Note that there is no constructor taking just a single character; one can use `s(1,'x')` instead. If ***s*** is an object of class **string**, then

```
string s1(s);
```

will create a string initialized with a copy of a string ***s*** (the copy-constructor);

```
string s2(s,n);
```

creates a string initialized with a copy of a substring of string ***s*** which starts at its ***n***-th character (counting, as usually, from zero). Argument ***n*** is of type **size_type**. If it is larger or equal than the length of ***s***, exception ***out_of_range*** is thrown. Note that if ***s*** were a C-string, then ***n*** would be interpreted as the number of characters from the beginning of ***s*** which are to be taken as the initializer!

```
string s2(s,n,k);
```

creates a string initialized with a copy of a substring of string ***s*** which starts at its ***n***-th character and has at most ***k*** characters. If the value of ***n+k*** is larger than the length of ***s***, no error occurs: it is interpreted as *all characters starting from position* ***n***. In particular ***k*** ca be ***npos***. Al three constructors above are special cases of one constructor with two default arguments:

```
string(const string& s, size_type start = 0,
                        size_type   ile = npos);
```

There is also a constructor using manifestly the fact that **string** objects are collections of characters; its special case is the constructor taking two pointers to characters: the string will be initialized with characters from the segment of memory which starts at position pointed to by the first pointer (inclusive) up to, but *exclusive*, the character pointed to by the second pointer; for example

```
const char* cstr = "0123456789";
string s(cstr+1,cstr+7);
cout << s << endl;
```

will print '123456'. This is an example of a more general mechanism of iterators, which will be descrobed in section on iterators . For the moment, we can treat iterators as a generalization of pointers. For example

```
string  s("Barcelona");
```

```
string  s1(s.begin()+3, s.end()-2);
cout << s1 << endl;
```

will print 'celo'. Methods **begin** and **end** return iterators (pointers) pointing to the first and the “first after the last” characters of a string. Adding and subtracting integer values for iterators works here as it does for pointers. As usual, the character pointed to by the first iterator is the first included, while that pointed to by the second is the first *excluded*.

17.2.2 Methods and operators

One can use a variety of operators with objects of class **string**. For example, it is possible to assign to a string another string, a character or a C-string:

```
const char* cstr = "strin";
string s1, s2, s3, s(" C++");
s1 = cstr;
s2 = 'g';
s3 = s;
cout << s1 << s2 << s3 << endl;
```

will print 'string C++'. Assignment is deep, i.e., after `s1=s2` the two objects are identical but completely independent; future modifications of ***s1*** will not have any impact on ***s2*** and *vice versa*.

Like in Java, strings can be concatenated with overloaded '+' operator. C-strings, single characters and other strings can be “added” to a string (of course, one of the arguments must be a string). For example:

```
string s1 = "C";
const char* cn = "string";

string s = s1 + '-' + cn;
cout << s << endl;
```

will create and print 'C-string', as the result of the first addition will be a string'C-', which will be then concatenated with a C-string 'string' yielding a string. However, concatenation like this

```
const char* cn = "C";
string s1 = "string";

string s = cn + '-' + s1;
cout << s << endl;
```

will not work, as the first “addition” here is applied to a C-string and a character, but there is no C++ string argument.

The ' `+=`' operator appends its right argument, which can be a string, a C-string or a single character, to the string on the left-hand side.

A string can also be viewed as an array of characters. Indexing works as one could expect:

```
string s("Benny");
s[0] = 'J';
for (int i = 0; i < 5; i++) cout << s[i];
```

will print 'Jenny'; the expression `s[i]` is a reference to the ***i***-th character of the string (counting from zero, as usual). It is not checked if an index falls into legal range; if it does not, behavior of the program is undefined (on the other hand, indexing works much faster

this way).

Operators ' ==', ' !=', ' >', ' >=', ' <', ' <=' work as expected. One can use a C-string on either side (but not both) of an operator; strings are then compared lexicographically. Therefore, in the following program

P147: writer.cpp Sorting strings

```
1. #include <iostream>
2. #include <string>
3. #include <iomanip>
4. using namespace std;
5.
6. void insertionSort(string[],int);
7.
8. int main() {
9.     int i;
10.     string writers[] = {
11.         string("Lampedusa"), string("Shakespeare"),
12.         string("Babel"),     string("Goethe"),
13.         string("Kafka"),     string("Schulz")
14.                         };
15.
16.     const int ile = sizeof(writers)/sizeof(string);
17.
18.     insertionSort(writers, ile);
19.
20.     for ( i = 0; i < ile; i++ )
21.         cout << setw(11) << writers[i] << endl;
22. }
23.
24. void insertionSort(string a[], int size) {
25.     if ( size <= 1 ) return;
26.
27.     for ( int i = 1 ; i < size ; ++i ) {
28.         int j = i;
29.         string v = a[i];
30.         while ( j >= 1 && v < a[j-1] ) {
31.             a[j] = a[j-1];
32.             j--;
33.         }
34.         a[j] = v;
35.     }
36. }
```

string can be treated like numerical values in a sorting procedure; compare this program with **writers.cpp**. The program prints:

```
      Babel
     Goethe
      Kafka
  Lampedusa
     Schulz
Shakespeare
```

Standard stream insertion and extraction operators (' **<<**' and ' **>>**') are overloaded for the class **string**. As usual, the ' **>>**' operator skips leading white characters and terminates reading when a white character is encountered after a string — it is therefore not well suited for reading strings composed of many words.

As a full-fledged class, **string** defines many useful *methods* allowing programmers to manipulate with strings easily. Being methods, they are always called for a specified object of the class:

size_type size( )
size_type length( ) — returns the length of string; e.g., if `s="Joe"`, then `s.size()` is 3.

bool empty( ) — returns a **bool** value answering the question *is this string empty?*

char& at(size_type n) — returns a reference to the ***n***-th character (starting from 0) of the string. The function checks if the value of ***n*** is legal and, if it is not, throws ***out_of_range*** exception. It is therefore safer (but less efficient) than indexing operator. For example, `string("Joe").at(2)` will return a reference to the letter 'e'.

void resize(size_type n, char c = '\0') — changes the size (length) of the string to ***n***. If ***n*** is smaller than the current value of the length, superfluous characters will be lost; if it is larger, then the string is padded with characters ***c*** — '\0' by default.

void clear( ) — makes the string empty; equivalent to invoking **resize(0)**.

string substr(size_type start = 0, size_type cnt = npos) — returns a string which is identical to a substring of the original string starting from position ***start*** and consisting of ***cnt*** characters (or all remaining characters if ***start+cnt*** is larger than the length of the string). For example

```
string s("Pernambuco");
cout << s.substr(5,3) << endl;
```

will print 'mbu'.

size_type copy(char cn[], size_type cnt, size_type start = 0) — copies *into* a C-string ***cn*** a substring of the string starting at position ***start*** (0 by default) and containing at most ***cnt*** characters. Returns the number of characters copied. Does *not* append '\0'. Note the order of arguments ***start*** and ***cnt*** — different than in **substr**. For example

```
char nap[] = "xxxxxx";
string s("Barbara");
string::size_type siz = s.copy(nap,3,2);
cout << "Copied " << siz << " characters: \n"
     <<    nap    << endl;
```

will print 'Copied 3 characters: rbaxxx'.

void swap(string s1) — swaps the contents of *this* string and ***s1***. For example

```
string s("Arles"), s1("Berlin");
s.swap(s1);
cout << s << " " << s1 << endl;
```

prints 'Berlin Arles'.

string& assign(const string& pat)
string& assign(const char* pat)
string& assign(string pat, size_type start, size_type cnt)
string& assign(const char* pat, size_type cnt)
string& assign(size_type cnt, char c)
string& assign(const char* start, const char* fin) — sets the contents of the string and returns a reference to *this* string (acts like an assignment). Arguments are similar to those used in constructors. In the last method, the arguments can be iterators pointing to characters in a collection; in the simplest case they are just pointers to characters in a C-string. Usually, the argument of **assign** is another string or a C-string; e.g., after

```
string s1("xxx"), s2("Betty");
s1.assign(s2);
s2.assign("Cathy");
```

the value of ***s1*** will be 'Betty' and ***s2*** will contain 'Cathy'.
As for constructors, the argument can be a substring of a string or a C-string which starts at a given position and contains a given number of characters (in case of a string, if it is too large, all its characters from the starting position will be taken and there will be no error); e.g.,

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(s1,2,5);
s2.assign(p, 5);
cout << s1 << " " << s2 << endl;
```

prints '23456 01234'.
Using **assign** one can create a string containing ***cnt*** repetitions of a character (the fifth form from those listed above). One can also use pointers to characters in a C-string as iterators defining a substring — as usual, the substring will *not* contain a character specifying the upper limit of the substring:

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(5,'x');
s2.assign(p+3,p+5);
cout << s1 << " " << s2 << endl;
```

will print 'xxxxx 34'.

string& insert(size_type where, const string* pat)
string& insert(size_type where, const char* pat)
string& insert(size_type where, string pat, size_type start, size_type cnt)
string& insert(size_type where, const char* pat, size_type cnt)
string& insert(size_type where, size_type cnt, char c) — modifies *this* string by inserting characters at position ***where***. Returns a reference to *this* string. Characters to be inserted are taken from another string or C-string: the rôle of arguments is similar as for **assign**. Characters starting from the position ***where*** are shifted to the right of the inserted substring; for example

```
string s1("watcher"), s2("likes");
const char* p = "mary";
s1.insert(5,p,2).insert(7,s2,2,1);
cout << s1 << endl;
```

will print 'watchmaker'.
There are three more forms of this method which use iterators:

iterator insert(iterator where, char c)
void insert(iterator where, size_type cnt, char c)
void insert(iterator where, const char* start, const char* fin) — where the first argument is an iterator (generalized pointer) pointing to character before which a substring, specified by all the other arguments, is to be inserted. In the third of these forms, ***start*** and ***fin*** can also be of iterator type (e.g., **string::iterator**). For example

```
string s1("abbccdd");
s1.insert(s1.insert(s1.begin()+5,'c')+1,2,'d');
cout << s1 << endl;
```

prints 'abbcccdddd'.

string& append(const string& pat)
string& append(const string& pat, size_type start, size_type cnt)
string& append(const char* pat)
string& append(const char* pat, size_type cnt)
string& append(size_type cnt, char c)
string& append(const char* start, const char* fin) — modifies string by appending a sequence of characters specified by arguments. Returns a reference to *this* string. Meaning of arguments — as for the previous method. In the last form, iterators can be used instead of character pointers.

string& erase(size_type start = 0, size_type cnt = npos)
iterator erase(iterator start)
iterator erase(iterator start, iterator fin) — erases a fragment of *this* string. Returns a reference to *this* string or an iterator pointing to the first character *after* the fragment which has been removed. The first form removes ***cnt*** characters (***npos*** by default, i.e., all of them) starting from position ***start*** (0 by default). The second form removes all characters from position pointed to by ***start***, and the third form from ***start*** to the character *preceding* that pointed to by ***fin***. For example

```
string s("0123456789");
string::iterator it = s.erase(s.begin()+3,s.end()-3);
cout << s << " " << *it << endl;
```

will print '012789 7'.

string& replace(size_type start, size_type cnt, const string& pat)
string& replace(size_type start, size_type cnt, const string& pat, size_type s, size_type i)
string& replace(size_type start, size_type cnt, const char* pat, size_type i)
string& replace(size_type start, size_type cnt, const char* pat)
string& replace(size_type start, size_type cnt, size_type i, char c)
string& replace(iterator start, iterator fin, const string& pat)
string& replace(iterator start, iterator fin, const char* pat)
string& replace(iterator start, iterator fin, const char* pat, size_type i)
string& replace(iterator start, iterator fin, size_type i, char c)
string& replace(iterator start, iterator fin, const char* st1, const char* kn1) — erases a substring specified by the firts two arguments and substitutes it with a sequence of characters given by all the other arguments (in the last form, iterators can be used instead of pointers). The meaning of arguments is similar as in **insert**. Methods **replace** return a reference to *this* string. For example

```
string s("0123456789");
const char* p("abcdef");
s.replace(0,2,p,2).replace(s.end()-2,s.end(),p+4,p+6);
cout << s << endl;
```

will print 'ab234567ef'.

size_type find(const string* s, size_type start = 0)
size_type find(const char* p, size_type start = 0)
size_type find(const char* p, size_type start, size_type cnt)
size_type find(char c, size_type start = 0)
size_type rfind(const string* s, size_type start = npos)
size_type rfind(const char* p, size_type start = npos)
size_type rfind(const char* p, size_type start, size_type cnt)

size_type rfind(char c, size_type start = npos) — find, starting from position ***start*** in *this* string, the first occurence of a sequence specified by the remaining arguments. Functions from **rfind** family act in a similar way, but searching proceeds backwards. The position (index) of the beginning of the substring found is returned, or ***npos*** if the search failed. In the example below, **find** scans the string 'abc345abcAB', starting from the position 3 (i.e., digit '3'), looking for the first occurence of a sequence composed of first two characters of C-string ***p*** (i.e., 'ab'):

```
string s("abc345abcAB");
const char* p("abcdef");
string::size_type i = s.find(p,3,2);
cout << s.substr(i-1,5) << endl;
```

The fragment would print '5abcA'.

size_type find_first_of(/* args */)
size_type find_last_of(/* args */)
size_type find_first_not_of(/* args */)
size_type find_last_not_of(/* args */) — have types of return value and arguments as the methods from **find**/ **rfind** family. First two methods look, starting from position ***start***, for the first occurence of *any* character belonging to character sequence specified by the remaining arguments. The direction of searching is forward for methods having **_first_** in their name and backwards for those with **_last_**. The methods with **_not_** in their names act in the same way, but look for any character *not* belonging to character sequence specified by the remaining arguments. In all cases the position of the character found is returned, or ***npos*** if the search failed. For example

```
1.  string s("abc123.,!");
2.  const char* p = "!.,?:1234";
3.  string::size_type i = s.find_first_of(p);
4.  string::size_type k = s.find_last_not_of(p,s.size()-1,5);
5.  string s1(s.begin()+i,s.begin()+k+1);
6.  cout << i << " " << k << " " << s1 << endl;
```

will print '3 5 123'. On line 4 we set the second argument to `s.size()-1`, as searching will be backwards, so we have to start from the last character. We take into account only five first characters of the pattern ***p***, i.e., characters '`!.,?:`'. on line 5 we create a new object of class **string** initialized with a slice of string ***s***. We add 1 in the second argument (`s.begin()+k`), as we want the (`s.begin()+k`)-th character to be included.

int compare(const string& pat)
int compare(size_type start, size_type cnt, const string& pat)
int compare(size_type start, size_type cnt, const string& pat, size_type s, size_type cnt1)
int compare(const char* p)
int compare(size_type start, size_type cnt, const char* p, size_type i = npos) — compares *this* string, or its substring specified by ***start*** and ***cnt***, with a sequence of characters defined by the remaining arguments. Returns `-1`, if *this* string is lexicographically earlier, zero if it is equal, or `+1` if it lexicographically follows the sequence.

void push_back(char c) — appends character ***c*** to *this* string. Invocation `s.push_back(c)` acts as `s.insert(s.end(),c)` but does not return any value (**insert** would return an iterator). Same effect can be obtained by invoking **append** or '`+=`' operator.

const char* c_str( ) — returns a pointer to a constant C-string with the same contents as

this string (with terminating ' \0'). The C-string returned cannot be modified.

iterator begin( ) — returns an iterator (see section on iterators) pointing to the first character of the string.

iterator end( ) — returns an iterator pointing to the first after the last character of the string.

reverse_iterator rbegin( )
reverse_iterator rend( ) — as **begin** and **end** but returns reverse iterator; e.g., after

```
string s1("ABCD");
string s2(s1.rbegin(),s1.rend());
```

s2 will be 'DCBA'.

After including the header file `string`, we also have a very useful function **getline** (which is *not* a method of class **string**):

istream& getline(istream& str, string& s)
istream& getline(istream& str, string& s, char eol) — reads one line of text from input stream ***str*** into string ***s***. The line *can* contain white characters other than end-of-line character. The end-of-line character itself is extracted from the stream but is *not* put into ***s***. If a character is given as the third argument, it will play the rôle of the end-of-line character. The method returns a reference to the stream ***str***, so a code like

```
string s1,s2;
getline(cin,s1) >> s2;
```

will work and will put the first line into ***s1*** and the first word of the next line into ***s2***.

---

18. Templates

The mechanism of templates is very characteristic for the C++ language; there is no such mechanism in majority of other object-oriented languages. Templates make it possible to define functions and classes in an abstract way in terms of parameters, which are *types* of data. The compiler can produce, on the basis of defined templates, various forms of functions and classes when they are needed. This mechanism is a corner stone of the standard library, which actually consists mainly of templates — that is why it is called STL: **Standard Template Library**.

The general idea for function and class templates is the same; the details, however, depend on whether we deal with functions or classes. Function templates have already been (rather briefly) described in chapter on function templates ; here we will say a few words about tempales of classes. Of course, this will only be a short introduction — the subject is extremely rich, so the reader will have to consult dedicated text-books on C++ Standard Library for details.

Subsections

- 18.1 Templates of classes

18.1 Templates of classes

In a similar way, one can create templates of whole *classes* — together with constructors, destructors, methods and fields. The syntax is similar:

```
template <typename T, typename M>
class AClass {
    // here we use types T and M
};
```

defines a class template **AClass** parametrized with two types. How can one create objects of class concretized from this template? We cannot just use

```
AClass x;
```

because there would be no way for the compiler to figure out what types should be substituted for **T** and **M** in the template. Hence, we have to specify it explicitly: we do it by writing names of types (built-in, form a library or our own) which are required for this concretization in angle brackets, exactly as we could do it for functions. For functions, however, it is rarely needed, because the compiler can deduce appropriate types looking at the types of arguments of a given call; for classes we always have to do it manually, e.g.:

```
AClass<double,int> x;
```

The variable ***x*** will now be an object of class resulting from concretization of the template **AClass** by substituting **double** for all occurences of **T** and **int** for all occurences of **M**. The name of the class created in this way will be **AClass<double,int>**. We can refer to the class by this name, e.g., to create more objects; no concretization will then be performed, because the class has already been created and compiled. However, using different types:

```
AClass<int,Person> z;
```

we will force the compiler to generate another concretization of the same template: class **AClass<int,Person>** will now be created and it will *not* have anything in common with the previously generated class **AClass<double,int>** (except the fact that they both were created on the basis of the same template).

Parameter of the template can also refer to a value of a given type. For example:

```
template <typename T, int size>
class AClass {
    // we use type 'T' and value
    // 'size' in the definition
};
```

We can concretize this template defining an object

```
AClass<Person,100> t;
```

The class generated will have the name **AClass<Person,100>**. Note that passing a different value as the second template argument (e.g., 150 instead of 100), we would get a different, completely independent class.

It poses some problems for beginners to define a method outside a class defined as a template. As we remember, when defining such a method, one has to use its full qualified name, including the name of the class that the method comes from. In case of a template, one has to use the name of the template with, in angle brackets, the names of its parameters, but without the keyword **class** or **typename**:

```
    template <typename T, int size>
    class AClass {
        void method1() {
            // definition of method1
        }
        T* method2(double); // declaration only

        // ...
    };

    //
    // ...
    //

    // definition of method2
    template <typename T, int size>
    T* AClass<T,size>::method2(double x) {
        // body of the definition
    }
```

A method **method1** is defined directly inside the class template, but **method2** is only declared there and defined somewhere else. Analogously, one can define, outside the template, constructors, destructors etc.

Let us consider an implementation of stack as a template (in order to be able to use stacks of various types). As before, we have to implement methods for

- adding an element on the top (**push**);
- removing and returning the element from the top (**pop**);
- checking if the stack is empty (**empty**).

Concretizations of template **Stack** will differ only with types of elements and sizes of the allocated array:

P148: stackt.cpp Class template of stack with array implemetation

```
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

template <typename Data, int size>
class Stack {
    Data* data;
    int   top;
public:
    Stack();
    bool empty() const;
    void push(Data);
    Data pop();
    ~Stack();
};

template <typename Data, int size>
Stack<Data,size>::Stack() {
    data = new Data[size];
    top = 0;
```

```
22. }
23.
24. template <typename Data, int size>
25. inline bool Stack<Data,size>::empty() const {
26.     return top == 0;
27. }
28.
29. template <typename Data, int size>
30. inline void Stack<Data,size>::push(Data dat) {
31.     data[top++] = dat;
32. }
33.
34. template <typename Data, int size>
35. inline Data Stack<Data,size>::pop() {
36.     return data[--top];
37. }
38.
39. template <typename Data, int size>
40. inline Stack<Data,size>::~Stack() {
41.     delete [] data;
42. }
43.
44.   // template of a global function
45. template <typename Data, int size>
46. void clear(Stack<Data,size>* p_stos) {
47.     cout << "Stack of type " << typeid(Data).name() << ": ";
48.     while ( ! p_stos->empty() ) {
49.         cout << p_stos->pop() << " ";
50.     }
51.     cout << endl;
52. }
53.
54. int main() {
55.     Stack<int,20> stack_i;
56.     stack_i.push(11);
57.     stack_i.push(36);
58.     stack_i.push(49);
59.     stack_i.push(92);
60.
61.     Stack<string,15> stack_s;
62.     stack_s.push("Alice");
63.     stack_s.push("Emmy");
64.     stack_s.push("Winnie");
65.     stack_s.push("Una");
66.
67.     clear(&stack_i);
68.     clear(&stack_s);
69. }
```

The parameters of the template **Stos** are **Data** (representing type of elements) and integer number ***size*** representing the size of the stack. For clarity, error handling is omitted here.

Constructor, destructor and all the methods are declared inside the class template, but defined later (lines 18-42). They are all so simple that the compiler will probably be able to inline them — therefore, we add the keyword **inline** to their definition (see chapter on inline functions).

We also define a template of global functions **clear** (lines 44-52), which pop and print all elements from a stack. Their argument is of type *pointer to a stack*, where the stack is of a type concretized from the template **Stack**. Again, the function prints the actual type associated with type **Data** (using **typeid** operator).

Function **main** creates two stacks: a stack of integers, ***stack_i***, with maximum size equal to 20, and a stack of strings, ***stack_s***, with size 15. We push a few elements on both stacks and then we clear them using function **clear** (lines 67 and 68). We get:

```
Stack of type i: 92 49 36 11
Stack of type Ss: Una Winnie Emmy Alice
```

As we already know, internal names of types (in our example **i** for **int** and **Ss** for **string**) can depend on the compiler.

Note that calling the function **clear**, we did not specify any types as arguments for the template, although we could have done it by writing explicitely

```
clear<int,20>(&stack_i);
clear<string,15>(&stack_s);
```

As we can see, the compiler did not need this hint to deduce the appropriate type for concretization of template **clear** — the information was provided by the type of arguments.

Note again that concretization of template **Stack** depends on both type substituted for **Data** and value substituted for ***size***. Therefore, classes **Stack<int,10>** and **Stack<int,11>** would be two unrelated data types.

Prev: 18. Templates **Up:** 18. Templates **Next:** 19. Operator overloading. Move semantics

19. Operator overloading. Move semantics and smart pointers

By **operator overloading** we mean a mechanism which allows the programmer to use operators (like '+', '&') with data of his/her own types (classes). Operator overloading can usually be easily replaced by defining “normal” functions or methods, but is still used because can make the code more natural and easier to understand (if it is implemented in a right way...). There are many rich and flexible languages where operator overloading cannot be used (as C or Java) and others where it is used more or less similarly as in C++ (like Python, Haskell or C#).

At the end of this chapter, we will also talk about move sematics, which is somewhat related to the operator overloading.

Subsections

- 19.1 Introduction
- 19.2 Overloading with global functions
  - 19.2.1 Binary operators
  - 19.2.2 Unary operators
- 19.3 Overloading with methods of classes
  - 19.3.1 Binary operators
  - 19.3.2 Unary operators
- 19.4 “Special” operators
  - 19.4.1 Assignment operator
  - 19.4.2 Indexing operator
  - 19.4.3 Function call operator
  - 19.4.4 Overloading indirect member selection operator
- 19.5 Move semantics
- 19.6 Smart pointers
  - 19.6.1 Smart pointers of the *unique_ptr* type
  - 19.6.2 Smart pointers of the *shared_ptr* type

19.1 Introduction

Using operators (like ' +', ' ->', ' &&' etc.) in a program is in fact another way of an invocation of a function and passing operands as arguments to this function. For example, the statement

```
c = a + b;
```

will invoke function which will add ***a*** and ***b***. Which function it will be depends on type of arguments; if both are of type **double**, then a function which is able to add two **double** s will be used, if both are **int** s, another function will be selected by the compiler. If one is **double** and the other is **int**, then the **int** value will be converted to type **double** and the function adding **double** s will be used, yielding a result also of type **double**. Everything will go smoothly for built-in types, as for them adding operation has already been defined.

There will be a problem, however, when one or both operands are of a user-defined type. Then the compiler cannot know what adding really means. Therefore, if we still want to use this notation, we have to define appropriate functions to be called ourselves. We can do it by defining (in a special way) methods of our class or global functions usually, although not necessarily, declared as friends of our class.

This is called **operator overloading** in analogy with function overloading, i.e., defining several functions which are visible in the same scope and have the same name. As for functions, a method (function) which should be invoked is selected by the compiler on the basis of types of arguments.

The following operators can be overloaded:

+	-	*	/	%	^	&	|	~	!	
=	<	>	+=	-=	*=	/=	%=	^=	&=	
|=	<<	>>	<<=	>>=	==	!=	<=	>=	&&	
||	++	--	,	->*	->	new	delete	()	[]	

Table 19.1: Operators which can be overloaded

where operatos ' &', ' *', ' -', ' +' can be overloaded both as one- and two-argument operators. Operators '++' and '- -' can be overloaded for prefix and postfix forms separately.

Operators ' =', ' ->',' ()' and ' []' are somewhat special, as we will see in the following. Overloading ' **new**' and ' **delete**' operators is a rather delicate problem, often implementation dependent, so we will not consider them.

Operators ' .', ' .*', ' ::', ' ?:' and **sizeof** simply *cannot* be overloaded.

Some operators are defined automatically for all, even user-defined, types (with some exceptions which we will be talking about later). These are: ' &' (address), ' =' (assignment),

' , ' (comma), **new** and **delete** operators. It is better not to touch the first; the last three usually do not need any redefinition.

Overloading operators does not modify their precedence and associativity (left or right, see chapter on operators) In the statement

```
a = b + c * d;
```

the operation assigned to ' *' will precede the execution of ' +' no matter how, in this context, these two operators are (re)defined. Similarly,

```
a = b = c;
```

will always be equivalent to

```
a = ( b = c );
```

and not to

```
( a = b ) = c;
```

as the assignment operator has right associativity.

The number of arguments of an overloaded operator is not arbitrary either: unary operators have to remain unary and binary operators must be binary even in their overloaded version.

Overloading operators one has to remember about the pronciple *of least astonishment*; if we overload an operator, its behavior and meaning should be easy to guess and as natural as possible. For example, overloaded ' + operator should return the result (possibly, of the same type as arguments) by value, so it is *not* an l-value, because this is how “normal” adding behaves like.

Similarly, having overloaded ' +', we should consider overloading also ' +=', perhaps subtracting as well or decrementation and incrementation operators. On the other hand, implementing something like **String** or **StringBuffer** classes from Java, it is natural that addition should correspond to concatenation, but there is no intuitively obvious interpretation of subtracting.

Generally, operators can be overloaded by defining a method of a class or, with some exceptions, a global function: the compiler will know which methods or functions are used for operator overloading by their special names.

Prev: 19. Operator overloading. Move semantics **Up:** 19. Operator overloading. Move semantics **Next:** 19.2 Overloading with global functions

Subsections

- 19.2.1 Binary operators
- 19.2.2 Unary operators

19.2 Overloading with global functions

We will consider unary and binary operators separately.

19.2.1 Binary operators

A global function overloading a *binary* operator is defined as a function with two arguments, of which at least one has to be of a user-defined type. The name of the function must be of the form '`operator`', where '' should be substituted by the symbol of an operator listed in table of operators and corresponding to a binary operator (as, e.g., ' `+`' or '`<<`'). Note that the name is special: the word **operator** is a reserved keyword; moreover, characters like ' `+`' or '`<<`' cannot normally be used in identifiers.

Therefore, prototype of such a function would be

```
Type operator@(Type1, Type2);
```

where at leat one of parameter types should correspond to a user defined class (or structure). The type of the returned value is arbitrary (can also be **void**). Of course, instead of ' `@`' one should use the symbol of an operator being overloaded. The function can be a **friend** of our class if we want it to have direct access to **private** and **protected** members of the class; this is, however, not always necessary.

A function defined in this way can be invoked directly “by name” (which is ' **operator@**'), but usually we do not use this form; after all, we overload operators to be able to use shorthand notation which employs just symbols of operators, not their full names. The point is that the compiler will substitute invocation of an appropriate function when an expression like

```
a @ b
```

is encounterd and the symbol of operator and types of operands match those used in declaration/definition of overloading. Therefore, the expression above will be equivalent to

```
operator@(a,b)
```

In the example below, we define a simple class **Modulo**:

P149: modsev.cpp Overloading the addition operator

```
1. #include <iostream>
```

```
 2. using namespace std;
 3.
 4. struct Modulo {
 5.     static const int modul;
 6.
 7.     int numb;
 8.
 9.     Modulo() : numb(0)
10.     { }
11.
12.     Modulo(int numb) : numb(numb%modul)
13.     { }
14. };
15. const int Modulo::modul = 7;
16.
17. Modulo operator+(Modulo m, Modulo n) {
18.     return Modulo(m.numb + n.numb);
19. }
20.
21. int main() {
22.     Modulo m(5), n(6), k;
23.
24.     k = m + n;
25.     cout << m.numb << " + " << n.numb
26.          << " (mod " << Modulo::modul
27.          << ") = " << k.numb << endl;
28.
29.     k = operator+(m,n);
30.     cout << m.numb << " + " << n.numb
31.          << " (mod " << Modulo::modul
32.          << ") = " << k.numb << endl;
33. }
```

Objects of the class represent integer numbers *modulo* 7 (any number is represented by the remainder it gives when divided by 7; e.g., 11 and 32 are both represented by 4). There are only seven distinct values of **Modulo** — $[0,..., 6]$. When two such numbers are added, the result should take one of the seven possible values. For example

$$5+6 \equiv 4 \pmod 7$$

We redefine then the operation of adding numbers of type **Modulo**. On lines 17-19 we define a function named `operator+`, with two parameters of type **Modulo**. This function will be called whenever a ' +' symbol is encountered with expressions of type **Modulo** as operands on both sides of the operator. This will happen on line 24: our function will be invoked and will return an object of type **Modulo** representing the sum of the arguments (or whatever we put into the function's definition).

Line 29 shows that, if we insist, the function can be called explicitly “by its name”; the result will be identical:

```
5 + 6 (mod 7) = 4
5 + 6 (mod 7) = 4
```

In this example overloading subtraction, incrementation or multiplication would be quite natural; we do not do it to keep the example short.

Note that the type of the result is **Modulo**; it is returned by value and hence is *not* an l-value (in accordance to what one would expect for numbers).

The '<<' (stream insertion) operator belongs to the most often overloaded operators. Overloading this operator makes it possible to output string representation of objects of user-defined types in the same way as for built-in types. What should be the type of

parameters for the corresponding function? When '<<' operator is used, we have a stream object (e.g., ***cout***) on the left hand side and "something" to be printed on the right hand side. This "something" can be of type **int**, **double** etc.; compiler will know what to do in all these cases because its behavior in such situations has already been programmed. There will be a problem, however, if this "something" on the right hand side is of a user-defined type. To handle this situation, we have to define an overloading of '`operator<<`' function with parameters of types **ostream&** (left operand) and **AClass** or **AClass&** (right opearand), where **AClass** is the name of our class. What about the return type? If all we need is something like

```
cout << zzz;
```

where ***zzz*** is the identifier of an object of our class, then the return type would basically be irrelevant. However, if we want '<<' to work sequentially, in a cascade,

```
cout << zzz << " " << yyy;
```

then the value of '`cout << zzz`' should be again ***cout***, so it becomes the left operand of the next stream insertion operator. Hence, the prototype of the overloading function should be

```
ostream& operator<<(ostream&, const AClass&);
```

The first parameter, of type **ostream&**, cannot be **const**, because insertion to a stream *can* modify an object representing it. It cannot be **ostream** either, because passing the argument by value would require copying it, but the copy constructor in class **ostream** is **private**; for the same reason the result has to be returned by reference and not by value.

An object of our class **AClass** (second parameter) can be passed either by value or by reference. In the former case, we have to consider if the class has been equipped in a correct copy constructor and if the process of copying will not deteriorate efficiency too much. In the second case, we have to remember that the function will work on the original object — it is therefore reasonable — although not obligatory — to declare this parameter as **const**: after all, the function's task is to display some information, not to modify any objects.

Functions overloading stream insertion operator often need access to private members of the class they are defined to work with; in such cases one can declare them as friends of the class:

```
class AClass {
    // ...
    friend ostream& operator<<(ostream&, const AClass&);
}
```

The function should always return a reference to exactly the same stream object which has been passed as the first argument — of course, it does not have to be ***cout***; it can be any object of class **ostream** or a class derived from it.

Let us consider an example:

P150: strinsop.cpp Overloading stream insertion operator

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
```

```
 4.
 5. class Person {
 6.     string name;
 7.     int    age;
 8. public:
 9.     Person(const string& name, int age)
10.         : name(name), age(age)
11.     { }
12.
13.      // ... other members
14.
15.     friend ostream& operator<<(ostream&, const Person&);
16. };
17.
18. ostream& operator<<(ostream& str, const Person& k) {
19.     return str << k.name << " (" << k.age << " yo)";
20. }
21.
22. int main() {
23.     Person t[] = {  Person("Joe",18), Person("Sue",26),
24.                     Person("Eve",35), Person("Tim",11)  };
25.
26.     for (int i = 0; i < 4; i++)
27.         cout << t[i] << endl;
28. }
```

On line 15, inside the definition of class **Person**, we declare that the function overloading '<<' operator for objects of the class will be its friend; in this way the function, although not a member of the class, can nevertheless directly access private members ***name*** and ***age***, as we can see from the program's output

```
Joe (18 yo)
Sue (26 yo)
Eve (35 yo)
Tim (11 yo)
```

The overloading function itself is defined On lines 18-20 according to the rules we have described above. It is a global function so it is not called for an object: the object which is meant here is the right operand of '<<' operator and is passed to the function as the second argument by reference (there is no **this** pointer in the function). Note that the value of the whole expression appearing in the **return** statement is the stream object ***str***, which is exactly what the function should return — due to this, the body of the function could have been reduced to one **return** statement.

An overloaded binary operator can have *both* parameters of, possibly different, user-defined types. In the example below ' +' operator is overloaded twice: one version adds vectors (objects of class **Vector**) to vectors with the result being another vector, the other adds vectors to points giving points. The compiler will be able to select the correct version on the basis of types of arguments (lines 50 and 53):

P151: vecpoin.cpp Overloading binary operators

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. class Point;
 5.
 6. class Vector {
 7.     double x, y, z;
 8. public:
```

```
9.      Vector(double x = 0, double y = 0, double z = 0)
10.         : x(x), y(y), z(z)
11.     { }
12.     friend Point    operator+(const Point&,  const Vector&);
13.     friend Vector   operator+(const Vector&, const Vector&);
14.     friend ostream& operator<<(ostream&, const Vector&);
15. };
16.
17. class Point {
18.     double x, y, z;
19. public:
20.     Point(double x = 0, double y = 0, double z = 0)
21.         : x(x), y(y), z(z)
22.     { }
23.     friend Point operator+(const Point&, const Vector&);
24.     friend ostream& operator<<(ostream&, const Point&);
25. };
26.
27. Point operator+(const Point&  p, const Vector& v) {
28.     return Point(p.x+v.x, p.y+v.y, p.z+v.z);
29. }
30.
31. Vector operator+(const Vector& v1, const Vector& v2) {
32.     return Vector(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
33. }
34.
35. ostream& operator<<(ostream& str, const Point& p) {
36.     return str << "P(" << p.x << "," << p.y
37.                << ","  << p.z << ")";
38. }
39.
40. ostream& operator<<(ostream& str, const Vector& v) {
41.     return str << "V[" << v.x << "," << v.y
42.                << ","  << v.z << "]";
43. }
44.
45. int main() {
46.
47.     Vector v1(1,1,1), v2(2,2,2);
48.     Point  p1(1,2,3);
49.
50.     Vector v = v1 + v2;
51.     cout << "v: " << v << endl;
52.
53.     Point  p = p1 + v;
54.     cout << "p: " << p << endl;
55. }
```

Functions overloading addition and stream insertion use private members of the classes directly, therefore they had to be declared as **friend** s in the definitions of the classes. Note that forward declaration on line 4 is necessary here: we use the name **Point** in the definition of **Vector** and *vice versa*. Note also that both addition functions return result by value so it is *not* an l-value: that is how addition normally behaves.

19.2.2 Unary operators

Unary (one-argument) operators can be overloaded by global one-parameter functions with prototype of the form

```
Type operator@(Typearg);
```

Type **Typearg** must be a user-defined type, type **Type** is arbitrary. As before, symbol '' stands here for the symbol of any unary operator (like '&' or '!'). When such a function is defined, and ***a*** denotes an object (or reference to an object) of a user-defined type, expression

```
@a
```

is equivalent to invocation

```
operator@(a)
```

In the example below, we overload unary operator ' ! ' for objects of class **AClass**. This class has a field of type **string**, and the overloading function defined on lines 13-15 returns values of type **bool** which answer the question whether the string has more than 5 characters:

P152: oneargop.cpp Overloading unary operators

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  struct AClass {
6.      string name;
7.
8.      AClass(const string& name)
9.          : name(name)
10.     { }
11. };
12.
13. bool operator!(const AClass& c) {
14.     return c.name.size() > 5;
15. }
16.
17. int main() {
18.     AClass t[] = { AClass("Marlon"), AClass("Henry"),
19.                    AClass("Dave"),   AClass("Horatio"),
20.                    AClass("Sue"),    AClass("Alice")   };
21.
22.     for (int i = 0; i < 6; ++i)
23.         if ( !t[i] ) cout << t[i].name << endl;
24. }
```

On line 23 we print names corresponding to obejcts from array ***t***; the use of ' ! ' operator causes only long names to be printed:

```
Marlon
Horatio
```

Prev: 19.1 Introduction **Up:** 19. Operator overloading. Move semantics **Next:** 19.3 Overloading with methods of

Subsections

- 19.2.1 Binary operators
- 19.2.2 Unary operators

19.2 Overloading with global functions

We will consider unary and binary operators separately.

19.2.1 Binary operators

A global function overloading a *binary* operator is defined as a function with two arguments, of which at least one has to be of a user-defined type. The name of the function must be of the form '`operator`', where '' should be substituted by the symbol of an operator listed in table of operators and corresponding to a binary operator (as, e.g., ' `+`' or '`<<`'). Note that the name is special: the word **operator** is a reserved keyword; moreover, characters like ' `+`' or '`<<`' cannot normally be used in identifiers.

Therefore, prototype of such a function would be

```
Type operator@(Type1, Type2);
```

where at leat one of parameter types should correspond to a user defined class (or structure). The type of the returned value is arbitrary (can also be **void**). Of course, instead of ' `@`' one should use the symbol of an operator being overloaded. The function can be a **friend** of our class if we want it to have direct access to **private** and **protected** members of the class; this is, however, not always necessary.

A function defined in this way can be invoked directly “by name” (which is ' **operator@**'), but usually we do not use this form; after all, we overload operators to be able to use shorthand notation which employs just symbols of operators, not their full names. The point is that the compiler will substitute invocation of an appropriate function when an expression like

```
a @ b
```

is encounterd and the symbol of operator and types of operands match those used in declaration/definition of overloading. Therefore, the expression above will be equivalent to

```
operator@(a,b)
```

In the example below, we define a simple class **Modulo**:

P149: modsev.cpp Overloading the addition operator

```
1. #include <iostream>
```

```
 2. using namespace std;
 3.
 4. struct Modulo {
 5.     static const int modul;
 6.
 7.     int numb;
 8.
 9.     Modulo() : numb(0)
10.     { }
11.
12.     Modulo(int numb) : numb(numb%modul)
13.     { }
14. };
15. const int Modulo::modul = 7;
16.
17. Modulo operator+(Modulo m, Modulo n) {
18.     return Modulo(m.numb + n.numb);
19. }
20.
21. int main() {
22.     Modulo m(5), n(6), k;
23.
24.     k = m + n;
25.     cout << m.numb << " + " << n.numb
26.          << " (mod " << Modulo::modul
27.          << ") = " << k.numb << endl;
28.
29.     k = operator+(m,n);
30.     cout << m.numb << " + " << n.numb
31.          << " (mod " << Modulo::modul
32.          << ") = " << k.numb << endl;
33. }
```

Objects of the class represent integer numbers *modulo* 7 (any number is represented by the remainder it gives when divided by 7; e.g., 11 and 32 are both represented by 4). There are only seven distinct values of **Modulo** — $[0,..., 6]$. When two such numbers are added, the result should take one of the seven possible values. For example

$$5+6 \equiv 4 \pmod 7$$

We redefine then the operation of adding numbers of type **Modulo**. On lines 17-19 we define a function named `operator+`, with two parameters of type **Modulo**. This function will be called whenever a ' +' symbol is encountered with expressions of type **Modulo** as operands on both sides of the operator. This will happen on line 24: our function will be invoked and will return an object of type **Modulo** representing the sum of the arguments (or whatever we put into the function's definition).

Line 29 shows that, if we insist, the function can be called explicitly “by its name”; the result will be identical:

```
5 + 6 (mod 7) = 4
5 + 6 (mod 7) = 4
```

In this example overloading subtraction, incrementation or multiplication would be quite natural; we do not do it to keep the example short.

Note that the type of the result is **Modulo**; it is returned by value and hence is *not* an l-value (in accordance to what one would expect for numbers).

The '<<' (stream insertion) operator belongs to the most often overloaded operators. Overloading this operator makes it possible to output string representation of objects of user-defined types in the same way as for built-in types. What should be the type of

parameters for the corresponding function? When '<<' operator is used, we have a stream object (e.g., ***cout***) on the left hand side and "something" to be printed on the right hand side. This "something" can be of type **int**, **double** etc.; compiler will know what to do in all these cases because its behavior in such situations has already been programmed. There will be a problem, however, if this "something" on the right hand side is of a user-defined type. To handle this situation, we have to define an overloading of '`operator<<`' function with parameters of types **ostream&** (left operand) and **AClass** or **AClass&** (right opearand), where **AClass** is the name of our class. What about the return type? If all we need is something like

```
cout << zzz;
```

where ***zzz*** is the identifier of an object of our class, then the return type would basically be irrelevant. However, if we want '<<' to work sequentially, in a cascade,

```
cout << zzz << " " << yyy;
```

then the value of '`cout << zzz`' should be again ***cout***, so it becomes the left operand of the next stream insertion operator. Hence, the prototype of the overloading function should be

```
ostream& operator<<(ostream&, const AClass&);
```

The first parameter, of type **ostream&**, cannot be **const**, because insertion to a stream *can* modify an object representing it. It cannot be **ostream** either, because passing the argument by value would require copying it, but the copy constructor in class **ostream** is **private**; for the same reason the result has to be returned by reference and not by value.

An object of our class **AClass** (second parameter) can be passed either by value or by reference. In the former case, we have to consider if the class has been equipped in a correct copy constructor and if the process of copying will not deteriorate efficiency too much. In the second case, we have to remember that the function will work on the original object — it is therefore reasonable — although not obligatory — to declare this parameter as **const**: after all, the function's task is to display some information, not to modify any objects.

Functions overloading stream insertion operator often need access to private members of the class they are defined to work with; in such cases one can declare them as friends of the class:

```
class AClass {
    // ...
    friend ostream& operator<<(ostream&, const AClass&);
}
```

The function should always return a reference to exactly the same stream object which has been passed as the first argument — of course, it does not have to be ***cout***; it can be any object of class **ostream** or a class derived from it.

Let us consider an example:

P150: strinsop.cpp Overloading stream insertion operator

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
```

```
 4.
 5. class Person {
 6.     string name;
 7.     int    age;
 8. public:
 9.     Person(const string& name, int age)
10.         : name(name), age(age)
11.     { }
12.
13.      // ... other members
14.
15.     friend ostream& operator<<(ostream&, const Person&);
16. };
17.
18. ostream& operator<<(ostream& str, const Person& k) {
19.     return str << k.name << " (" << k.age << " yo)";
20. }
21.
22. int main() {
23.     Person t[] = {  Person("Joe",18), Person("Sue",26),
24.                     Person("Eve",35), Person("Tim",11)  };
25.
26.     for (int i = 0; i < 4; i++)
27.         cout << t[i] << endl;
28. }
```

On line 15, inside the definition of class **Person**, we declare that the function overloading '<<' operator for objects of the class will be its friend; in this way the function, although not a member of the class, can nevertheless directly access private members ***name*** and ***age***, as we can see from the program's output

```
Joe (18 yo)
Sue (26 yo)
Eve (35 yo)
Tim (11 yo)
```

The overloading function itself is defined On lines 18-20 according to the rules we have described above. It is a global function so it is not called for an object: the object which is meant here is the right operand of '<<' operator and is passed to the function as the second argument by reference (there is no **this** pointer in the function). Note that the value of the whole expression appearing in the **return** statement is the stream object ***str***, which is exactly what the function should return — due to this, the body of the function could have been reduced to one **return** statement.

An overloaded binary operator can have *both* parameters of, possibly different, user-defined types. In the example below ' +' operator is overloaded twice: one version adds vectors (objects of class **Vector**) to vectors with the result being another vector, the other adds vectors to points giving points. The compiler will be able to select the correct version on the basis of types of arguments (lines 50 and 53):

P151: vecpoin.cpp Overloading binary operators

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. class Point;
 5.
 6. class Vector {
 7.     double x, y, z;
 8. public:
```

```
9.      Vector(double x = 0, double y = 0, double z = 0)
10.         : x(x), y(y), z(z)
11.     { }
12.     friend Point    operator+(const Point&,  const Vector&);
13.     friend Vector   operator+(const Vector&, const Vector&);
14.     friend ostream& operator<<(ostream&, const Vector&);
15. };
16.
17. class Point {
18.     double x, y, z;
19. public:
20.     Point(double x = 0, double y = 0, double z = 0)
21.         : x(x), y(y), z(z)
22.     { }
23.     friend Point operator+(const Point&, const Vector&);
24.     friend ostream& operator<<(ostream&, const Point&);
25. };
26.
27. Point operator+(const Point&  p, const Vector& v) {
28.     return Point(p.x+v.x, p.y+v.y, p.z+v.z);
29. }
30.
31. Vector operator+(const Vector& v1, const Vector& v2) {
32.     return Vector(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
33. }
34.
35. ostream& operator<<(ostream& str, const Point& p) {
36.     return str << "P(" << p.x << "," << p.y
37.                << ","  << p.z << ")";
38. }
39.
40. ostream& operator<<(ostream& str, const Vector& v) {
41.     return str << "V[" << v.x << "," << v.y
42.                << ","  << v.z << "]";
43. }
44.
45. int main() {
46.
47.     Vector v1(1,1,1), v2(2,2,2);
48.     Point  p1(1,2,3);
49.
50.     Vector v = v1 + v2;
51.     cout << "v: " << v << endl;
52.
53.     Point  p = p1 + v;
54.     cout << "p: " << p << endl;
55. }
```

Functions overloading addition and stream insertion use private members of the classes directly, therefore they had to be declared as **friend** s in the definitions of the classes. Note that forward declaration on line 4 is necessary here: we use the name **Point** in the definition of **Vector** and *vice versa*. Note also that both addition functions return result by value so it is *not* an l-value: that is how addition normally behaves.

19.2.2 Unary operators

Unary (one-argument) operators can be overloaded by global one-parameter functions with prototype of the form

```
Type operator@(Typearg);
```

Type **Typearg** must be a user-defined type, type **Type** is arbitrary. As before, symbol '' stands here for the symbol of any unary operator (like '&' or '!'). When such a function is defined, and ***a*** denotes an object (or reference to an object) of a user-defined type, expression

```
@a
```

is equivalent to invocation

```
operator@(a)
```

In the example below, we overload unary operator ' ! ' for objects of class **AClass**. This class has a field of type **string**, and the overloading function defined on lines 13-15 returns values of type **bool** which answer the question whether the string has more than 5 characters:

P152: oneargop.cpp Overloading unary operators

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  struct AClass {
6.      string name;
7.
8.      AClass(const string& name)
9.          : name(name)
10.     { }
11. };
12.
13. bool operator!(const AClass& c) {
14.     return c.name.size() > 5;
15. }
16.
17. int main() {
18.     AClass t[] = { AClass("Marlon"), AClass("Henry"),
19.                    AClass("Dave"),   AClass("Horatio"),
20.                    AClass("Sue"),    AClass("Alice")   };
21.
22.     for (int i = 0; i < 6; ++i)
23.         if ( !t[i] ) cout << t[i].name << endl;
24. }
```

On line 23 we print names corresponding to obejcts from array ***t***; the use of ' ! ' operator causes only long names to be printed:

```
Marlon
Horatio
```

Prev: 19.1 Introduction **Up:** 19. Operator overloading. Move semantics **Next:** 19.3 Overloading with methods of

Subsections

- 19.2.1 Binary operators
- 19.2.2 Unary operators

19.2 Overloading with global functions

We will consider unary and binary operators separately.

19.2.1 Binary operators

A global function overloading a *binary* operator is defined as a function with two arguments, of which at least one has to be of a user-defined type. The name of the function must be of the form '`operator`', where '' should be substituted by the symbol of an operator listed in table of operators and corresponding to a binary operator (as, e.g., ' `+`' or '`<<`'). Note that the name is special: the word **operator** is a reserved keyword; moreover, characters like ' `+`' or '`<<`' cannot normally be used in identifiers.

Therefore, prototype of such a function would be

```
Type operator@(Type1, Type2);
```

where at leat one of parameter types should correspond to a user defined class (or structure). The type of the returned value is arbitrary (can also be **void**). Of course, instead of ' `@`' one should use the symbol of an operator being overloaded. The function can be a **friend** of our class if we want it to have direct access to **private** and **protected** members of the class; this is, however, not always necessary.

A function defined in this way can be invoked directly “by name” (which is ' **operator@**'), but usually we do not use this form; after all, we overload operators to be able to use shorthand notation which employs just symbols of operators, not their full names. The point is that the compiler will substitute invocation of an appropriate function when an expression like

```
a @ b
```

is encounterd and the symbol of operator and types of operands match those used in declaration/definition of overloading. Therefore, the expression above will be equivalent to

```
operator@(a,b)
```

In the example below, we define a simple class **Modulo**:

P149: modsev.cpp Overloading the addition operator

```
1. #include <iostream>
```

```
 2. using namespace std;
 3.
 4. struct Modulo {
 5.     static const int modul;
 6.
 7.     int numb;
 8.
 9.     Modulo() : numb(0)
10.     { }
11.
12.     Modulo(int numb) : numb(numb%modul)
13.     { }
14. };
15. const int Modulo::modul = 7;
16.
17. Modulo operator+(Modulo m, Modulo n) {
18.     return Modulo(m.numb + n.numb);
19. }
20.
21. int main() {
22.     Modulo m(5), n(6), k;
23.
24.     k = m + n;
25.     cout << m.numb << " + " << n.numb
26.          << " (mod " << Modulo::modul
27.          << ") = " << k.numb << endl;
28.
29.     k = operator+(m,n);
30.     cout << m.numb << " + " << n.numb
31.          << " (mod " << Modulo::modul
32.          << ") = " << k.numb << endl;
33. }
```

Objects of the class represent integer numbers *modulo* 7 (any number is represented by the remainder it gives when divided by 7; e.g., 11 and 32 are both represented by 4). There are only seven distinct values of **Modulo** — $[0,..., 6]$. When two such numbers are added, the result should take one of the seven possible values. For example

$$5+6 \equiv 4 \pmod 7$$

We redefine then the operation of adding numbers of type **Modulo**. On lines 17-19 we define a function named `operator+`, with two parameters of type **Modulo**. This function will be called whenever a ' +' symbol is encountered with expressions of type **Modulo** as operands on both sides of the operator. This will happen on line 24: our function will be invoked and will return an object of type **Modulo** representing the sum of the arguments (or whatever we put into the function's definition).

Line 29 shows that, if we insist, the function can be called explicitly “by its name”; the result will be identical:

```
5 + 6 (mod 7) = 4
5 + 6 (mod 7) = 4
```

In this example overloading subtraction, incrementation or multiplication would be quite natural; we do not do it to keep the example short.

Note that the type of the result is **Modulo**; it is returned by value and hence is *not* an l-value (in accordance to what one would expect for numbers).

The '<<' (stream insertion) operator belongs to the most often overloaded operators. Overloading this operator makes it possible to output string representation of objects of user-defined types in the same way as for built-in types. What should be the type of

parameters for the corresponding function? When '<<' operator is used, we have a stream object (e.g., ***cout***) on the left hand side and "something" to be printed on the right hand side. This "something" can be of type **int**, **double** etc.; compiler will know what to do in all these cases because its behavior in such situations has already been programmed. There will be a problem, however, if this "something" on the right hand side is of a user-defined type. To handle this situation, we have to define an overloading of '`operator<<`' function with parameters of types **ostream&** (left operand) and **AClass** or **AClass&** (right opearand), where **AClass** is the name of our class. What about the return type? If all we need is something like

```
cout << zzz;
```

where ***zzz*** is the identifier of an object of our class, then the return type would basically be irrelevant. However, if we want '<<' to work sequentially, in a cascade,

```
cout << zzz << " " << yyy;
```

then the value of '`cout << zzz`' should be again ***cout***, so it becomes the left operand of the next stream insertion operator. Hence, the prototype of the overloading function should be

```
ostream& operator<<(ostream&, const AClass&);
```

The first parameter, of type **ostream&**, cannot be **const**, because insertion to a stream *can* modify an object representing it. It cannot be **ostream** either, because passing the argument by value would require copying it, but the copy constructor in class **ostream** is **private**; for the same reason the result has to be returned by reference and not by value.

An object of our class **AClass** (second parameter) can be passed either by value or by reference. In the former case, we have to consider if the class has been equipped in a correct copy constructor and if the process of copying will not deteriorate efficiency too much. In the second case, we have to remember that the function will work on the original object — it is therefore reasonable — although not obligatory — to declare this parameter as **const**: after all, the function's task is to display some information, not to modify any objects.

Functions overloading stream insertion operator often need access to private members of the class they are defined to work with; in such cases one can declare them as friends of the class:

```
class AClass {
    // ...
    friend ostream& operator<<(ostream&, const AClass&);
}
```

The function should always return a reference to exactly the same stream object which has been passed as the first argument — of course, it does not have to be ***cout***; it can be any object of class **ostream** or a class derived from it.

Let us consider an example:

P150: strinsop.cpp Overloading stream insertion operator

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
```

```
 4.
 5. class Person {
 6.     string name;
 7.     int    age;
 8. public:
 9.     Person(const string& name, int age)
10.         : name(name), age(age)
11.     { }
12.
13.      // ... other members
14.
15.     friend ostream& operator<<(ostream&, const Person&);
16. };
17.
18. ostream& operator<<(ostream& str, const Person& k) {
19.     return str << k.name << " (" << k.age << " yo)";
20. }
21.
22. int main() {
23.     Person t[] = {  Person("Joe",18), Person("Sue",26),
24.                     Person("Eve",35), Person("Tim",11)  };
25.
26.     for (int i = 0; i < 4; i++)
27.         cout << t[i] << endl;
28. }
```

On line 15, inside the definition of class **Person**, we declare that the function overloading '<<' operator for objects of the class will be its friend; in this way the function, although not a member of the class, can nevertheless directly access private members ***name*** and ***age***, as we can see from the program's output

```
Joe (18 yo)
Sue (26 yo)
Eve (35 yo)
Tim (11 yo)
```

The overloading function itself is defined On lines 18-20 according to the rules we have described above. It is a global function so it is not called for an object: the object which is meant here is the right operand of '<<' operator and is passed to the function as the second argument by reference (there is no **this** pointer in the function). Note that the value of the whole expression appearing in the **return** statement is the stream object ***str***, which is exactly what the function should return — due to this, the body of the function could have been reduced to one **return** statement.

An overloaded binary operator can have *both* parameters of, possibly different, user-defined types. In the example below ' +' operator is overloaded twice: one version adds vectors (objects of class **Vector**) to vectors with the result being another vector, the other adds vectors to points giving points. The compiler will be able to select the correct version on the basis of types of arguments (lines 50 and 53):

P151: vecpoin.cpp Overloading binary operators

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. class Point;
 5.
 6. class Vector {
 7.     double x, y, z;
 8. public:
```

```
9.      Vector(double x = 0, double y = 0, double z = 0)
10.         : x(x), y(y), z(z)
11.     { }
12.     friend Point    operator+(const Point&,  const Vector&);
13.     friend Vector   operator+(const Vector&, const Vector&);
14.     friend ostream& operator<<(ostream&, const Vector&);
15. };
16.
17. class Point {
18.     double x, y, z;
19. public:
20.     Point(double x = 0, double y = 0, double z = 0)
21.         : x(x), y(y), z(z)
22.     { }
23.     friend Point operator+(const Point&, const Vector&);
24.     friend ostream& operator<<(ostream&, const Point&);
25. };
26.
27. Point operator+(const Point&  p, const Vector& v) {
28.     return Point(p.x+v.x, p.y+v.y, p.z+v.z);
29. }
30.
31. Vector operator+(const Vector& v1, const Vector& v2) {
32.     return Vector(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
33. }
34.
35. ostream& operator<<(ostream& str, const Point& p) {
36.     return str << "P(" << p.x << "," << p.y
37.                << ","  << p.z << ")";
38. }
39.
40. ostream& operator<<(ostream& str, const Vector& v) {
41.     return str << "V[" << v.x << "," << v.y
42.                << ","  << v.z << "]";
43. }
44.
45. int main() {
46.
47.     Vector v1(1,1,1), v2(2,2,2);
48.     Point  p1(1,2,3);
49.
50.     Vector v = v1 + v2;
51.     cout << "v: " << v << endl;
52.
53.     Point  p = p1 + v;
54.     cout << "p: " << p << endl;
55. }
```

Functions overloading addition and stream insertion use private members of the classes directly, therefore they had to be declared as **friend** s in the definitions of the classes. Note that forward declaration on line 4 is necessary here: we use the name **Point** in the definition of **Vector** and *vice versa*. Note also that both addition functions return result by value so it is *not* an l-value: that is how addition normally behaves.

19.2.2 Unary operators

Unary (one-argument) operators can be overloaded by global one-parameter functions with prototype of the form

```
Type operator@(Typearg);
```

Type **Typearg** must be a user-defined type, type **Type** is arbitrary. As before, symbol '' stands here for the symbol of any unary operator (like '&' or '!'). When such a function is defined, and ***a*** denotes an object (or reference to an object) of a user-defined type, expression

```
@a
```

is equivalent to invocation

```
operator@(a)
```

In the example below, we overload unary operator ' ! ' for objects of class **AClass**. This class has a field of type **string**, and the overloading function defined on lines 13-15 returns values of type **bool** which answer the question whether the string has more than 5 characters:

P152: oneargop.cpp Overloading unary operators

```
1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  struct AClass {
6.      string name;
7.
8.      AClass(const string& name)
9.          : name(name)
10.     { }
11. };
12.
13. bool operator!(const AClass& c) {
14.     return c.name.size() > 5;
15. }
16.
17. int main() {
18.     AClass t[] = { AClass("Marlon"), AClass("Henry"),
19.                    AClass("Dave"),   AClass("Horatio"),
20.                    AClass("Sue"),    AClass("Alice")   };
21.
22.     for (int i = 0; i < 6; ++i)
23.         if ( !t[i] ) cout << t[i].name << endl;
24. }
```

On line 23 we print names corresponding to obejcts from array ***t***; the use of ' ! ' operator causes only long names to be printed:

```
Marlon
Horatio
```

Prev: 19.1 Introduction **Up:** 19. Operator overloading. Move semantics **Next:** 19.3 Overloading with methods of

Subsections

- 19.3.1 Binary operators
- 19.3.2 Unary operators

19.3 Overloading with methods of classes

Operators, both unary and binary, can also be overloaded with class methods. In this case, as it is always with methods, one of arguments will be passed implicitly, without specifying it on the parameter list — this will be the pointer to the object for which the method has been invoked: inside the definition of the method one can refer to this pointer under the name **this**. Therefore, binary operators will be defined as *one*-parameter methods, and unary operators — as methods with empty parameter list.

Some operators are special and can be overloaded *only* with class methods and never with global functions. These are:

- assignment (' `=`') — binary, so overloaded with a parameterless method; the same applies to varinats ' `+=`', ' `*=`' etc.;
- function call (' `()`') — as the only operator can have arbitrary number of parameters;
- subscripting (indexing) (' `[]`') — binary, so overloaded with a one-parameter method;
- indirect member selection (' `->`') — unary, overloaded with a parameterless method.

These operators will be described separately later in this chapter.

19.3.1 Binary operators

When defining a method overloading a binary operator, we do not specify the first parameter: as for all methods it will be implicitly assumed to be the pointer to the object for which the method is called. This will always be an object which is *the left* operand of an operator. The right operand will be passed as the only explicit argument.

Suppose we have defined, in a class **AClass**, a method with prototype

```
Type operator@(Typearg);
```

or

```
Type operator@(Typearg&);
```

If ***a*** is an idnetifier (or a reference to) an object of class **AClass** and ***b*** is of type **Typearg**, then an expression

```
a @ b
```

will be equivalent to invoking the method for object ***a*** and passing, by value or by reference,

the value of ***b*** as the argument:

```
a.operator@(b)
```

The symbol ", as before, stands here for one of binary operator symbols from table of operators.

Let us consider another version of class **Modulo** from the program `modsev.cpp`. Here we have modified the overloading of addition operator: now it is implemented as a one-parameter method of the class (lines 19-21, declared on line 14):

P153: modsev1.cpp Overloading with a method

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Modulo {
5.      int numb;
6.  public:
7.      static const int modul;
8.      Modulo() : numb(0)
9.      { }
10.
11.     Modulo(int numb) : numb(numb%modul)
12.     { }
13.
14.     Modulo operator+(const Modulo&) const;
15.     friend ostream& operator<<(ostream&, const Modulo&);
16. };
17. const int Modulo::modul = 7;
18.
19. inline Modulo Modulo::operator+(const Modulo& n) const {
20.     return Modulo(numb + n.numb);
21. }
22.
23. ostream& operator<<(ostream& str, const Modulo& n) {
24.     return str << n.numb;
25. }
26.
27. int main() {
28.     Modulo m(5), n(6), k;
29.
30.     k = m + n;
31.     cout << m << " + " << n
32.          << " (mod " << Modulo::modul
33.          << ") = " << k << endl;
34.
35.     k = k + 8;
36.     cout << "k + 8 (mod " << Modulo::modul
37.          << ") = " << k << endl;
38. }
```

The method will be invoked when an expression

```
m + k
```

is encounterd, with ***m*** and ***k*** being objects of class **Modulo**. It will be called for object ***m***, while ***k*** will be passed as the argument (by reference, according to the signature of the method). This is what we can see on line 30.

Note that overloading a binary operator with a method implies that it will be called if the *left* operand is an object of our class. Therefore, we cannot do it for '<<' operator, as in this case the object appearing on the left-hand side of the operator is of type **ostream** — we

would have to define this overloading in class **ostream**! That is why we used a friend global function to overload this operator (lines 23-25).

One can notice a mysterious phenomenon on line 35 ('`k=k+8`'). We have an object of class **Modulo** on the left-hand side of the '`+`' operator, but a plain **int** on the right-hand side. We could make it work by defining a method with prototype

```
Modulo operator+(int);
```

but we did not do it. Still, the program does work and gives

```
5 + 6 (mod 7) = 4
k + 8 (mod 7) = 5
```

which looks correct, as $4 + 8 = 12 \equiv 5 (\mathrm{mod}\ 7)$. Why does it work? This will be explained in more detail in chapter on conversions . In short: the compiler will try to convert **int** value into an object of class **Modulo** *and will succeed* to do so due to the presence of a constructor of **Modulo** which takes one **int** (lines 11-12).

As the second example let us consider a class implementing singly linked list, somewhat differently than that from program `simplist.c`. Class **List** has only one field — a pointer ***head*** which points to the first node of the list. Nodes themselves are objects of class (structure) **Node** (lines 6-13). This class is declared (and defined) inside the private section of class **List** and is therefore not accessible from the outside. Objects of this class contain a data (an **int** in our simple case) and a pointer to the next element of the list.

P154: list.cpp Lists with overloaded operators

```
1. #include <iostream>
2. using namespace std;
3.
4. class List {
5.
6.     struct Node {
7.         int    elem;
8.         Node* next;
9.
10.        Node(int elem, Node* next = 0)
11.            : elem(elem), next(next)
12.        { }
13.    };
14.
15.    Node* head;
16.
17. public:
18.    List()
19.        : head(0)
20.    { }
21.
22.    List& operator+(int elem) {
23.        Node* w = new Node(elem);
24.        if (head) {
25.            Node *h = head;
26.            while (h->next) h = h->next;
27.            h->next = w;
28.        } else {
29.            head = w;
30.        }
31.        return *this;
32.    }
33.
```

```
34.     List& operator-(int elem) {
35.         head = new Node(elem,head);
36.         return *this;
37.     }
38.
39.     int operator!() const {
40.         int cnt = 0;
41.         for (Node* h = head; h ; h = h->next, ++cnt);
42.         return cnt;
43.     }
44.
45.     ~List() {
46.         Node *prev, *curr = head;
47.         while (curr) {
48.             prev = curr;
49.             curr = curr->next;
50.             cerr << "deleting " << prev->elem << endl;
51.             delete prev;
52.         }
53.     }
54.
55.     friend ostream& operator<<(ostream&, const List&);
56. };
57.
58. ostream& operator<<(ostream& s, const List& L) {
59.         for(List::Node* h = L.head ; h ; h = h->next)
60.             s << h->elem << " ";
61.         return s;
62. }
63.
64. int main() {
65.     List list;
66.
67.     list + 1;
68.     list + 2 - 0 - (-1);
69.     cout <<  list+3  << endl;
70.     cout << "List has " << !list << " elements" << endl;
71. }
```

In order to be able to add element to an existing list, we have overloaded binary operators of addition and subtraction. The method overloading '`+`' operator (lines 22-32) creates new node with data equal to the right operand of the '`+`' operator (which is the only argument passed to the method). The new node is then linked to the list at its end (code in lines 25-27). Note that the method returns the reference to the object it has been called for ('`*this`'). For example, if ***list*** is an object of class **List**, evaluation of expression

```
list + 5
```

will cause:

- invocation of the method **operator+** with argument equal to 5;
- creation of an object of class **Node** with its member ***elem*** equal to 5 and ***next*** equal to **nullptr** (line 23; we use the default value declared on line 10);
- linking the node to the end of the list;
- returning a reference to the list itself.

The value of the whole expression is a reference to the list, so it is possible to invoke other methods in a cascade, e.g., the statement

```
list + 5 + 7
```

causes addition to the list an element with data equal to 5 and then adding to the resulting list another element, this time with element equal to 7. This kind of expression is used on lines 68-69 of the program.

Subtraction operator is overloaded in a similar way, but new nodes are added at the beginning of the list, becoming its head (linie 34-37).

On lines 39-43 we overload the '`!`' operator. It is a unary operator, so it will be overloaded with a parameterless method. As one can see from its definition, it returns the number of elements in the list: it is used on line 70 of the program (see also the next sunsection).

The printout of the program is

```
-1 0 1 2 3
List has 5 elements
deleting -1
deleting 0
deleting 1
deleting 2
deleting 3
```

We can see also a trace of the destructor of the class: all nodes of the list are deleted when the program exits the **main** function. The printout has been produced with the help of overloaded stream insertion operator; the function overloading this operator (lines 58-62) is a friend of the class, as it needs access not only to member ***head*** but also to the name **Node** declared in the scope of the class **List** in its private section. The function, as we already know, has to be a global function, not a method of our class.

19.3.2 Unary operators

Unary (one-parameter) operators can be overloaded with parameterless methods. The pointer to an object for which the method is called will be passed as their implicit argument, as always for nonstatic methods. For unary prefix operators, notation

```
@a
```

is equivalent to invocation

```
a.operator@()
```

and declaration of the corresponding method has the form

```
Type operator@();
```

We have already seen an example of such overloading in the previous subsection, in the program `list.cpp`. Expression '`!list`' (line 70) triggers a call of method **operator!** (defined in lines 39-43) for the object ***list*** .

There is a little problem with unary operators '`++`' and '`--`'. They need a special consideration, because, as we know, they come in two flavors: as prefix and postfix operators, i.e., they can precede or follow their operand. We can overload them, as other unary operators, with a parameterless method and this will overload their *prefix* form, i.e., the corresponding methods will be invoked when expressions like

```
++a; --b;
```

are encountered. Of course, according to the principle of *least astonishment*, they should have something to do with incrementation or decrementation and they should return an l-value: this is what any user would expect by analogy with their implementation for numeric types.

The prototype of a method overloading, say, preincrementation, will look like this:

```
Type operator++();
```

But for postincrementation it should look the same — the name must be **operator++** and it should also be a parameterless method, as it overloads a unary operator. To distinguish between the two, for *postfix* operators (postincrementation, postdecrementation) one spurious parameter of type **int** is added to overloading function. The variable corresponding to this parameter is not used in the body of the function (formally, it could be used — its value will always be 0), so one does not have to assign any name to the parameter; its presence itself signals to the compiler that a postfix operator is meant.

Therefore, declaration of a function oveloading *postfix* incrementation operator would have the form

```
Typ operator++(int);
```

As usually, when overloading prefix and postfix incrementation or decrementation operators, one should remember about the principle of least astonishment. Operators in *postfix* forms should return *unmodified* values of their argument, and modification of arguments should be a side effect (having something to do with an incrementation or decrementation). The values returned should *not* be l-values. Usually, a copy of *this* object is created, then *this* object is modified, and finally the copy is returned by value.

For *prefix* forms, the value returned should be equal to already *modified* value of argument and *should be* an l-value. Therefore, methods overloading such operators usually return a reference to *this* object by executing `return *this` at the end.

The '-' and '+' operators have the binary and unary forms; this, however, does not pose any problem, because both forms can easily be distinguished by the number of parameters: binary '+' overloaded with a method will have one parameter, while the unary version will be a parameterless method.

In the example below, we overload all operators “with -”: binary and unary, the latter in both postfix and prefix forms:

P155: minus.cpp Overloading “minus” operators

```
1. #include <iostream>
2. using namespace std;
3.
4. class A {
5.     int data;
6. public:
7.     A(int data = 0) : data(2*(data/2)) { }
8.
9.     const A operator-() const {
10.         return A(-data);
11.     }
12.
13.     const A operator-(const A& a) const {
14.         return A(data - a.data);
15.     }
16.
17.     A& operator--() {
18.         ----data;
```

```
19.         return *this;
20.     }
21.
22.     const A operator--(int) {
23.         A x(data);
24.         ----data;
25.         return x;
26.     }
27.
28.     friend ostream& operator<<(ostream&,A);
29. };
30.
31. ostream& operator<<(ostream& strum, A d) {
32.     return strum << d.data;
33. }
34.
35. int main() {
36.     A data(7);
37.
38.     cout << "a.   data    = " <<   data   << endl;
39.     cout << "b.   data--  = " <<   data-- << endl;
40.     cout << "c.   data    = " <<   data   << endl;
41.     cout << "d. --data    = " << --data   << endl;
42.     cout << "e.   data    = " <<   data   << endl;
43.     cout << "f.  -data    = " <<  -data   << endl;
44.     cout << "g.   data    = " <<   data   << endl;
45. }
```

Class **A** has only one field ***data*** of type **int**. A number stored in this member will always be even: this is ensured by the only constructor of the class and by the fact that decrementing is implemented in such a way that the value of ***data*** is deceremented always by two. We can see the following overloadingsL:

- unary '-' operator, i.e., sign change operator (lines 9-11). A new object is returned by value; the original, *this* object, is not modified, so the method have been declared as **const**;
- binary '-' operator, i.e., subtracting operator (lines 13-15). It returns by value a new object, representing the difference between *this* object and the one passed by argument (i.e., the right hand side operand of the operator). The original, *this* object, is not modified, so again the method is declared as **const**;
- unary predecrementation operator ' –' (lines 17-20). The method decrements the member ***data*** by 2 and returns a reference to *this* object. Therefore, the value returned is an l-value and is equal to the value of the object *after* modification;
- unary postdecrementation operator (lines 22-26). A spurious argument of type **int** has been added. We do not use it, so it is not even given a name; its only purpose is to inform the compiler that we are oveloading *post*decrementation, and not *pre*decrementation. The methods produces a copy of *this* object, then modifies the member ***data***, and then it returns by value the *copy* which has been made before modification. The value returned is *not* an l-value.

We also overload the '<<' operator with a friend global function defined on lines 31-33. The printout

```
a.   data    = 6
b.   data--  = 6
c.   data    = 4
d. --data    = 2
e.   data    = 2
f.  -data    = -2
g.   data    = 2
```

shows that everything works as expected.

Let us consider one more example:

P156: tabinc.cpp Overloading incrementation operator

```
1. #include <iostream>
2. #include <cstring>  // memcpy
3. using namespace std;
4.
5. class Tablica {
6.     int size;
7.     int* tab;
8. public:
9.     Tablica(int size, const int* t)
10.         : size(size),
11.           tab((int*)memcpy(new int[size], t,
12.                            size*sizeof(int)))
13.     { }
14.
15.     Tablica(const Tablica& t)
16.         : size(t.size),
17.           tab((int*)memcpy(new int[size], t.tab,
18.                            size*sizeof(int)))
19.     { }
20.
21.     ~Tablica() { delete [] tab; }
22.
23.     Tablica& operator++();
24.     Tablica operator++(int);
25.     void showTab(const char* nap);
26. };
27.
28. Tablica& Tablica::operator++() {
29.     for (int i = 0; i < size; ++i)
30.         ++tab[i];
31.     return *this;
32. }
33.
34. Tablica Tablica::operator++(int) {
35.     Tablica t(*this);
36.     ++*this;
37.     return t;
38. }
39.
40. void Tablica::showTab(const char* nap) {
41.     cout << nap;
42.     for (int i = 0; i < size; i++)
43.         cout << tab[i] << " ";
44.     cout << endl;
45. }
46.
47. int main() {
48.     int tab[] = {1,2,3,4};
49.
50.     Tablica T(4,tab);
51.     T.showTab("Tablica wyjsciowa T: ");
52.     Tablica t = ++T;
53.     t.showTab("  Po t = ++T t jest: ");
54.     T.showTab("           a T jest: ");
55.
56.     Tablica S(4,tab);
57.     S.showTab("Tablica wyjsciowa S: ");
58.     Tablica s = S++;
59.     s.showTab("  Po s = S++ s jest: ");
60.     S.showTab("           a S jest: ");
61. }
```

Class **Array** has a pointer field pointing to a dynamically allocated array of **int** s. Therefore,

we need a destructor to deallocate memory occupied by the array when an object is deleted. Every object holds information on the size of array pointed to by ***arr*** in member ***size***.

Note how arrays are copied in constructors (lines 11 and 17). We copy the whole region of memory, instead of copying elements of the array one by one in a loop. This method is much more efficient for simple types, but *can fail* if elements of the array are of an object type!

Note that postfix incrementation operator, defined in lines 34-38, returns an object which is identical to *this* object *before* modification. Incrementation is a side effect which will be visible when the object is accessed next time. In the definition of postincrementation (line 36), we used the *prefix* incrementation operator, defined earlier, on lines 28-32. On line 35, using the copy-constructor, we create a copy of *this* object, and then invoke preincrementation for ***this** what modifies the object that the method has been invoked for; afterwards we return (line 37) the unmodified copy.

The program prints:

```
Original array T is: 1 2 3 4
 after t = ++T t is: 2 3 4 5
           and T is: 2 3 4 5
Original array S is: 1 2 3 4
 after s = ++S s is: 1 2 3 4
           and S is: 2 3 4 5
```

The class has a pointer field, so normally should be equipped with a custom assignment operator which would perform *deep* assignment, i.e., applying not to the member pointer, but rather the data it points to. Overloading ' =' operator will be discussed in the next subsection.

Prev: 19.2 Overloading with global functions **Up:** 19. Operator overloading. Move semantics **Next:** 19.4 “Special” operators

Subsections

- 19.3.1 Binary operators
- 19.3.2 Unary operators

19.3 Overloading with methods of classes

Operators, both unary and binary, can also be overloaded with class methods. In this case, as it is always with methods, one of arguments will be passed implicitly, without specifying it on the parameter list — this will be the pointer to the object for which the method has been invoked: inside the definition of the method one can refer to this pointer under the name **this**. Therefore, binary operators will be defined as *one*-parameter methods, and unary operators — as methods with empty parameter list.

Some operators are special and can be overloaded *only* with class methods and never with global functions. These are:

- assignment (' `=`') — binary, so overloaded with a parameterless method; the same applies to varinats ' `+=`', ' `*=`' etc.;
- function call (' `()`') — as the only operator can have arbitrary number of parameters;
- subscripting (indexing) (' `[]`') — binary, so overloaded with a one-parameter method;
- indirect member selection (' `->`') — unary, overloaded with a parameterless method.

These operators will be described separately later in this chapter.

19.3.1 Binary operators

When defining a method overloading a binary operator, we do not specify the first parameter: as for all methods it will be implicitly assumed to be the pointer to the object for which the method is called. This will always be an object which is *the left* operand of an operator. The right operand will be passed as the only explicit argument.

Suppose we have defined, in a class **AClass**, a method with prototype

```
Type operator@(Typearg);
```

or

```
Type operator@(Typearg&);
```

If ***a*** is an idnetifier (or a reference to) an object of class **AClass** and ***b*** is of type **Typearg**, then an expression

```
a @ b
```

will be equivalent to invoking the method for object ***a*** and passing, by value or by reference,

the value of ***b*** as the argument:

```
a.operator@(b)
```

The symbol ", as before, stands here for one of binary operator symbols from table of operators.

Let us consider another version of class **Modulo** from the program `modsev.cpp`. Here we have modified the overloading of addition operator: now it is implemented as a one-parameter method of the class (lines 19-21, declared on line 14):

P153: modsev1.cpp Overloading with a method

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Modulo {
5.      int numb;
6.  public:
7.      static const int modul;
8.      Modulo() : numb(0)
9.      { }
10.
11.     Modulo(int numb) : numb(numb%modul)
12.     { }
13.
14.     Modulo operator+(const Modulo&) const;
15.     friend ostream& operator<<(ostream&, const Modulo&);
16. };
17. const int Modulo::modul = 7;
18.
19. inline Modulo Modulo::operator+(const Modulo& n) const {
20.     return Modulo(numb + n.numb);
21. }
22.
23. ostream& operator<<(ostream& str, const Modulo& n) {
24.     return str << n.numb;
25. }
26.
27. int main() {
28.     Modulo m(5), n(6), k;
29.
30.     k = m + n;
31.     cout << m << " + " << n
32.          << " (mod " << Modulo::modul
33.          << ") = " << k << endl;
34.
35.     k = k + 8;
36.     cout << "k + 8 (mod " << Modulo::modul
37.          << ") = " << k << endl;
38. }
```

The method will be invoked when an expression

```
m + k
```

is encounterd, with ***m*** and ***k*** being objects of class **Modulo**. It will be called for object ***m***, while ***k*** will be passed as the argument (by reference, according to the signature of the method). This is what we can see on line 30.

Note that overloading a binary operator with a method implies that it will be called if the *left* operand is an object of our class. Therefore, we cannot do it for '<<' operator, as in this case the object appearing on the left-hand side of the operator is of type **ostream** — we

would have to define this overloading in class **ostream**! That is why we used a friend global function to overload this operator (lines 23-25).

One can notice a mysterious phenomenon on line 35 ('`k=k+8`'). We have an object of class **Modulo** on the left-hand side of the '`+`' operator, but a plain **int** on the right-hand side. We could make it work by defining a method with prototype

```
Modulo operator+(int);
```

but we did not do it. Still, the program does work and gives

```
5 + 6 (mod 7) = 4
k + 8 (mod 7) = 5
```

which looks correct, as $4 + 8 = 12 \equiv 5 (\mathrm{mod}\ 7)$. Why does it work? This will be explained in more detail in chapter on conversions . In short: the compiler will try to convert **int** value into an object of class **Modulo** *and will succeed* to do so due to the presence of a constructor of **Modulo** which takes one **int** (lines 11-12).

As the second example let us consider a class implementing singly linked list, somewhat differently than that from program `simplist.c`. Class **List** has only one field — a pointer ***head*** which points to the first node of the list. Nodes themselves are objects of class (structure) **Node** (lines 6-13). This class is declared (and defined) inside the private section of class **List** and is therefore not accessible from the outside. Objects of this class contain a data (an **int** in our simple case) and a pointer to the next element of the list.

P154: list.cpp Lists with overloaded operators

```
1. #include <iostream>
2. using namespace std;
3.
4. class List {
5.
6.     struct Node {
7.         int    elem;
8.         Node* next;
9.
10.        Node(int elem, Node* next = 0)
11.            : elem(elem), next(next)
12.        { }
13.    };
14.
15.    Node* head;
16.
17. public:
18.    List()
19.        : head(0)
20.    { }
21.
22.    List& operator+(int elem) {
23.        Node* w = new Node(elem);
24.        if (head) {
25.            Node *h = head;
26.            while (h->next) h = h->next;
27.            h->next = w;
28.        } else {
29.            head = w;
30.        }
31.        return *this;
32.    }
33.
```

```
34.     List& operator-(int elem) {
35.         head = new Node(elem,head);
36.         return *this;
37.     }
38.
39.     int operator!() const {
40.         int cnt = 0;
41.         for (Node* h = head; h ; h = h->next, ++cnt);
42.         return cnt;
43.     }
44.
45.     ~List() {
46.         Node *prev, *curr = head;
47.         while (curr) {
48.             prev = curr;
49.             curr = curr->next;
50.             cerr << "deleting " << prev->elem << endl;
51.             delete prev;
52.         }
53.     }
54.
55.     friend ostream& operator<<(ostream&, const List&);
56. };
57.
58. ostream& operator<<(ostream& s, const List& L) {
59.         for(List::Node* h = L.head ; h ; h = h->next)
60.             s << h->elem << " ";
61.         return s;
62. }
63.
64. int main() {
65.     List list;
66.
67.     list + 1;
68.     list + 2 - 0 - (-1);
69.     cout <<  list+3  << endl;
70.     cout << "List has " << !list << " elements" << endl;
71. }
```

In order to be able to add element to an existing list, we have overloaded binary operators of addition and subtraction. The method overloading '`+`' operator (lines 22-32) creates new node with data equal to the right operand of the '`+`' operator (which is the only argument passed to the method). The new node is then linked to the list at its end (code in lines 25-27). Note that the method returns the reference to the object it has been called for ('`*this`'). For example, if ***list*** is an object of class **List**, evaluation of expression

```
list + 5
```

will cause:

- invocation of the method **operator+** with argument equal to 5;
- creation of an object of class **Node** with its member ***elem*** equal to 5 and ***next*** equal to **nullptr** (line 23; we use the default value declared on line 10);
- linking the node to the end of the list;
- returning a reference to the list itself.

The value of the whole expression is a reference to the list, so it is possible to invoke other methods in a cascade, e.g., the statement

```
list + 5 + 7
```

causes addition to the list an element with data equal to 5 and then adding to the resulting list another element, this time with element equal to 7. This kind of expression is used on lines 68-69 of the program.

Subtraction operator is overloaded in a similar way, but new nodes are added at the beginning of the list, becoming its head (linie 34-37).

On lines 39-43 we overload the '`!`' operator. It is a unary operator, so it will be overloaded with a parameterless method. As one can see from its definition, it returns the number of elements in the list: it is used on line 70 of the program (see also the next sunsection).

The printout of the program is

```
-1 0 1 2 3
List has 5 elements
deleting -1
deleting 0
deleting 1
deleting 2
deleting 3
```

We can see also a trace of the destructor of the class: all nodes of the list are deleted when the program exits the **main** function. The printout has been produced with the help of overloaded stream insertion operator; the function overloading this operator (lines 58-62) is a friend of the class, as it needs access not only to member ***head*** but also to the name **Node** declared in the scope of the class **List** in its private section. The function, as we already know, has to be a global function, not a method of our class.

19.3.2 Unary operators

Unary (one-parameter) operators can be overloaded with parameterless methods. The pointer to an object for which the method is called will be passed as their implicit argument, as always for nonstatic methods. For unary prefix operators, notation

```
@a
```

is equivalent to invocation

```
a.operator@()
```

and declaration of the corresponding method has the form

```
Type operator@();
```

We have already seen an example of such overloading in the previous subsection, in the program `list.cpp`. Expression '`!list`' (line 70) triggers a call of method **operator!** (defined in lines 39-43) for the object ***list*** .

There is a little problem with unary operators '`++`' and '`--`'. They need a special consideration, because, as we know, they come in two flavors: as prefix and postfix operators, i.e., they can precede or follow their operand. We can overload them, as other unary operators, with a parameterless method and this will overload their *prefix* form, i.e., the corresponding methods will be invoked when expressions like

```
++a; --b;
```

are encountered. Of course, according to the principle of *least astonishment*, they should have something to do with incrementation or decrementation and they should return an l-value: this is what any user would expect by analogy with their implementation for numeric types.

The prototype of a method overloading, say, preincrementation, will look like this:

```
Type operator++();
```

But for postincrementation it should look the same — the name must be **operator++** and it should also be a parameterless method, as it overloads a unary operator. To distinguish between the two, for *postfix* operators (postincrementation, postdecrementation) one spurious parameter of type **int** is added to overloading function. The variable corresponding to this parameter is not used in the body of the function (formally, it could be used — its value will always be 0), so one does not have to assign any name to the parameter; its presence itself signals to the compiler that a postfix operator is meant.

Therefore, declaration of a function oveloading *postfix* incrementation operator would have the form

```
Typ operator++(int);
```

As usually, when overloading prefix and postfix incrementation or decrementation operators, one should remember about the principle of least astonishment. Operators in *postfix* forms should return *unmodified* values of their argument, and modification of arguments should be a side effect (having something to do with an incrementation or decrementation). The values returned should *not* be l-values. Usually, a copy of *this* object is created, then *this* object is modified, and finally the copy is returned by value.

For *prefix* forms, the value returned should be equal to already *modified* value of argument and *should be* an l-value. Therefore, methods overloading such operators usually return a reference to *this* object by executing `return *this` at the end.

The '-' and '+' operators have the binary and unary forms; this, however, does not pose any problem, because both forms can easily be distinguished by the number of parameters: binary '+' overloaded with a method will have one parameter, while the unary version will be a parameterless method.

In the example below, we overload all operators “with -”: binary and unary, the latter in both postfix and prefix forms:

P155: minus.cpp Overloading “minus” operators

```
1. #include <iostream>
2. using namespace std;
3.
4. class A {
5.     int data;
6. public:
7.     A(int data = 0) : data(2*(data/2)) { }
8.
9.     const A operator-() const {
10.         return A(-data);
11.     }
12.
13.     const A operator-(const A& a) const {
14.         return A(data - a.data);
15.     }
16.
17.     A& operator--() {
18.         ----data;
```

```
19.         return *this;
20.     }
21.
22.     const A operator--(int) {
23.         A x(data);
24.         ----data;
25.         return x;
26.     }
27.
28.     friend ostream& operator<<(ostream&,A);
29. };
30.
31. ostream& operator<<(ostream& strum, A d) {
32.     return strum << d.data;
33. }
34.
35. int main() {
36.     A data(7);
37.
38.     cout << "a.   data    = " <<   data   << endl;
39.     cout << "b.   data--  = " <<   data-- << endl;
40.     cout << "c.   data    = " <<   data   << endl;
41.     cout << "d. --data    = " << --data   << endl;
42.     cout << "e.   data    = " <<   data   << endl;
43.     cout << "f.  -data    = " <<  -data   << endl;
44.     cout << "g.   data    = " <<   data   << endl;
45. }
```

Class **A** has only one field ***data*** of type **int**. A number stored in this member will always be even: this is ensured by the only constructor of the class and by the fact that decrementing is implemented in such a way that the value of ***data*** is deceremented always by two. We can see the following overloadingsL:

- unary '-' operator, i.e., sign change operator (lines 9-11). A new object is returned by value; the original, *this* object, is not modified, so the method have been declared as **const**;
- binary '-' operator, i.e., subtracting operator (lines 13-15). It returns by value a new object, representing the difference between *this* object and the one passed by argument (i.e., the right hand side operand of the operator). The original, *this* object, is not modified, so again the method is declared as **const**;
- unary predecrementation operator ' –' (lines 17-20). The method decrements the member ***data*** by 2 and returns a reference to *this* object. Therefore, the value returned is an l-value and is equal to the value of the object *after* modification;
- unary postdecrementation operator (lines 22-26). A spurious argument of type **int** has been added. We do not use it, so it is not even given a name; its only purpose is to inform the compiler that we are oveloading *post*decrementation, and not *pre*decrementation. The methods produces a copy of *this* object, then modifies the member ***data***, and then it returns by value the *copy* which has been made before modification. The value returned is *not* an l-value.

We also overload the '<<' operator with a friend global function defined on lines 31-33. The printout

```
a.   data    = 6
b.   data--  = 6
c.   data    = 4
d. --data    = 2
e.   data    = 2
f.  -data    = -2
g.   data    = 2
```

shows that everything works as expected.

Let us consider one more example:

P156: tabinc.cpp Overloading incrementation operator

```
1. #include <iostream>
2. #include <cstring>  // memcpy
3. using namespace std;
4.
5. class Tablica {
6.     int size;
7.     int* tab;
8. public:
9.     Tablica(int size, const int* t)
10.         : size(size),
11.           tab((int*)memcpy(new int[size], t,
12.                            size*sizeof(int)))
13.     { }
14.
15.     Tablica(const Tablica& t)
16.         : size(t.size),
17.           tab((int*)memcpy(new int[size], t.tab,
18.                            size*sizeof(int)))
19.     { }
20.
21.     ~Tablica() { delete [] tab; }
22.
23.     Tablica& operator++();
24.     Tablica operator++(int);
25.     void showTab(const char* nap);
26. };
27.
28. Tablica& Tablica::operator++() {
29.     for (int i = 0; i < size; ++i)
30.         ++tab[i];
31.     return *this;
32. }
33.
34. Tablica Tablica::operator++(int) {
35.     Tablica t(*this);
36.     ++*this;
37.     return t;
38. }
39.
40. void Tablica::showTab(const char* nap) {
41.     cout << nap;
42.     for (int i = 0; i < size; i++)
43.         cout << tab[i] << " ";
44.     cout << endl;
45. }
46.
47. int main() {
48.     int tab[] = {1,2,3,4};
49.
50.     Tablica T(4,tab);
51.     T.showTab("Tablica wyjsciowa T: ");
52.     Tablica t = ++T;
53.     t.showTab("  Po t = ++T t jest: ");
54.     T.showTab("           a T jest: ");
55.
56.     Tablica S(4,tab);
57.     S.showTab("Tablica wyjsciowa S: ");
58.     Tablica s = S++;
59.     s.showTab("  Po s = S++ s jest: ");
60.     S.showTab("           a S jest: ");
61. }
```

Class **Array** has a pointer field pointing to a dynamically allocated array of **int** s. Therefore,

we need a destructor to deallocate memory occupied by the array when an object is deleted. Every object holds information on the size of array pointed to by ***arr*** in member ***size***.

Note how arrays are copied in constructors (lines 11 and 17). We copy the whole region of memory, instead of copying elements of the array one by one in a loop. This method is much more efficient for simple types, but *can fail* if elements of the array are of an object type!

Note that postfix incrementation operator, defined in lines 34-38, returns an object which is identical to *this* object *before* modification. Incrementation is a side effect which will be visible when the object is accessed next time. In the definition of postincrementation (line 36), we used the *prefix* incrementation operator, defined earlier, on lines 28-32. On line 35, using the copy-constructor, we create a copy of *this* object, and then invoke preincrementation for ***this** what modifies the object that the method has been invoked for; afterwards we return (line 37) the unmodified copy.

The program prints:

```
Original array T is: 1 2 3 4
 after t = ++T t is: 2 3 4 5
           and T is: 2 3 4 5
Original array S is: 1 2 3 4
 after s = ++S s is: 1 2 3 4
           and S is: 2 3 4 5
```

The class has a pointer field, so normally should be equipped with a custom assignment operator which would perform *deep* assignment, i.e., applying not to the member pointer, but rather the data it points to. Overloading ' =' operator will be discussed in the next subsection.

Prev: 19.2 Overloading with global functions **Up:** 19. Operator overloading. Move semantics **Next:** 19.4 “Special” operators

Subsections

- 19.3.1 Binary operators
- 19.3.2 Unary operators

19.3 Overloading with methods of classes

Operators, both unary and binary, can also be overloaded with class methods. In this case, as it is always with methods, one of arguments will be passed implicitly, without specifying it on the parameter list — this will be the pointer to the object for which the method has been invoked: inside the definition of the method one can refer to this pointer under the name **this**. Therefore, binary operators will be defined as *one*-parameter methods, and unary operators — as methods with empty parameter list.

Some operators are special and can be overloaded *only* with class methods and never with global functions. These are:

- assignment (' `=`') — binary, so overloaded with a parameterless method; the same applies to varinats ' `+=`', ' `*=`' etc.;
- function call (' `()`') — as the only operator can have arbitrary number of parameters;
- subscripting (indexing) (' `[]`') — binary, so overloaded with a one-parameter method;
- indirect member selection (' `->`') — unary, overloaded with a parameterless method.

These operators will be described separately later in this chapter.

19.3.1 Binary operators

When defining a method overloading a binary operator, we do not specify the first parameter: as for all methods it will be implicitly assumed to be the pointer to the object for which the method is called. This will always be an object which is *the left* operand of an operator. The right operand will be passed as the only explicit argument.

Suppose we have defined, in a class **AClass**, a method with prototype

```
Type operator@(Typearg);
```

or

```
Type operator@(Typearg&);
```

If ***a*** is an idnetifier (or a reference to) an object of class **AClass** and ***b*** is of type **Typearg**, then an expression

```
a @ b
```

will be equivalent to invoking the method for object ***a*** and passing, by value or by reference,

the value of ***b*** as the argument:

```
a.operator@(b)
```

The symbol ", as before, stands here for one of binary operator symbols from table of operators.

Let us consider another version of class **Modulo** from the program `modsev.cpp`. Here we have modified the overloading of addition operator: now it is implemented as a one-parameter method of the class (lines 19-21, declared on line 14):

P153: modsev1.cpp Overloading with a method

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Modulo {
5.      int numb;
6.  public:
7.      static const int modul;
8.      Modulo() : numb(0)
9.      { }
10.
11.     Modulo(int numb) : numb(numb%modul)
12.     { }
13.
14.     Modulo operator+(const Modulo&) const;
15.     friend ostream& operator<<(ostream&, const Modulo&);
16. };
17. const int Modulo::modul = 7;
18.
19. inline Modulo Modulo::operator+(const Modulo& n) const {
20.     return Modulo(numb + n.numb);
21. }
22.
23. ostream& operator<<(ostream& str, const Modulo& n) {
24.     return str << n.numb;
25. }
26.
27. int main() {
28.     Modulo m(5), n(6), k;
29.
30.     k = m + n;
31.     cout << m << " + " << n
32.          << " (mod " << Modulo::modul
33.          << ") = " << k << endl;
34.
35.     k = k + 8;
36.     cout << "k + 8 (mod " << Modulo::modul
37.          << ") = " << k << endl;
38. }
```

The method will be invoked when an expression

```
m + k
```

is encounterd, with ***m*** and ***k*** being objects of class **Modulo**. It will be called for object ***m***, while ***k*** will be passed as the argument (by reference, according to the signature of the method). This is what we can see on line 30.

Note that overloading a binary operator with a method implies that it will be called if the *left* operand is an object of our class. Therefore, we cannot do it for '<<' operator, as in this case the object appearing on the left-hand side of the operator is of type **ostream** — we

would have to define this overloading in class **ostream**! That is why we used a friend global function to overload this operator (lines 23-25).

One can notice a mysterious phenomenon on line 35 ('`k=k+8`'). We have an object of class **Modulo** on the left-hand side of the '`+`' operator, but a plain **int** on the right-hand side. We could make it work by defining a method with prototype

```
Modulo operator+(int);
```

but we did not do it. Still, the program does work and gives

```
5 + 6 (mod 7) = 4
k + 8 (mod 7) = 5
```

which looks correct, as $4 + 8 = 12 \equiv 5 (\mathrm{mod}\ 7)$. Why does it work? This will be explained in more detail in chapter on conversions . In short: the compiler will try to convert **int** value into an object of class **Modulo** *and will succeed* to do so due to the presence of a constructor of **Modulo** which takes one **int** (lines 11-12).

As the second example let us consider a class implementing singly linked list, somewhat differently than that from program `simplist.c`. Class **List** has only one field — a pointer ***head*** which points to the first node of the list. Nodes themselves are objects of class (structure) **Node** (lines 6-13). This class is declared (and defined) inside the private section of class **List** and is therefore not accessible from the outside. Objects of this class contain a data (an **int** in our simple case) and a pointer to the next element of the list.

P154: list.cpp Lists with overloaded operators

```
1. #include <iostream>
2. using namespace std;
3.
4. class List {
5.
6.     struct Node {
7.         int    elem;
8.         Node* next;
9.
10.        Node(int elem, Node* next = 0)
11.            : elem(elem), next(next)
12.        { }
13.    };
14.
15.    Node* head;
16.
17. public:
18.    List()
19.        : head(0)
20.    { }
21.
22.    List& operator+(int elem) {
23.        Node* w = new Node(elem);
24.        if (head) {
25.            Node *h = head;
26.            while (h->next) h = h->next;
27.            h->next = w;
28.        } else {
29.            head = w;
30.        }
31.        return *this;
32.    }
33.
```

```
34.     List& operator-(int elem) {
35.         head = new Node(elem,head);
36.         return *this;
37.     }
38.
39.     int operator!() const {
40.         int cnt = 0;
41.         for (Node* h = head; h ; h = h->next, ++cnt);
42.         return cnt;
43.     }
44.
45.     ~List() {
46.         Node *prev, *curr = head;
47.         while (curr) {
48.             prev = curr;
49.             curr = curr->next;
50.             cerr << "deleting " << prev->elem << endl;
51.             delete prev;
52.         }
53.     }
54.
55.     friend ostream& operator<<(ostream&, const List&);
56. };
57.
58. ostream& operator<<(ostream& s, const List& L) {
59.         for(List::Node* h = L.head ; h ; h = h->next)
60.             s << h->elem << " ";
61.         return s;
62. }
63.
64. int main() {
65.     List list;
66.
67.     list + 1;
68.     list + 2 - 0 - (-1);
69.     cout <<  list+3  << endl;
70.     cout << "List has " << !list << " elements" << endl;
71. }
```

In order to be able to add element to an existing list, we have overloaded binary operators of addition and subtraction. The method overloading '`+`' operator (lines 22-32) creates new node with data equal to the right operand of the '`+`' operator (which is the only argument passed to the method). The new node is then linked to the list at its end (code in lines 25-27). Note that the method returns the reference to the object it has been called for ('`*this`'). For example, if ***list*** is an object of class **List**, evaluation of expression

```
list + 5
```

will cause:

- invocation of the method **operator+** with argument equal to 5;
- creation of an object of class **Node** with its member ***elem*** equal to 5 and ***next*** equal to **nullptr** (line 23; we use the default value declared on line 10);
- linking the node to the end of the list;
- returning a reference to the list itself.

The value of the whole expression is a reference to the list, so it is possible to invoke other methods in a cascade, e.g., the statement

```
list + 5 + 7
```

causes addition to the list an element with data equal to 5 and then adding to the resulting list another element, this time with element equal to 7. This kind of expression is used on lines 68-69 of the program.

Subtraction operator is overloaded in a similar way, but new nodes are added at the beginning of the list, becoming its head (linie 34-37).

On lines 39-43 we overload the '`!`' operator. It is a unary operator, so it will be overloaded with a parameterless method. As one can see from its definition, it returns the number of elements in the list: it is used on line 70 of the program (see also the next sunsection).

The printout of the program is

```
-1 0 1 2 3
List has 5 elements
deleting -1
deleting 0
deleting 1
deleting 2
deleting 3
```

We can see also a trace of the destructor of the class: all nodes of the list are deleted when the program exits the **main** function. The printout has been produced with the help of overloaded stream insertion operator; the function overloading this operator (lines 58-62) is a friend of the class, as it needs access not only to member ***head*** but also to the name **Node** declared in the scope of the class **List** in its private section. The function, as we already know, has to be a global function, not a method of our class.

19.3.2 Unary operators

Unary (one-parameter) operators can be overloaded with parameterless methods. The pointer to an object for which the method is called will be passed as their implicit argument, as always for nonstatic methods. For unary prefix operators, notation

```
@a
```

is equivalent to invocation

```
a.operator@()
```

and declaration of the corresponding method has the form

```
Type operator@();
```

We have already seen an example of such overloading in the previous subsection, in the program `list.cpp`. Expression '`!list`' (line 70) triggers a call of method **operator!** (defined in lines 39-43) for the object ***list*** .

There is a little problem with unary operators '`++`' and '`--`'. They need a special consideration, because, as we know, they come in two flavors: as prefix and postfix operators, i.e., they can precede or follow their operand. We can overload them, as other unary operators, with a parameterless method and this will overload their *prefix* form, i.e., the corresponding methods will be invoked when expressions like

```
++a; --b;
```

are encountered. Of course, according to the principle of *least astonishment*, they should have something to do with incrementation or decrementation and they should return an l-value: this is what any user would expect by analogy with their implementation for numeric types.

The prototype of a method overloading, say, preincrementation, will look like this:

```
Type operator++();
```

But for postincrementation it should look the same — the name must be **operator++** and it should also be a parameterless method, as it overloads a unary operator. To distinguish between the two, for *postfix* operators (postincrementation, postdecrementation) one spurious parameter of type **int** is added to overloading function. The variable corresponding to this parameter is not used in the body of the function (formally, it could be used — its value will always be 0), so one does not have to assign any name to the parameter; its presence itself signals to the compiler that a postfix operator is meant.

Therefore, declaration of a function oveloading *postfix* incrementation operator would have the form

```
Typ operator++(int);
```

As usually, when overloading prefix and postfix incrementation or decrementation operators, one should remember about the principle of least astonishment. Operators in *postfix* forms should return *unmodified* values of their argument, and modification of arguments should be a side effect (having something to do with an incrementation or decrementation). The values returned should *not* be l-values. Usually, a copy of *this* object is created, then *this* object is modified, and finally the copy is returned by value.

For *prefix* forms, the value returned should be equal to already *modified* value of argument and *should be* an l-value. Therefore, methods overloading such operators usually return a reference to *this* object by executing `return *this` at the end.

The '-' and '+' operators have the binary and unary forms; this, however, does not pose any problem, because both forms can easily be distinguished by the number of parameters: binary '+' overloaded with a method will have one parameter, while the unary version will be a parameterless method.

In the example below, we overload all operators “with -”: binary and unary, the latter in both postfix and prefix forms:

P155: minus.cpp Overloading “minus” operators

```
1. #include <iostream>
2. using namespace std;
3.
4. class A {
5.     int data;
6. public:
7.     A(int data = 0) : data(2*(data/2)) { }
8.
9.     const A operator-() const {
10.         return A(-data);
11.     }
12.
13.     const A operator-(const A& a) const {
14.         return A(data - a.data);
15.     }
16.
17.     A& operator--() {
18.         ----data;
```

```
19.         return *this;
20.     }
21.
22.     const A operator--(int) {
23.         A x(data);
24.         ----data;
25.         return x;
26.     }
27.
28.     friend ostream& operator<<(ostream&,A);
29. };
30.
31. ostream& operator<<(ostream& strum, A d) {
32.     return strum << d.data;
33. }
34.
35. int main() {
36.     A data(7);
37.
38.     cout << "a.   data    = " <<   data   << endl;
39.     cout << "b.   data--  = " <<   data-- << endl;
40.     cout << "c.   data    = " <<   data   << endl;
41.     cout << "d. --data    = " << --data   << endl;
42.     cout << "e.   data    = " <<   data   << endl;
43.     cout << "f.  -data    = " <<  -data   << endl;
44.     cout << "g.   data    = " <<   data   << endl;
45. }
```

Class **A** has only one field ***data*** of type **int**. A number stored in this member will always be even: this is ensured by the only constructor of the class and by the fact that decrementing is implemented in such a way that the value of ***data*** is deceremented always by two. We can see the following overloadingsL:

- unary '-' operator, i.e., sign change operator (lines 9-11). A new object is returned by value; the original, *this* object, is not modified, so the method have been declared as **const**;
- binary '-' operator, i.e., subtracting operator (lines 13-15). It returns by value a new object, representing the difference between *this* object and the one passed by argument (i.e., the right hand side operand of the operator). The original, *this* object, is not modified, so again the method is declared as **const**;
- unary predecrementation operator ' –' (lines 17-20). The method decrements the member ***data*** by 2 and returns a reference to *this* object. Therefore, the value returned is an l-value and is equal to the value of the object *after* modification;
- unary postdecrementation operator (lines 22-26). A spurious argument of type **int** has been added. We do not use it, so it is not even given a name; its only purpose is to inform the compiler that we are oveloading *post*decrementation, and not *pre*decrementation. The methods produces a copy of *this* object, then modifies the member ***data***, and then it returns by value the *copy* which has been made before modification. The value returned is *not* an l-value.

We also overload the '<<' operator with a friend global function defined on lines 31-33. The printout

```
a.   data    = 6
b.   data--  = 6
c.   data    = 4
d. --data    = 2
e.   data    = 2
f.  -data    = -2
g.   data    = 2
```

shows that everything works as expected.

Let us consider one more example:

P156: tabinc.cpp Overloading incrementation operator

```
1. #include <iostream>
2. #include <cstring>  // memcpy
3. using namespace std;
4.
5. class Tablica {
6.     int size;
7.     int* tab;
8. public:
9.     Tablica(int size, const int* t)
10.         : size(size),
11.           tab((int*)memcpy(new int[size], t,
12.                            size*sizeof(int)))
13.     { }
14.
15.     Tablica(const Tablica& t)
16.         : size(t.size),
17.           tab((int*)memcpy(new int[size], t.tab,
18.                            size*sizeof(int)))
19.     { }
20.
21.     ~Tablica() { delete [] tab; }
22.
23.     Tablica& operator++();
24.     Tablica operator++(int);
25.     void showTab(const char* nap);
26. };
27.
28. Tablica& Tablica::operator++() {
29.     for (int i = 0; i < size; ++i)
30.         ++tab[i];
31.     return *this;
32. }
33.
34. Tablica Tablica::operator++(int) {
35.     Tablica t(*this);
36.     ++*this;
37.     return t;
38. }
39.
40. void Tablica::showTab(const char* nap) {
41.     cout << nap;
42.     for (int i = 0; i < size; i++)
43.         cout << tab[i] << " ";
44.     cout << endl;
45. }
46.
47. int main() {
48.     int tab[] = {1,2,3,4};
49.
50.     Tablica T(4,tab);
51.     T.showTab("Tablica wyjsciowa T: ");
52.     Tablica t = ++T;
53.     t.showTab("  Po t = ++T t jest: ");
54.     T.showTab("           a T jest: ");
55.
56.     Tablica S(4,tab);
57.     S.showTab("Tablica wyjsciowa S: ");
58.     Tablica s = S++;
59.     s.showTab("  Po s = S++ s jest: ");
60.     S.showTab("           a S jest: ");
61. }
```

Class **Array** has a pointer field pointing to a dynamically allocated array of **int** s. Therefore,

we need a destructor to deallocate memory occupied by the array when an object is deleted. Every object holds information on the size of array pointed to by ***arr*** in member ***size***.

Note how arrays are copied in constructors (lines 11 and 17). We copy the whole region of memory, instead of copying elements of the array one by one in a loop. This method is much more efficient for simple types, but *can fail* if elements of the array are of an object type!

Note that postfix incrementation operator, defined in lines 34-38, returns an object which is identical to *this* object *before* modification. Incrementation is a side effect which will be visible when the object is accessed next time. In the definition of postincrementation (line 36), we used the *prefix* incrementation operator, defined earlier, on lines 28-32. On line 35, using the copy-constructor, we create a copy of *this* object, and then invoke preincrementation for ***this** what modifies the object that the method has been invoked for; afterwards we return (line 37) the unmodified copy.

The program prints:

```
Original array T is: 1 2 3 4
 after t = ++T t is: 2 3 4 5
           and T is: 2 3 4 5
Original array S is: 1 2 3 4
 after s = ++S s is: 1 2 3 4
           and S is: 2 3 4 5
```

The class has a pointer field, so normally should be equipped with a custom assignment operator which would perform *deep* assignment, i.e., applying not to the member pointer, but rather the data it points to. Overloading ' =' operator will be discussed in the next subsection.

Prev: 19.2 Overloading with global functions **Up:** 19. Operator overloading. Move semantics **Next:** 19.4 “Special” operators

Subsections

- 19.4.1 Assignment operator
- 19.4.2 Indexing operator
- 19.4.3 Function call operator
- 19.4.4 Overloading indirect member selection operator

19.4 “Special” operators

The '=', '->', '()' and '[]' operators are in a way special, so we will discuss them in separate subsections. They all can be overloaded only by defining methods, never with global functions.

19.4.1 Assignment operator

Assignment operator '=' should be overloaded almost always when there are pointer fields in a class (as we already know, also destructors and copy-constructors are usually needed in such situation). If it is not overloaded, the system will provide a default implementation of assignment, which will assign field by field of the two objects: this is often not what we would want to happen for pointer members. We would rather want to assign another value to data which is pointed to by a pointer, not the value of the pointer itself.

No default assignment operator will be generated for classes containing const fields, fields of a reference type, or fields which are objects of classes with private assignment operator, or for which, for the same reasons, assignment operator is not defined at all.

In such situations, if we really need assignments, we not only can but have to overload assignment opeartor ourselves.

Let us see what can go wrong if a class with a pointer field does not overload assignment operator properly:

P157: ovrlderr.cpp Lack of proper assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
```

```
 8.     A(const char* s)
 9.         : name(strcpy(new char[strlen(s)+1],s)) {
10.         cerr << "  ctor: " << (void*)name << endl;
11.     }
12.
13.     A(const A& k)
14.         : name(strcpy(new char[strlen(k.name)+1],k.name)) {
15.         cerr << "cpctor: " << (void*)name << endl;
16.     }
17.
18.     ~A() {
19.         cerr << "  dtor: "  << (void*)name << endl;
20.         delete [] name;
21.     }
22. };
23.
24. A ob1("ob1");
25.
26. int main() {
27.     cerr << "MAIN" << endl;
28.     A ob2(ob1);
29.     A ob3 = ob2; // copy-ctor
30.
31.     ob1 = ob3;
32.
33.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
34.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
35.
36.     cerr << "THE END" << endl;
37. }
```

There is a constructor, defined on lines 8-11, which fabricates a new object given a C-string: it allocates new region of memory to store the C-string and correctly makes a copy of the string (line 9) — this is similar to what we know from program `person3.cpp`. All this can be placed in the initialization list (see section on initialization lists). The expression

```
new char[strlen(s)+1]
```

allocates enough memory for the C-string ***s*** including terminating '\0' character. Then **strcpy** copies the string from ***s*** to the newly allocated segment of memory, and returns its address — this address is then used to initialize the member ***name*** of the object being constructed (functions **strlen** and **strcpy** have been described in chapter on strings).

The copy-constructor (lines 13-16) works in a similar way (see also section on copy-constructors).

Lines 18-21 define destructor, which releases memeory allocated in constructors. It seems that everything should work correctly; however, running the program reveals some problems:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
  ob1.name: 0x804b028
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b028
```

After returning from the **main** dstructor is called three times: this is not a surprise, as three objects have been created. However, two of them release the same segment of memory (starting at address 0x804b028). On the other hand, memory allocated for ***name*** belonging originally to object ***ob1*** (at address 0x804b008) has not been released at all!

One can see why it happened. During assignment from line 31, the content of object ***ob3*** was copied to ***ob1***. In particular, the address in member ***name*** of ***ob3***, allocated during creation of this object by the copy-constructor (line 29), was copied to the member ***name*** of ***ob1*** erasing what was there before, i.e., the address 0x804b008. This address is therefore completely lost and memory at this location will never be released. After the assignment, the two objects, ***ob1*** and ***ob3*** are distinct objects, but hold in ***name*** the address of the same region of memory (0x804b028 in our example).

It is now clear that when objects are destroyed after leaving **main**, destructor will try to release the same memory twice: first when destroying ***ob3*** and then when destroying ***ob1***. This is illegal and can crash the program — even if it does not, the state of the program becomes unpredictable.

Hence, we have to redefine assignment operation for objects of our class. We have to do it by overloading a method: asignment operator cannot be overloaded with global functions. The method should sensibly assign data corresponding to the object on the right-hand side of ' =' operator (passed as an argument) to data belonging to the object on the left-hand side (which will be *this* object). Arrays and objects pointed to by pointer members of *this* object have to be first released and then reallocated and populated with data from corresponding arrays or objects pointed to by pointer members of the other object. As an assignment `a=a` should always be legal, we have to take care not to release memory holding data which still will be needed! Moreover, for `a=b=c` to be valid, the operator (method) should return a reference to *this* object (' `return *this`'), so its value can be used in next assignment (`b=c` returns reference to ***b***, and the value of ***b*** *after* this assignment can then be assigned to ***a***). The argument (right operand of the operator) is usually passed by reference, not by value, to save on unnecessary copying. Therefore, the prototype of a method overloading assignment opearator for a class **A** will be

```
A& operator=(const A&);
```

Let us see all this in practice:

P158: ovrldeq.cpp Overloading assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
8.     A() : name(new char[1]) {
9.         cerr << "dfctor: " << (void*)name << endl;
10.        name[0] = '\0';
11.    }
12.
13.    A(const char* s)
14.        : name(strcpy(new char[strlen(s)+1],s)) {
15.        cerr << "  ctor: " << (void*)name << endl;
16.    }
17.
18.    A(const A& k)
19.        : name(strcpy(new char[strlen(k.name)+1],k.name)) {
20.        cerr << "cpctor: " << (void*)name << endl;
21.    }
22.
23.    A& operator=(const A& k) {
24.
25.        if (this == &k) return *this;
26.
27.        cerr << "delete: " << (void*)name << endl;
```

```
28.         delete [] name;
29.         name = strcpy(new char[strlen(k.name)+1],k.name);
30.         cerr << "   op=: " << (void*)name << endl;
31.         return *this;
32.     }
33.
34.     ~A() {
35.         cerr << "  dtor: "  << (void*)name << endl;
36.         delete [] name;
37.     }
38. };
39.
40. A ob1("ob1");
41.
42. int main() {
43.     cerr << "MAIN" << endl;
44.     A ob2(ob1);
45.     A ob3 = ob2; // copy-ctor
46.
47.     ob1 = ob3;
48.
49.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
50.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
51.
52.     cerr << "THE END" << endl;
53. }
```

Class **A** is almost identical to the one from the previous program. On lines 8-11, we have added a default (parameterless) constructor to make the class more complete. Note that even allocating an empty C-string, which takes one byte for '\0' character, we use the array form of **new** operator (with square brackets), although in principle we could have allocated a single **char** without using square brackets. However, the array form is necessary here, because the destructor always uses array form of **delete**.

We define a method overloading the assignment operator on lines 23-32. Its structure is typical for this kind of methods:

- it returns by reference the object for which it was invoked (`return *this`);
- it first checks (line 25), if *this* object is not the same as the object passed by argument. This can be done by comparing addresses of the two object: one is the value of **this** and the second is `&k`, the address of the variable passed by reference to the method. Equality of addresses means that the two objects are in fact the same object, so there is nothing to do and the methods returns;
- knowing that this is not the '`a=a`' case, we can delete the array (C-string) belonging to *this* object (line 28);
- we allocate a new region of memory for the array and we copy the contents of the corresponding array from the object `k`. Note that in '`a=a`' case, the source array would not exist after deleteing the array from *this* object, because these two arrays would be in fact the same array;
- we always return ***this***.

The main program itself does not differ from the previous one, but the results *do* differ:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
delete: 0x804b008
   op=: 0x804b008
  ob1.name: 0x804b008
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
```

```
dtor: 0x804b018
dtor: 0x804b008
```

When the program is executing the assignment from line 47 ('`ob1=ob3`'), memory for the C-string belonging to object ***ob1*** (under address 0x804b008) is first released, and then a new segment is allocated — in our example, the system allocated in fact the same region under address 0x804b008, but this does not matter: equally well it could have been any other segment. What is important is the fact that now every region allocated by **new[]** is eventually released by a corresponding **delete[]** and the destructors call **delete[]** for different addresses.

Let us recall here that an expression

```
A a = b;
```

does *not* denote an assignment: it is just another form of creating a new object using the copy-constructor — it is essentially equivalent to '`A a(b)`'. Symbol '`=`' denotes an assignment only when an l-value designating an *existing* object appears on the left-hand side of the operator.

Another example of overloading assignment operator can be found in the following program:

P159: ope.cpp An array-like class with assignment operator

```
1. #include <iostream>
2. #include <cstring>   // memcpy
3. using namespace std;
4.
5. class ArrInt {
6.     static int ID;
7.     int        id;
8.     int      size;
9.     int      *arr;
10. public:
11.     ArrInt(const int *t, int size)
12.         : id(++ID), size(size),
13.           arr((int*)memcpy(new int[size], t,
14.                            size*sizeof(int))) {
15.         cout << "      ctor: id = " << id << endl;
16.     }
17.
18.     ArrInt(const ArrInt& t)
19.         : id(++ID), size(t.size),
20.           arr((int*)memcpy(new int[size], t.arr,
21.                            size*sizeof(int))) {
22.         cout << " copy-ctor: id = " << t.id
23.              << "-->" << id << endl;
24.     }
25.
26.     ~ArrInt() {
27.         cout << "  Deleting: id = " << id << endl;
28.         delete [] arr;
29.     }
30.
31.     ArrInt& operator=(const ArrInt& t) {
32.         cout << "Assignment: id = " << id
33.              << "<--" << t.id << endl;
34.         if (this != &t) {
35.             delete [] arr;
36.             size = t.size;
37.             arr = (int*)memcpy(new int[size], t.arr,
38.                                size*sizeof(int));
39.         }
40.         return *this;
41.     }
42. };
```

```
43. int ArrInt::ID = 0;
44.
45. int main() {
46.     int arr[] = {1,2,3};
47.     int size  = sizeof(arr)/sizeof(int);
48.
49.     ArrInt *pt1 = new ArrInt(arr,size);
50.     ArrInt   t2 = *pt1;
51.     ArrInt   t3(t2);
52.
53.     *pt1 = t2;
54. }
```

Note that we have used **memcpy** instead of **strcpy**. It copies segments of memory without copying element by element in a loop and is very efficient. From the output

```
      ctor: id = 1
 copy-ctor: id = 1-->2
 copy-ctor: id = 2-->3
Assignment: id = 1<--2
  Deleting: id = 3
  Deleting: id = 2
```

we can see that the first element (created on line 49) has not been deleted. This is because it was created on the heap (with **new** operator) and not explicitely removed; the other objects were created on the stack, so they are deleted (and their destructor is called) automatically when the flow of control leaves the **main**.

19.4.2 Indexing operator

The indexing operator '`[]`' is a binary operator: in an expression '`a[i]`' the name ***a*** denotes the first argument (operand) and ***i*** the second one. A method overloading this operator must be a one-parameter method — it will be called for ***a*** with ***i*** passed as an argument. The "index" ***i*** does *not* have to be of an integer type; also the return type is arbitrary.

The prototype of a method overloading '`[]`' operator has the form

```
Type operator[](Type_arg);
```

with **Type** and **Type_arg** being return type and argument type, respectively. The method is invoked for object ***a*** when an expression

```
a[i]
```

is encounterd and ***a*** is the name of an object (or a reference to an object) of the class where the operator '`[]`' has been overloaded; of course the type of ***i*** must correspond to the declared type of the parameter of the method. The invocation will have the form

```
a.operator[](i)
```

Of course, the method may do whatever we want it to do, but it is reasonable if it resembles some sort of "indexing". If this is the rôle of the method, then, not to surprise the user, it should return an l-value (by reference), because normally indexing can appear both on the right and on the left-hand side of an assignment:

```
x = arr[i];
arr[j] = y;
```

A simple class **Letter** in the following program illustrates overloading of indexing operator, but also copy-constructor, destructor and oveloading of assignment — three elements ususally needed when a class contains a pointer field.

P160: letter.cpp Overloading indexing operator

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Letter {
6.     char* name;
7. public:
8.     Letter(const Letter& k)
9.         : name(strcpy(new char[strlen(k.name)+1],
10.                       k.name))
11.     { }
12.
13.     Letter(const char* name)
14.         : name(strcpy(new char[strlen(name)+1],
15.                       name))
16.     { }
17.
18.     Letter& operator=(const Letter&);
19.     char& operator[](int);
20.
21.     ~Letter() { delete [] name; }
22.
23.     friend ostream& operator<<(ostream&,const Letter&);
24. };
25.
26. char& Letter::operator[](int n) {
27.     int len = strlen(name);
28.     if ( n < 0 || n >= len )
29.         // reference to NUL if index is wrong
30.         return name[len];
31.     else
32.         return name[n];
33. }
34.
35. Letter& Letter::operator=(const Letter& k) {
36.     if (this == &k ) return *this;
37.     delete [] name;
38.     name = strcpy(new char[strlen(k.name)+1],k.name);
39.     return *this;
40. }
41.
42. ostream& operator<<(ostream& str, const Letter& k) {
43.     return str << k.name;
44. }
45.
46. int main() {
47.     Letter a("Benny");
48.     cout << "a=" << a << endl;
49.
50.     char c = a[100];
51.     if (c == '\0') cout << "Out of range!" << endl;
52.     else           cout << "character: "   << c << endl;
53.
54.     c = a[4];
55.     if (c == '\0') cout << "Out of range!" << endl;
56.     else           cout << "character: "   << c << endl;
57.
58.     a[0] = 'J';
59.     cout << "a=" << a << endl;
60. }
```

As we can see from lines 26-33, indexing returns a reference to the ***n***-th character of a C-

string ***name***. Defining such a method makes sense, because it can check if the value of an index is valid and, if not, return a reference to ' \0' character rather than some random value, as it would be the case for “normal” arrays. The printout

```
a=Benny
Out of range!
character: y
a=Jenny
```

shows that indeed the indexed name of an object of the class can be treated as the name of an array: it can appear on both sides of assignments.

However, the “index” does not have to be an *int*:

P161: ranges.cpp Floating point “index”

```
1. #include <iostream>
2. using namespace std;
3.
4. class Ranges {
5.     int    cntr[3];
6.     double min, max;
7. public:
8.     Ranges(double min, double max)
9.         : min(min), max(max) {
10.        cntr[0]=cntr[1]=cntr[2]=0;
11.    }
12.
13.    int& operator[](double x) {
14.        int i;
15.
16.        if      ( x >  max ) i = 2;
17.        else if ( x >= min ) i = 1;
18.        else                 i = 0;
19.
20.        return cntr[i];
21.    }
22.
23.    friend ostream& operator<<(ostream&, const Ranges&);
24. };
25.
26. ostream& operator<<(ostream& str, const Ranges& r) {
27.    return str << "Range = [" << r.min << ", " << r.max
28.               << "]:  " << "Below " << r.cntr[0]
29.               << "; inside " << r.cntr[1] << "; above "
30.               << r.cntr[2];
31. }
32.
33. int main() {
34.    Ranges range(3.0, 5.5);
35.    double x;
36.    cout << "Enter numbers; 0 terminates:" << endl;
37.
38.    while ( ( cin >> x) && x ) range[x]++;
39.
40.    cout << range << endl;
41.    cout << "x = 4.7 -> cntr = " << range[4.7] << endl;
42. }
```

The class **Ranges** defines fields ***min***, ***max*** and a three-element array of **int** s whose elements play the rôle of counters for data below ***min***, in the range [***min***, ***max***], and above ***max***, respectively. Indexing operator is overloaded with a method whose name is **operator[]** and which takes one argument of type **double**. If ***range*** is an object of class **Ranges** and ***x*** is of type **double**, expression `range[x]` is a reference to an element of

member array ***cntr***: `cntr[0]`, if the value of ***x*** is below ***min***, `cntr[1]`, it it is in the range [***min***, ***max***], and `licz[2]` if it is above ***max***. Therefore, expression on line 38 (`zakres[x]++`) increments an appropriate element of ***cntr*** according to the value of ***x***. Similarily, value of `zakres[4.7]` from line 41 is the value of this element of the array which corresponds to the value of the argument (4.7 lies in the range [3,5.5], so in our case it will be the current value of counter ***cntr[1]***). Th eoutput of the program

```
Enter numbers; 0 terminates:
8 7 6 5 4 6.8 9.4 1.2
3 4 5 1 6   3   2   7  7.9  3.1  0
Range = [3, 5.5]:  Below 3; inside 7; above 8
x = 4.7 -> cntr = 7
```

shows the effect of such overloading in action. On line 38 we used a logical conjuction in the condition part of a **while** loop. The loop terminates if either the stream ***cin*** has become *bad* (because an error ocurred or end of file had been reached) or number zero has been read.

In this example, indexing operator returns reference to a private member of the object: generally, such practice cannot be recommended, as it completely breaks privacy of a class.

19.4.3 Function call operator

Function call operator '`()`' is the only one which can take an arbitraty number of arguments. It can be overloaded only with a method, whose name is therefore **operator()** — here parentheses are part of the name, which in turn is followed by a list, perhaps empty, of parameters enclosed in another pair of parentheses. Therefore, the declaration of such a method looks like this:

```
Type operator()(Type_arg1, Type_arg2, Type_arg3);
```

where the number of parameters is arbitrary. The method will be called when an expression like

```
obj(a,b,c)
```

is encountered, where ***obj*** is the name of an object of a class with overloaded function call operator and types of ***a***, ***b*** and ***c*** match the types of parameters of the overloading method. The call will be equivalent to

```
obj.operator()(a,b,c)
```

Note that the form `obj(a,b,c)` looks like a “normal” function invocation, but here ***obj*** in the name of an *object*, not of a function. Such objects, which “can be called”, are often referred to as **function objects** or **callable objects**.

Invocation of such a method does not have to return an l-value — exactly as it is for “normal” functions, which most often return their results by value. However, we are not limited to such methods — equally well an overloading method may return a reference (which *is* an l-value).

The example below shows how to use overloading of the function call operator to get objects which behave as three-dimensional, dynamically allocated matrices (i.e., with dimensions which can be calculated or read in during execution). The method **operator()**

returns reference to a number of type **int** which is an element of the matrix represented by object. A call with three arguments plays the rôle of triple indexing of a matrix:

P162: arr3dim.cpp Overloading function call operator

```
1. #include <iostream>
2. #include <cstring>
3. #include <iomanip>
4. using namespace std;
5.
6. class Arr3D {
7.     int  dim1, dim2, dim3;
8.     int* arr;
9. public:
10.     Arr3D() : dim1(1),dim2(1),dim3(1),
11.               arr(new int[1])
12.     { }
13.
14.     Arr3D(int dim1, int dim2, int dim3)
15.         : dim1(dim1), dim2(dim2), dim3(dim3),
16.           arr(new int[dim1*dim2*dim3])
17.     { }
18.
19.     Arr3D(const Arr3D& t)
20.         : dim1(t.dim1), dim2(t.dim2), dim3(t.dim3),
21.           arr((int*)memcpy(new int[dim1*dim2*dim3],
22.                            t.arr,
23.                            dim1*dim2*dim3*sizeof(int)))
24.     { }
25.
26.     ~Arr3D() { delete [] arr; }
27.
28.     Arr3D& operator=(const Arr3D&);
29.     int& operator()(int,int,int);
30. };
31.
32. int& Arr3D::operator()(int n1, int n2, int n3)  {
33.     return *(arr + n1*dim2*dim3 + n2*dim3 + n3);
34. }
35.
36. Arr3D& Arr3D::operator=(const Arr3D& t) {
37.     if ( this != &t ) {
38.         delete [] arr;
39.         dim1 = t.dim1;
40.         dim2 = t.dim2;
41.         dim3 = t.dim3;
42.         arr  = (int*)memcpy(new int[dim1*dim2*dim3],
43.                             t.arr,
44.                             dim1*dim2*dim3*sizeof(int));
45.     }
46.     return *this;
47. }
48.
49. int main() {
50.     int dim1 = 1000, dim2 = 1000, dim3 = 50;
51.
52.     Arr3D T1(dim1,dim2,dim3), T2;
53.
54.     for (int i = 0; i < dim1; i++)
55.         for (int j = 0; j < dim2; j++)
56.             for (int k = 0; k < dim3; k++)
57.                 T1(i,j,k) = i+j+k;
58.     T2 = T1;
59.
60.     cout << "T2(999, 999 , 2) = "
61.          << setw(4) << T2(999,999,2) << endl;
62.
63.     cout << "T2(  0,    0, 9) = "
64.          << setw(4) << T2(  0,  0,9) << endl;
65. }
```

Note that we have defined a copy-constructor, destructor and assignment operator overloading: this is normal for classes with pointer fields. Objects of class **Arr3D** represent 3-dimensional matrices (with three indices). Actually, the class member ***arr*** is a pointer to a normal *one*-dimensional array of size equal to the product of dimensions ***dim1***, ***dim2*** and ***dim3***. However, the only way the array can be accessed is through the **operator()** method (lines 32-34), i.e., with the help of expressions like `T(i,j,k)`, where ***T*** is the name of an object of class **Arr3D**. The method treats the three arguments as matrix indices and calculates which element of the one-dimensional array would correspond to element of 3-dimensional array with these indices (note that the first dimension, ***dim1***, is not needed for that). The method returns reference to this element, so `T(i,j,k)` behaves like `T[i][j][k]` would behave for a "normal" 3-dimensional array. In particular, it is an l-value and can appear on the left-hand side of assignments (as on line 57).

We create two objects of class **Arr3D**. The first, ***T1***, corresponds to a 3-dimensional array with dimensions $1000\times1000\times50$; the second, ***T2***, is created by the default constructor and has dimensions $1\times1\times1$ (line 52). The array in ***T1*** is represented by one-dimensional array of 50 million elements; it is a private member, so the user does not have to know it. We asign values to elements in lines 54-57 — for the purpose of this example these values are equal to the sum of indices of each element. On line 58 we make an assignment of objects of our class. We then print two values

```
T2( 999, 999 , 2) = 2000
T2(   0,    0, 9) =    9
```

to check the correctness of assignment and function call overloading. One can see that indeed `T(i,j,k)` can be treated as elements of a 3-dimensional array.

Operating on large matrices often requires careful considerations and experience; for example, changing the order of loops in lines 54-56 would yield equivalent program, but on most machines it would be 3-6 times slower.

19.4.4 Overloading indirect member selection operator

Operator of indirect member selection ('`->`') can be overloaded as a *one*-argument (unary) operator. Therefore, one overloads it with a parameterless method. It will be called for an object which appears on the left-hand side of the operator, without passing any arguments. Note that to the left of '`->`' operator we have an *object*, not a pointer! Let us recall here that the direct member selection operator, "dot" operator, cannot be overloaded at all).

Any method overloading the '`->`' oparator must return a pointer value (an address), which is then used as the left operand of "normal" '`->`' operator. It is, of course, possible, that what will be returned will be again an object of a class with overloaded `->` operator, which in turn returns a pointer (or again an object...).

Suppose ***obj*** is the name of an object of a class with overloaded '`->`' operator. Then an expression

```
obj->b
```

is equivalent to

```
temp = obj.operator->(), temp->b
```

In other words:

- method **operator->** is invoked for the object ***obj***;
- a value returned, denoted here by ***temp***, must be of a pointer type, i.e., is the address of an object of a class (not necessarily the same as that of object ***obj***!);
- from the object pointed to by ***temp***, a member ***b*** is selected (a member of that name must, of course, exist in the class of ****temp***). The value of this member will be the value of the whole expression.

Let us look at the following example. Variable ***AB*** is an object of class **Segment**, while `AB->x` will return x-coordinate of one of the two endpoints — objects of class **Point** — which define the segment. The return type of **operator->** method in class **Segment** is *pointer to constant object of class **Point*** (and *not* of class **Segment**):

P163: ovrlskl.cpp Overloading indirect member selection operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      int x, y;
6.
7.      Point(int x = 0, int y = 0) : x(x), y(y)
8.      { }
9.
10.     double r2() const { return x*x + y*y; }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.
16.     Segment(Point A = Point(), Point B = Point())
17.         : A(A), B(B)
18.     { }
19.
20.     const Point* operator->() const {
21.         return (A.r2() < B.r2()) ? &A : &B;
22.     }
23. };
24.
25. ostream& operator<<(ostream& str, const Point& A) {
26.     return str << "P[" << A.x << "," << A.y << "]";
27. }
28.
29. ostream& operator<<(ostream& str, const Segment& AB) {
30.     return str << AB.A << "--" << AB.B;
31. }
32.
33. int main() {
34.
35.     Point    A(1,0),  B(8,6),  C(4,3);
36.     Segment AB(A,B), BC(B,C), CA(C,A);
37.
38.     cout << "AB = " << AB << ": AB->x = "
39.                    << AB->x << endl;
40.
41.     cout << "BC = " << BC << ": BC->y = "
42.                    << BC->y << endl;
43.
44.     cout << "CA = " << CA << ": CA->x = "
45.                    << CA->x << endl;
46. }
```

The overloading method returns the pointer to (address of) the endpoint which lies closer to the origin of the coordinate system (line 21). From the object (point) pointed to by this pointer, member ***x*** or ***y*** is then selected by “ordinary” ' `->`' operator, as one can see on

lines 38-45, which print

```
AB = P[1,0]--P[8,6]: AB->x = 1
BC = P[8,6]--P[4,3]: BC->y = 3
CA = P[4,3]--P[1,0]: CA->x = 1
```

For example, ***BC*** on lines 41-42 is the name of an object of class **Segment** representing a segment with endpoints corresponding to objects ***B*** and ***C*** of class **Point**. In class **Segment**, the '`->`' operator is overloaded (lines 20-22) and returns the address of one of points which are members of the class — the one closer to the origin, in our case it is point $[4, 3]$. Then, from this object, the member ***y*** is selected. Note that in the class of ***BC***, which is class **Segment**, there is *no* field named ***y***. Additionally, ***BC*** is the name of an object, not of a pointer, so normally one would have to use dot operator instead of the “arrow”.

Prev: 19.3 Overloading with methods of **Up:** 19. Operator overloading. Move semantics
Next: 19.5 Move semantics

T.R. Werner, September 27, 2023; 22:32

Subsections

- 19.4.1 Assignment operator
- 19.4.2 Indexing operator
- 19.4.3 Function call operator
- 19.4.4 Overloading indirect member selection operator

19.4 “Special” operators

The ' =', ' ->', ' ()' and ' []' operators are in a way special, so we will discuss them in separate subsections. They all can be overloaded only by defining methods, never with global functions.

19.4.1 Assignment operator

Assignment operator ' =' should be overloaded almost always when there are pointer fields in a class (as we already know, also destructors and copy-constructors are usually needed in such situation). If it is not overloaded, the system will provide a default implementation of assignment, which will assign field by field of the two objects: this is often not what we would want to happen for pointer members. We would rather want to assign another value to data which is pointed to by a pointer, not the value of the pointer itself.

No default assignment operator will be generated for classes containing const fields, fields of a reference type, or fields which are objects of classes with private assignment operator, or for which, for the same reasons, assignment operator is not defined at all.

In such situations, if we really need assignments, we not only can but have to overload assignment opeartor ourselves.

Let us see what can go wrong if a class with a pointer field does not overload assignment operator properly:

P157: ovrlderr.cpp Lack of proper assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
```

```
 8.     A(const char* s)
 9.         : name(strcpy(new char[strlen(s)+1],s)) {
10.         cerr << "  ctor: " << (void*)name << endl;
11.     }
12.
13.     A(const A& k)
14.         : name(strcpy(new char[strlen(k.name)+1],k.name)) {
15.         cerr << "cpctor: " << (void*)name << endl;
16.     }
17.
18.     ~A() {
19.         cerr << "  dtor: "  << (void*)name << endl;
20.         delete [] name;
21.     }
22. };
23.
24. A ob1("ob1");
25.
26. int main() {
27.     cerr << "MAIN" << endl;
28.     A ob2(ob1);
29.     A ob3 = ob2; // copy-ctor
30.
31.     ob1 = ob3;
32.
33.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
34.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
35.
36.     cerr << "THE END" << endl;
37. }
```

There is a constructor, defined on lines 8-11, which fabricates a new object given a C-string: it allocates new region of memory to store the C-string and correctly makes a copy of the string (line 9) — this is similar to what we know from program `person3.cpp`. All this can be placed in the initialization list (see section on initialization lists). The expression

```
new char[strlen(s)+1]
```

allocates enough memory for the C-string ***s*** including terminating '\0' character. Then **strcpy** copies the string from ***s*** to the newly allocated segment of memory, and returns its address — this address is then used to initialize the member ***name*** of the object being constructed (functions **strlen** and **strcpy** have been described in chapter on strings).

The copy-constructor (lines 13-16) works in a similar way (see also section on copy-constructors).

Lines 18-21 define destructor, which releases memeory allocated in constructors. It seems that everything should work correctly; however, running the program reveals some problems:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
  ob1.name: 0x804b028
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b028
```

After returning from the **main** dstructor is called three times: this is not a surprise, as three objects have been created. However, two of them release the same segment of memory (starting at address 0x804b028). On the other hand, memory allocated for ***name*** belonging originally to object ***ob1*** (at address 0x804b008) has not been released at all!

One can see why it happened. During assignment from line 31, the content of object ***ob3*** was copied to ***ob1***. In particular, the address in member ***name*** of ***ob3***, allocated during creation of this object by the copy-constructor (line 29), was copied to the member ***name*** of ***ob1*** erasing what was there before, i.e., the address 0x804b008. This address is therefore completely lost and memory at this location will never be released. After the assignment, the two objects, ***ob1*** and ***ob3*** are distinct objects, but hold in ***name*** the address of the same region of memory (0x804b028 in our example).

It is now clear that when objects are destroyed after leaving **main**, destructor will try to release the same memory twice: first when destroying ***ob3*** and then when destroying ***ob1***. This is illegal and can crash the program — even if it does not, the state of the program becomes unpredictable.

Hence, we have to redefine assignment operation for objects of our class. We have to do it by overloading a method: asignment operator cannot be overloaded with global functions. The method should sensibly assign data corresponding to the object on the right-hand side of ' =' operator (passed as an argument) to data belonging to the object on the left-hand side (which will be *this* object). Arrays and objects pointed to by pointer members of *this* object have to be first released and then reallocated and populated with data from corresponding arrays or objects pointed to by pointer members of the other object. As an assignment `a=a` should always be legal, we have to take care not to release memory holding data which still will be needed! Moreover, for `a=b=c` to be valid, the operator (method) should return a reference to *this* object (' `return *this`'), so its value can be used in next assignment (`b=c` returns reference to ***b***, and the value of ***b*** *after* this assignment can then be assigned to ***a***). The argument (right operand of the operator) is usually passed by reference, not by value, to save on unnecessary copying. Therefore, the prototype of a method overloading assignment opearator for a class **A** will be

```
A& operator=(const A&);
```

Let us see all this in practice:

P158: ovrldeq.cpp Overloading assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
8.     A() : name(new char[1]) {
9.         cerr << "dfctor: " << (void*)name << endl;
10.        name[0] = '\0';
11.    }
12.
13.    A(const char* s)
14.        : name(strcpy(new char[strlen(s)+1],s)) {
15.        cerr << "  ctor: " << (void*)name << endl;
16.    }
17.
18.    A(const A& k)
19.        : name(strcpy(new char[strlen(k.name)+1],k.name)) {
20.        cerr << "cpctor: " << (void*)name << endl;
21.    }
22.
23.    A& operator=(const A& k) {
24.
25.        if (this == &k) return *this;
26.
27.        cerr << "delete: " << (void*)name << endl;
```

```
28.         delete [] name;
29.         name = strcpy(new char[strlen(k.name)+1],k.name);
30.         cerr << "   op=: " << (void*)name << endl;
31.         return *this;
32.     }
33.
34.     ~A() {
35.         cerr << "  dtor: "  << (void*)name << endl;
36.         delete [] name;
37.     }
38. };
39.
40. A ob1("ob1");
41.
42. int main() {
43.     cerr << "MAIN" << endl;
44.     A ob2(ob1);
45.     A ob3 = ob2; // copy-ctor
46.
47.     ob1 = ob3;
48.
49.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
50.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
51.
52.     cerr << "THE END" << endl;
53. }
```

Class **A** is almost identical to the one from the previous program. On lines 8-11, we have added a default (parameterless) constructor to make the class more complete. Note that even allocating an empty C-string, which takes one byte for '\0' character, we use the array form of **new** operator (with square brackets), although in principle we could have allocated a single **char** without using square brackets. However, the array form is necessary here, because the destructor always uses array form of **delete**.

We define a method overloading the assignment operator on lines 23-32. Its structure is typical for this kind of methods:

- it returns by reference the object for which it was invoked (`return *this`);
- it first checks (line 25), if *this* object is not the same as the object passed by argument. This can be done by comparing addresses of the two object: one is the value of **this** and the second is `&k`, the address of the variable passed by reference to the method. Equality of addresses means that the two objects are in fact the same object, so there is nothing to do and the methods returns;
- knowing that this is not the '`a=a`' case, we can delete the array (C-string) belonging to *this* object (line 28);
- we allocate a new region of memory for the array and we copy the contents of the corresponding array from the object `k`. Note that in '`a=a`' case, the source array would not exist after deleteing the array from *this* object, because these two arrays would be in fact the same array;
- we always return ***this***.

The main program itself does not differ from the previous one, but the results *do* differ:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
delete: 0x804b008
   op=: 0x804b008
  ob1.name: 0x804b008
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
```

```
dtor: 0x804b018
dtor: 0x804b008
```

When the program is executing the assignment from line 47 ('`ob1=ob3`'), memory for the C-string belonging to object ***ob1*** (under address 0x804b008) is first released, and then a new segment is allocated — in our example, the system allocated in fact the same region under address 0x804b008, but this does not matter: equally well it could have been any other segment. What is important is the fact that now every region allocated by **new[]** is eventually released by a corresponding **delete[]** and the destructors call **delete[]** for different addresses.

Let us recall here that an expression

```
A a = b;
```

does *not* denote an assignment: it is just another form of creating a new object using the copy-constructor — it is essentially equivalent to '`A a(b)`'. Symbol '`=`' denotes an assignment only when an l-value designating an *existing* object appears on the left-hand side of the operator.

Another example of overloading assignment operator can be found in the following program:

P159: ope.cpp An array-like class with assignment operator

```
1. #include <iostream>
2. #include <cstring>   // memcpy
3. using namespace std;
4.
5. class ArrInt {
6.     static int ID;
7.     int        id;
8.     int      size;
9.     int      *arr;
10. public:
11.     ArrInt(const int *t, int size)
12.         : id(++ID), size(size),
13.           arr((int*)memcpy(new int[size], t,
14.                            size*sizeof(int))) {
15.         cout << "      ctor: id = " << id << endl;
16.     }
17.
18.     ArrInt(const ArrInt& t)
19.         : id(++ID), size(t.size),
20.           arr((int*)memcpy(new int[size], t.arr,
21.                            size*sizeof(int))) {
22.         cout << " copy-ctor: id = " << t.id
23.              << "-->" << id << endl;
24.     }
25.
26.     ~ArrInt() {
27.         cout << "  Deleting: id = " << id << endl;
28.         delete [] arr;
29.     }
30.
31.     ArrInt& operator=(const ArrInt& t) {
32.         cout << "Assignment: id = " << id
33.              << "<--" << t.id << endl;
34.         if (this != &t) {
35.             delete [] arr;
36.             size = t.size;
37.             arr = (int*)memcpy(new int[size], t.arr,
38.                                size*sizeof(int));
39.         }
40.         return *this;
41.     }
42. };
```

```
43. int ArrInt::ID = 0;
44.
45. int main() {
46.     int arr[] = {1,2,3};
47.     int size  = sizeof(arr)/sizeof(int);
48.
49.     ArrInt *pt1 = new ArrInt(arr,size);
50.     ArrInt   t2 = *pt1;
51.     ArrInt   t3(t2);
52.
53.     *pt1 = t2;
54. }
```

Note that we have used **memcpy** instead of **strcpy**. It copies segments of memory without copying element by element in a loop and is very efficient. From the output

```
      ctor: id = 1
 copy-ctor: id = 1-->2
 copy-ctor: id = 2-->3
Assignment: id = 1<--2
  Deleting: id = 3
  Deleting: id = 2
```

we can see that the first element (created on line 49) has not been deleted. This is because it was created on the heap (with **new** operator) and not explicitely removed; the other objects were created on the stack, so they are deleted (and their destructor is called) automatically when the flow of control leaves the **main**.

19.4.2 Indexing operator

The indexing operator '`[]`' is a binary operator: in an expression '`a[i]`' the name ***a*** denotes the first argument (operand) and ***i*** the second one. A method overloading this operator must be a one-parameter method — it will be called for ***a*** with ***i*** passed as an argument. The "index" ***i*** does *not* have to be of an integer type; also the return type is arbitrary.

The prototype of a method overloading '`[]`' operator has the form

```
Type operator[](Type_arg);
```

with **Type** and **Type_arg** being return type and argument type, respectively. The method is invoked for object ***a*** when an expression

```
a[i]
```

is encounterd and ***a*** is the name of an object (or a reference to an object) of the class where the operator '`[]`' has been overloaded; of course the type of ***i*** must correspond to the declared type of the parameter of the method. The invocation will have the form

```
a.operator[](i)
```

Of course, the method may do whatever we want it to do, but it is reasonable if it resembles some sort of "indexing". If this is the rôle of the method, then, not to surprise the user, it should return an l-value (by reference), because normally indexing can appear both on the right and on the left-hand side of an assignment:

```
x = arr[i];
arr[j] = y;
```

A simple class **Letter** in the following program illustrates overloading of indexing operator, but also copy-constructor, destructor and oveloading of assignment — three elements ususally needed when a class contains a pointer field.

P160: letter.cpp Overloading indexing operator

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Letter {
6.     char* name;
7. public:
8.     Letter(const Letter& k)
9.         : name(strcpy(new char[strlen(k.name)+1],
10.                       k.name))
11.     { }
12.
13.     Letter(const char* name)
14.         : name(strcpy(new char[strlen(name)+1],
15.                       name))
16.     { }
17.
18.     Letter& operator=(const Letter&);
19.     char& operator[](int);
20.
21.     ~Letter() { delete [] name; }
22.
23.     friend ostream& operator<<(ostream&,const Letter&);
24. };
25.
26. char& Letter::operator[](int n) {
27.     int len = strlen(name);
28.     if ( n < 0 || n >= len )
29.         // reference to NUL if index is wrong
30.         return name[len];
31.     else
32.         return name[n];
33. }
34.
35. Letter& Letter::operator=(const Letter& k) {
36.     if (this == &k ) return *this;
37.     delete [] name;
38.     name = strcpy(new char[strlen(k.name)+1],k.name);
39.     return *this;
40. }
41.
42. ostream& operator<<(ostream& str, const Letter& k) {
43.     return str << k.name;
44. }
45.
46. int main() {
47.     Letter a("Benny");
48.     cout << "a=" << a << endl;
49.
50.     char c = a[100];
51.     if (c == '\0') cout << "Out of range!" << endl;
52.     else           cout << "character: "   << c << endl;
53.
54.     c = a[4];
55.     if (c == '\0') cout << "Out of range!" << endl;
56.     else           cout << "character: "   << c << endl;
57.
58.     a[0] = 'J';
59.     cout << "a=" << a << endl;
60. }
```

As we can see from lines 26-33, indexing returns a reference to the ***n***-th character of a C-

string ***name***. Defining such a method makes sense, because it can check if the value of an index is valid and, if not, return a reference to ' \0' character rather than some random value, as it would be the case for “normal” arrays. The printout

```
a=Benny
Out of range!
character: y
a=Jenny
```

shows that indeed the indexed name of an object of the class can be treated as the name of an array: it can appear on both sides of assignments.

However, the “index” does not have to be an *int*:

P161: ranges.cpp Floating point “index”

```
1. #include <iostream>
2. using namespace std;
3.
4. class Ranges {
5.     int    cntr[3];
6.     double min, max;
7. public:
8.     Ranges(double min, double max)
9.         : min(min), max(max) {
10.        cntr[0]=cntr[1]=cntr[2]=0;
11.    }
12.
13.    int& operator[](double x) {
14.        int i;
15.
16.        if      ( x >  max ) i = 2;
17.        else if ( x >= min ) i = 1;
18.        else                 i = 0;
19.
20.        return cntr[i];
21.    }
22.
23.    friend ostream& operator<<(ostream&, const Ranges&);
24. };
25.
26. ostream& operator<<(ostream& str, const Ranges& r) {
27.    return str << "Range = [" << r.min << ", " << r.max
28.               << "]:  " << "Below " << r.cntr[0]
29.               << "; inside " << r.cntr[1] << "; above "
30.               << r.cntr[2];
31. }
32.
33. int main() {
34.    Ranges range(3.0, 5.5);
35.    double x;
36.    cout << "Enter numbers; 0 terminates:" << endl;
37.
38.    while ( ( cin >> x) && x ) range[x]++;
39.
40.    cout << range << endl;
41.    cout << "x = 4.7 -> cntr = " << range[4.7] << endl;
42. }
```

The class **Ranges** defines fields ***min***, ***max*** and a three-element array of **int** s whose elements play the rôle of counters for data below ***min***, in the range [***min***, ***max***], and above ***max***, respectively. Indexing operator is overloaded with a method whose name is **operator[]** and which takes one argument of type **double**. If ***range*** is an object of class **Ranges** and ***x*** is of type **double**, expression `range[x]` is a reference to an element of

member array ***cntr***: `cntr[0]`, if the value of ***x*** is below ***min***, `cntr[1]`, it it is in the range [***min***, ***max***], and `licz[2]` if it is above ***max***. Therefore, expression on line 38 (`zakres[x]++`) increments an appropriate element of ***cntr*** according to the value of ***x***. Similarily, value of `zakres[4.7]` from line 41 is the value of this element of the array which corresponds to the value of the argument (4.7 lies in the range [3,5.5], so in our case it will be the current value of counter ***cntr[1]***). Th eoutput of the program

```
Enter numbers; 0 terminates:
8 7 6 5 4 6.8 9.4 1.2
3 4 5 1 6   3   2   7  7.9  3.1  0
Range = [3, 5.5]:  Below 3; inside 7; above 8
x = 4.7 -> cntr = 7
```

shows the effect of such overloading in action. On line 38 we used a logical conjuction in the condition part of a **while** loop. The loop terminates if either the stream ***cin*** has become *bad* (because an error ocurred or end of file had been reached) or number zero has been read.

In this example, indexing operator returns reference to a private member of the object: generally, such practice cannot be recommended, as it completely breaks privacy of a class.

19.4.3 Function call operator

Function call operator '`()`' is the only one which can take an arbitraty number of arguments. It can be overloaded only with a method, whose name is therefore **operator()** — here parentheses are part of the name, which in turn is followed by a list, perhaps empty, of parameters enclosed in another pair of parentheses. Therefore, the declaration of such a method looks like this:

```
Type operator()(Type_arg1, Type_arg2, Type_arg3);
```

where the number of parameters is arbitrary. The method will be called when an expression like

```
obj(a,b,c)
```

is encountered, where ***obj*** is the name of an object of a class with overloaded function call operator and types of ***a***, ***b*** and ***c*** match the types of parameters of the overloading method. The call will be equivalent to

```
obj.operator()(a,b,c)
```

Note that the form `obj(a,b,c)` looks like a “normal” function invocation, but here ***obj*** in the name of an *object*, not of a function. Such objects, which “can be called”, are often referred to as **function objects** or **callable objects**.

Invocation of such a method does not have to return an l-value — exactly as it is for “normal” functions, which most often return their results by value. However, we are not limited to such methods — equally well an overloading method may return a reference (which *is* an l-value).

The example below shows how to use overloading of the function call operator to get objects which behave as three-dimensional, dynamically allocated matrices (i.e., with dimensions which can be calculated or read in during execution). The method **operator()**

returns reference to a number of type **int** which is an element of the matrix represented by object. A call with three arguments plays the rôle of triple indexing of a matrix:

P162: arr3dim.cpp Overloading function call operator

```
1. #include <iostream>
2. #include <cstring>
3. #include <iomanip>
4. using namespace std;
5.
6. class Arr3D {
7.     int  dim1, dim2, dim3;
8.     int* arr;
9. public:
10.     Arr3D() : dim1(1),dim2(1),dim3(1),
11.               arr(new int[1])
12.     { }
13.
14.     Arr3D(int dim1, int dim2, int dim3)
15.         : dim1(dim1), dim2(dim2), dim3(dim3),
16.           arr(new int[dim1*dim2*dim3])
17.     { }
18.
19.     Arr3D(const Arr3D& t)
20.         : dim1(t.dim1), dim2(t.dim2), dim3(t.dim3),
21.           arr((int*)memcpy(new int[dim1*dim2*dim3],
22.                            t.arr,
23.                            dim1*dim2*dim3*sizeof(int)))
24.     { }
25.
26.     ~Arr3D() { delete [] arr; }
27.
28.     Arr3D& operator=(const Arr3D&);
29.     int& operator()(int,int,int);
30. };
31.
32. int& Arr3D::operator()(int n1, int n2, int n3)  {
33.     return *(arr + n1*dim2*dim3 + n2*dim3 + n3);
34. }
35.
36. Arr3D& Arr3D::operator=(const Arr3D& t) {
37.     if ( this != &t ) {
38.         delete [] arr;
39.         dim1 = t.dim1;
40.         dim2 = t.dim2;
41.         dim3 = t.dim3;
42.         arr  = (int*)memcpy(new int[dim1*dim2*dim3],
43.                             t.arr,
44.                             dim1*dim2*dim3*sizeof(int));
45.     }
46.     return *this;
47. }
48.
49. int main() {
50.     int dim1 = 1000, dim2 = 1000, dim3 = 50;
51.
52.     Arr3D T1(dim1,dim2,dim3), T2;
53.
54.     for (int i = 0; i < dim1; i++)
55.         for (int j = 0; j < dim2; j++)
56.             for (int k = 0; k < dim3; k++)
57.                 T1(i,j,k) = i+j+k;
58.     T2 = T1;
59.
60.     cout << "T2(999, 999 , 2) = "
61.          << setw(4) << T2(999,999,2) << endl;
62.
63.     cout << "T2(  0,    0, 9) = "
64.          << setw(4) << T2(  0,  0,9) << endl;
65. }
```

Note that we have defined a copy-constructor, destructor and assignment operator overloading: this is normal for classes with pointer fields. Objects of class **Arr3D** represent 3-dimensional matrices (with three indices). Actually, the class member ***arr*** is a pointer to a normal *one*-dimensional array of size equal to the product of dimensions ***dim1***, ***dim2*** and ***dim3***. However, the only way the array can be accessed is through the **operator()** method (lines 32-34), i.e., with the help of expressions like `T(i,j,k)`, where ***T*** is the name of an object of class **Arr3D**. The method treats the three arguments as matrix indices and calculates which element of the one-dimensional array would correspond to element of 3-dimensional array with these indices (note that the first dimension, ***dim1***, is not needed for that). The method returns reference to this element, so `T(i,j,k)` behaves like `T[i][j][k]` would behave for a "normal" 3-dimensional array. In particular, it is an l-value and can appear on the left-hand side of assignments (as on line 57).

We create two objects of class **Arr3D**. The first, ***T1***, corresponds to a 3-dimensional array with dimensions $1000\times1000\times50$; the second, ***T2***, is created by the default constructor and has dimensions $1\times1\times1$ (line 52). The array in ***T1*** is represented by one-dimensional array of 50 million elements; it is a private member, so the user does not have to know it. We asign values to elements in lines 54-57 — for the purpose of this example these values are equal to the sum of indices of each element. On line 58 we make an assignment of objects of our class. We then print two values

```
T2( 999, 999 , 2) = 2000
T2(   0,    0, 9) =    9
```

to check the correctness of assignment and function call overloading. One can see that indeed `T(i,j,k)` can be treated as elements of a 3-dimensional array.

Operating on large matrices often requires careful considerations and experience; for example, changing the order of loops in lines 54-56 would yield equivalent program, but on most machines it would be 3-6 times slower.

19.4.4 Overloading indirect member selection operator

Operator of indirect member selection ('`->`') can be overloaded as a *one*-argument (unary) operator. Therefore, one overloads it with a parameterless method. It will be called for an object which appears on the left-hand side of the operator, without passing any arguments. Note that to the left of '`->`' operator we have an *object*, not a pointer! Let us recall here that the direct member selection operator, "dot" operator, cannot be overloaded at all).

Any method overloading the '`->`' oparator must return a pointer value (an address), which is then used as the left operand of "normal" '`->`' operator. It is, of course, possible, that what will be returned will be again an object of a class with overloaded `->` operator, which in turn returns a pointer (or again an object...).

Suppose ***obj*** is the name of an object of a class with overloaded '`->`' operator. Then an expression

```
obj->b
```

is equivalent to

```
temp = obj.operator->(), temp->b
```

In other words:

- method **operator->** is invoked for the object ***obj***;
- a value returned, denoted here by ***temp***, must be of a pointer type, i.e., is the address of an object of a class (not necessarily the same as that of object ***obj***!);
- from the object pointed to by ***temp***, a member ***b*** is selected (a member of that name must, of course, exist in the class of ****temp***). The value of this member will be the value of the whole expression.

Let us look at the following example. Variable ***AB*** is an object of class **Segment**, while `AB->x` will return x-coordinate of one of the two endpoints — objects of class **Point** — which define the segment. The return type of **operator->** method in class **Segment** is *pointer to constant object of class **Point*** (and *not* of class **Segment**):

P163: ovrlskl.cpp Overloading indirect member selection operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      int x, y;
6.
7.      Point(int x = 0, int y = 0) : x(x), y(y)
8.      { }
9.
10.     double r2() const { return x*x + y*y; }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.
16.     Segment(Point A = Point(), Point B = Point())
17.         : A(A), B(B)
18.     { }
19.
20.     const Point* operator->() const {
21.         return (A.r2() < B.r2()) ? &A : &B;
22.     }
23. };
24.
25. ostream& operator<<(ostream& str, const Point& A) {
26.     return str << "P[" << A.x << "," << A.y << "]";
27. }
28.
29. ostream& operator<<(ostream& str, const Segment& AB) {
30.     return str << AB.A << "--" << AB.B;
31. }
32.
33. int main() {
34.
35.     Point    A(1,0),  B(8,6),  C(4,3);
36.     Segment AB(A,B), BC(B,C), CA(C,A);
37.
38.     cout << "AB = " << AB << ": AB->x = "
39.                    << AB->x << endl;
40.
41.     cout << "BC = " << BC << ": BC->y = "
42.                    << BC->y << endl;
43.
44.     cout << "CA = " << CA << ": CA->x = "
45.                    << CA->x << endl;
46. }
```

The overloading method returns the pointer to (address of) the endpoint which lies closer to the origin of the coordinate system (line 21). From the object (point) pointed to by this pointer, member ***x*** or ***y*** is then selected by “ordinary” ' `->`' operator, as one can see on

lines 38-45, which print

```
AB = P[1,0]--P[8,6]: AB->x = 1
BC = P[8,6]--P[4,3]: BC->y = 3
CA = P[4,3]--P[1,0]: CA->x = 1
```

For example, ***BC*** on lines 41-42 is the name of an object of class **Segment** representing a segment with endpoints corresponding to objects ***B*** and ***C*** of class **Point**. In class **Segment**, the '`->`' operator is overloaded (lines 20-22) and returns the address of one of points which are members of the class — the one closer to the origin, in our case it is point $[4, 3]$. Then, from this object, the member ***y*** is selected. Note that in the class of ***BC***, which is class **Segment**, there is *no* field named ***y***. Additionally, ***BC*** is the name of an object, not of a pointer, so normally one would have to use dot operator instead of the “arrow”.

Prev: 19.3 Overloading with methods of **Up:** 19. Operator overloading. Move semantics
Next: 19.5 Move semantics

T.R. Werner, September 27, 2023; 22:32

Subsections

- 19.4.1 Assignment operator
- 19.4.2 Indexing operator
- 19.4.3 Function call operator
- 19.4.4 Overloading indirect member selection operator

19.4 “Special” operators

The ' =', ' ->', ' ()' and ' []' operators are in a way special, so we will discuss them in separate subsections. They all can be overloaded only by defining methods, never with global functions.

19.4.1 Assignment operator

Assignment operator ' =' should be overloaded almost always when there are pointer fields in a class (as we already know, also destructors and copy-constructors are usually needed in such situation). If it is not overloaded, the system will provide a default implementation of assignment, which will assign field by field of the two objects: this is often not what we would want to happen for pointer members. We would rather want to assign another value to data which is pointed to by a pointer, not the value of the pointer itself.

No default assignment operator will be generated for classes containing const fields, fields of a reference type, or fields which are objects of classes with private assignment operator, or for which, for the same reasons, assignment operator is not defined at all.

In such situations, if we really need assignments, we not only can but have to overload assignment opeartor ourselves.

Let us see what can go wrong if a class with a pointer field does not overload assignment operator properly:

P157: ovrlderr.cpp Lack of proper assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
```

```
 8.     A(const char* s)
 9.         : name(strcpy(new char[strlen(s)+1],s)) {
10.         cerr << "  ctor: " << (void*)name << endl;
11.     }
12.
13.     A(const A& k)
14.         : name(strcpy(new char[strlen(k.name)+1],k.name)) {
15.         cerr << "cpctor: " << (void*)name << endl;
16.     }
17.
18.     ~A() {
19.         cerr << "  dtor: "  << (void*)name << endl;
20.         delete [] name;
21.     }
22. };
23.
24. A ob1("ob1");
25.
26. int main() {
27.     cerr << "MAIN" << endl;
28.     A ob2(ob1);
29.     A ob3 = ob2; // copy-ctor
30.
31.     ob1 = ob3;
32.
33.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
34.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
35.
36.     cerr << "THE END" << endl;
37. }
```

There is a constructor, defined on lines 8-11, which fabricates a new object given a C-string: it allocates new region of memory to store the C-string and correctly makes a copy of the string (line 9) — this is similar to what we know from program `person3.cpp`. All this can be placed in the initialization list (see section on initialization lists). The expression

```
new char[strlen(s)+1]
```

allocates enough memory for the C-string ***s*** including terminating '\0' character. Then **strcpy** copies the string from ***s*** to the newly allocated segment of memory, and returns its address — this address is then used to initialize the member ***name*** of the object being constructed (functions **strlen** and **strcpy** have been described in chapter on strings).

The copy-constructor (lines 13-16) works in a similar way (see also section on copy-constructors).

Lines 18-21 define destructor, which releases memeory allocated in constructors. It seems that everything should work correctly; however, running the program reveals some problems:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
  ob1.name: 0x804b028
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b028
```

After returning from the **main** dstructor is called three times: this is not a surprise, as three objects have been created. However, two of them release the same segment of memory (starting at address 0x804b028). On the other hand, memory allocated for ***name*** belonging originally to object ***ob1*** (at address 0x804b008) has not been released at all!

One can see why it happened. During assignment from line 31, the content of object ***ob3*** was copied to ***ob1***. In particular, the address in member ***name*** of ***ob3***, allocated during creation of this object by the copy-constructor (line 29), was copied to the member ***name*** of ***ob1*** erasing what was there before, i.e., the address 0x804b008. This address is therefore completely lost and memory at this location will never be released. After the assignment, the two objects, ***ob1*** and ***ob3*** are distinct objects, but hold in ***name*** the address of the same region of memory (0x804b028 in our example).

It is now clear that when objects are destroyed after leaving **main**, destructor will try to release the same memory twice: first when destroying ***ob3*** and then when destroying ***ob1***. This is illegal and can crash the program — even if it does not, the state of the program becomes unpredictable.

Hence, we have to redefine assignment operation for objects of our class. We have to do it by overloading a method: asignment operator cannot be overloaded with global functions. The method should sensibly assign data corresponding to the object on the right-hand side of ' =' operator (passed as an argument) to data belonging to the object on the left-hand side (which will be *this* object). Arrays and objects pointed to by pointer members of *this* object have to be first released and then reallocated and populated with data from corresponding arrays or objects pointed to by pointer members of the other object. As an assignment `a=a` should always be legal, we have to take care not to release memory holding data which still will be needed! Moreover, for `a=b=c` to be valid, the operator (method) should return a reference to *this* object (' `return *this`'), so its value can be used in next assignment (`b=c` returns reference to ***b***, and the value of ***b*** *after* this assignment can then be assigned to ***a***). The argument (right operand of the operator) is usually passed by reference, not by value, to save on unnecessary copying. Therefore, the prototype of a method overloading assignment opearator for a class **A** will be

```
A& operator=(const A&);
```

Let us see all this in practice:

P158: ovrldeq.cpp Overloading assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
8.     A() : name(new char[1]) {
9.         cerr << "dfctor: " << (void*)name << endl;
10.        name[0] = '\0';
11.    }
12.
13.    A(const char* s)
14.        : name(strcpy(new char[strlen(s)+1],s)) {
15.        cerr << "  ctor: " << (void*)name << endl;
16.    }
17.
18.    A(const A& k)
19.        : name(strcpy(new char[strlen(k.name)+1],k.name)) {
20.        cerr << "cpctor: " << (void*)name << endl;
21.    }
22.
23.    A& operator=(const A& k) {
24.
25.        if (this == &k) return *this;
26.
27.        cerr << "delete: " << (void*)name << endl;
```

```
28.         delete [] name;
29.         name = strcpy(new char[strlen(k.name)+1],k.name);
30.         cerr << "   op=: " << (void*)name << endl;
31.         return *this;
32.     }
33.
34.     ~A() {
35.         cerr << "  dtor: "  << (void*)name << endl;
36.         delete [] name;
37.     }
38. };
39.
40. A ob1("ob1");
41.
42. int main() {
43.     cerr << "MAIN" << endl;
44.     A ob2(ob1);
45.     A ob3 = ob2; // copy-ctor
46.
47.     ob1 = ob3;
48.
49.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
50.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
51.
52.     cerr << "THE END" << endl;
53. }
```

Class **A** is almost identical to the one from the previous program. On lines 8-11, we have added a default (parameterless) constructor to make the class more complete. Note that even allocating an empty C-string, which takes one byte for '\0' character, we use the array form of **new** operator (with square brackets), although in principle we could have allocated a single **char** without using square brackets. However, the array form is necessary here, because the destructor always uses array form of **delete**.

We define a method overloading the assignment operator on lines 23-32. Its structure is typical for this kind of methods:

- it returns by reference the object for which it was invoked (`return *this`);
- it first checks (line 25), if *this* object is not the same as the object passed by argument. This can be done by comparing addresses of the two object: one is the value of **this** and the second is `&k`, the address of the variable passed by reference to the method. Equality of addresses means that the two objects are in fact the same object, so there is nothing to do and the methods returns;
- knowing that this is not the '`a=a`' case, we can delete the array (C-string) belonging to *this* object (line 28);
- we allocate a new region of memory for the array and we copy the contents of the corresponding array from the object `k`. Note that in '`a=a`' case, the source array would not exist after deleteing the array from *this* object, because these two arrays would be in fact the same array;
- we always return ***this***.

The main program itself does not differ from the previous one, but the results *do* differ:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
delete: 0x804b008
   op=: 0x804b008
  ob1.name: 0x804b008
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
```

```
dtor: 0x804b018
dtor: 0x804b008
```

When the program is executing the assignment from line 47 ('`ob1=ob3`'), memory for the C-string belonging to object ***ob1*** (under address 0x804b008) is first released, and then a new segment is allocated — in our example, the system allocated in fact the same region under address 0x804b008, but this does not matter: equally well it could have been any other segment. What is important is the fact that now every region allocated by **new[]** is eventually released by a corresponding **delete[]** and the destructors call **delete[]** for different addresses.

Let us recall here that an expression

```
A a = b;
```

does *not* denote an assignment: it is just another form of creating a new object using the copy-constructor — it is essentially equivalent to '`A a(b)`'. Symbol '`=`' denotes an assignment only when an l-value designating an *existing* object appears on the left-hand side of the operator.

Another example of overloading assignment operator can be found in the following program:

P159: ope.cpp An array-like class with assignment operator

```
1. #include <iostream>
2. #include <cstring>   // memcpy
3. using namespace std;
4.
5. class ArrInt {
6.     static int ID;
7.     int        id;
8.     int      size;
9.     int      *arr;
10. public:
11.     ArrInt(const int *t, int size)
12.         : id(++ID), size(size),
13.           arr((int*)memcpy(new int[size], t,
14.                            size*sizeof(int))) {
15.         cout << "      ctor: id = " << id << endl;
16.     }
17.
18.     ArrInt(const ArrInt& t)
19.         : id(++ID), size(t.size),
20.           arr((int*)memcpy(new int[size], t.arr,
21.                            size*sizeof(int))) {
22.         cout << " copy-ctor: id = " << t.id
23.              << "-->" << id << endl;
24.     }
25.
26.     ~ArrInt() {
27.         cout << "  Deleting: id = " << id << endl;
28.         delete [] arr;
29.     }
30.
31.     ArrInt& operator=(const ArrInt& t) {
32.         cout << "Assignment: id = " << id
33.              << "<--" << t.id << endl;
34.         if (this != &t) {
35.             delete [] arr;
36.             size = t.size;
37.             arr = (int*)memcpy(new int[size], t.arr,
38.                                size*sizeof(int));
39.         }
40.         return *this;
41.     }
42. };
```

```
43. int ArrInt::ID = 0;
44.
45. int main() {
46.     int arr[] = {1,2,3};
47.     int size  = sizeof(arr)/sizeof(int);
48.
49.     ArrInt *pt1 = new ArrInt(arr,size);
50.     ArrInt   t2 = *pt1;
51.     ArrInt   t3(t2);
52.
53.     *pt1 = t2;
54. }
```

Note that we have used **memcpy** instead of **strcpy**. It copies segments of memory without copying element by element in a loop and is very efficient. From the output

```
      ctor: id = 1
 copy-ctor: id = 1-->2
 copy-ctor: id = 2-->3
Assignment: id = 1<--2
  Deleting: id = 3
  Deleting: id = 2
```

we can see that the first element (created on line 49) has not been deleted. This is because it was created on the heap (with **new** operator) and not explicitely removed; the other objects were created on the stack, so they are deleted (and their destructor is called) automatically when the flow of control leaves the **main**.

19.4.2 Indexing operator

The indexing operator '`[]`' is a binary operator: in an expression '`a[i]`' the name ***a*** denotes the first argument (operand) and ***i*** the second one. A method overloading this operator must be a one-parameter method — it will be called for ***a*** with ***i*** passed as an argument. The "index" ***i*** does *not* have to be of an integer type; also the return type is arbitrary.

The prototype of a method overloading '`[]`' operator has the form

```
Type operator[](Type_arg);
```

with **Type** and **Type_arg** being return type and argument type, respectively. The method is invoked for object ***a*** when an expression

```
a[i]
```

is encounterd and ***a*** is the name of an object (or a reference to an object) of the class where the operator '`[]`' has been overloaded; of course the type of ***i*** must correspond to the declared type of the parameter of the method. The invocation will have the form

```
a.operator[](i)
```

Of course, the method may do whatever we want it to do, but it is reasonable if it resembles some sort of "indexing". If this is the rôle of the method, then, not to surprise the user, it should return an l-value (by reference), because normally indexing can appear both on the right and on the left-hand side of an assignment:

```
x = arr[i];
arr[j] = y;
```

A simple class **Letter** in the following program illustrates overloading of indexing operator, but also copy-constructor, destructor and oveloading of assignment — three elements ususally needed when a class contains a pointer field.

P160: letter.cpp Overloading indexing operator

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Letter {
6.     char* name;
7. public:
8.     Letter(const Letter& k)
9.         : name(strcpy(new char[strlen(k.name)+1],
10.                       k.name))
11.     { }
12.
13.     Letter(const char* name)
14.         : name(strcpy(new char[strlen(name)+1],
15.                       name))
16.     { }
17.
18.     Letter& operator=(const Letter&);
19.     char& operator[](int);
20.
21.     ~Letter() { delete [] name; }
22.
23.     friend ostream& operator<<(ostream&,const Letter&);
24. };
25.
26. char& Letter::operator[](int n) {
27.     int len = strlen(name);
28.     if ( n < 0 || n >= len )
29.         // reference to NUL if index is wrong
30.         return name[len];
31.     else
32.         return name[n];
33. }
34.
35. Letter& Letter::operator=(const Letter& k) {
36.     if (this == &k ) return *this;
37.     delete [] name;
38.     name = strcpy(new char[strlen(k.name)+1],k.name);
39.     return *this;
40. }
41.
42. ostream& operator<<(ostream& str, const Letter& k) {
43.     return str << k.name;
44. }
45.
46. int main() {
47.     Letter a("Benny");
48.     cout << "a=" << a << endl;
49.
50.     char c = a[100];
51.     if (c == '\0') cout << "Out of range!" << endl;
52.     else           cout << "character: "   << c << endl;
53.
54.     c = a[4];
55.     if (c == '\0') cout << "Out of range!" << endl;
56.     else           cout << "character: "   << c << endl;
57.
58.     a[0] = 'J';
59.     cout << "a=" << a << endl;
60. }
```

As we can see from lines 26-33, indexing returns a reference to the ***n***-th character of a C-

string ***name***. Defining such a method makes sense, because it can check if the value of an index is valid and, if not, return a reference to ' \0' character rather than some random value, as it would be the case for “normal” arrays. The printout

```
a=Benny
Out of range!
character: y
a=Jenny
```

shows that indeed the indexed name of an object of the class can be treated as the name of an array: it can appear on both sides of assignments.

However, the “index” does not have to be an *int*:

P161: ranges.cpp Floating point “index”

```
1. #include <iostream>
2. using namespace std;
3.
4. class Ranges {
5.     int    cntr[3];
6.     double min, max;
7. public:
8.     Ranges(double min, double max)
9.         : min(min), max(max) {
10.        cntr[0]=cntr[1]=cntr[2]=0;
11.    }
12.
13.    int& operator[](double x) {
14.        int i;
15.
16.        if      ( x >  max ) i = 2;
17.        else if ( x >= min ) i = 1;
18.        else                 i = 0;
19.
20.        return cntr[i];
21.    }
22.
23.    friend ostream& operator<<(ostream&, const Ranges&);
24. };
25.
26. ostream& operator<<(ostream& str, const Ranges& r) {
27.    return str << "Range = [" << r.min << ", " << r.max
28.               << "]:  " << "Below " << r.cntr[0]
29.               << "; inside " << r.cntr[1] << "; above "
30.               << r.cntr[2];
31. }
32.
33. int main() {
34.    Ranges range(3.0, 5.5);
35.    double x;
36.    cout << "Enter numbers; 0 terminates:" << endl;
37.
38.    while ( ( cin >> x) && x ) range[x]++;
39.
40.    cout << range << endl;
41.    cout << "x = 4.7 -> cntr = " << range[4.7] << endl;
42. }
```

The class **Ranges** defines fields ***min***, ***max*** and a three-element array of **int** s whose elements play the rôle of counters for data below ***min***, in the range [***min***, ***max***], and above ***max***, respectively. Indexing operator is overloaded with a method whose name is **operator[]** and which takes one argument of type **double**. If ***range*** is an object of class **Ranges** and ***x*** is of type **double**, expression `range[x]` is a reference to an element of

member array ***cntr***: `cntr[0]`, if the value of ***x*** is below ***min***, `cntr[1]`, it it is in the range [***min***, ***max***], and `licz[2]` if it is above ***max***. Therefore, expression on line 38 (`zakres[x]++`) increments an appropriate element of ***cntr*** according to the value of ***x***. Similarily, value of `zakres[4.7]` from line 41 is the value of this element of the array which corresponds to the value of the argument (4.7 lies in the range [3,5.5], so in our case it will be the current value of counter ***cntr[1]***). Th eoutput of the program

```
Enter numbers; 0 terminates:
8 7 6 5 4 6.8 9.4 1.2
3 4 5 1 6   3   2   7  7.9  3.1  0
Range = [3, 5.5]:  Below 3; inside 7; above 8
x = 4.7 -> cntr = 7
```

shows the effect of such overloading in action. On line 38 we used a logical conjuction in the condition part of a **while** loop. The loop terminates if either the stream ***cin*** has become *bad* (because an error ocurred or end of file had been reached) or number zero has been read.

In this example, indexing operator returns reference to a private member of the object: generally, such practice cannot be recommended, as it completely breaks privacy of a class.

19.4.3 Function call operator

Function call operator '`()`' is the only one which can take an arbitraty number of arguments. It can be overloaded only with a method, whose name is therefore **operator()** — here parentheses are part of the name, which in turn is followed by a list, perhaps empty, of parameters enclosed in another pair of parentheses. Therefore, the declaration of such a method looks like this:

```
Type operator()(Type_arg1, Type_arg2, Type_arg3);
```

where the number of parameters is arbitrary. The method will be called when an expression like

```
obj(a,b,c)
```

is encountered, where ***obj*** is the name of an object of a class with overloaded function call operator and types of ***a***, ***b*** and ***c*** match the types of parameters of the overloading method. The call will be equivalent to

```
obj.operator()(a,b,c)
```

Note that the form `obj(a,b,c)` looks like a “normal” function invocation, but here ***obj*** in the name of an *object*, not of a function. Such objects, which “can be called”, are often referred to as **function objects** or **callable objects**.

Invocation of such a method does not have to return an l-value — exactly as it is for “normal” functions, which most often return their results by value. However, we are not limited to such methods — equally well an overloading method may return a reference (which *is* an l-value).

The example below shows how to use overloading of the function call operator to get objects which behave as three-dimensional, dynamically allocated matrices (i.e., with dimensions which can be calculated or read in during execution). The method **operator()**

returns reference to a number of type **int** which is an element of the matrix represented by object. A call with three arguments plays the rôle of triple indexing of a matrix:

P162: arr3dim.cpp Overloading function call operator

```
1. #include <iostream>
2. #include <cstring>
3. #include <iomanip>
4. using namespace std;
5.
6. class Arr3D {
7.     int  dim1, dim2, dim3;
8.     int* arr;
9. public:
10.     Arr3D() : dim1(1),dim2(1),dim3(1),
11.               arr(new int[1])
12.     { }
13.
14.     Arr3D(int dim1, int dim2, int dim3)
15.         : dim1(dim1), dim2(dim2), dim3(dim3),
16.           arr(new int[dim1*dim2*dim3])
17.     { }
18.
19.     Arr3D(const Arr3D& t)
20.         : dim1(t.dim1), dim2(t.dim2), dim3(t.dim3),
21.           arr((int*)memcpy(new int[dim1*dim2*dim3],
22.                            t.arr,
23.                            dim1*dim2*dim3*sizeof(int)))
24.     { }
25.
26.     ~Arr3D() { delete [] arr; }
27.
28.     Arr3D& operator=(const Arr3D&);
29.     int& operator()(int,int,int);
30. };
31.
32. int& Arr3D::operator()(int n1, int n2, int n3)  {
33.     return *(arr + n1*dim2*dim3 + n2*dim3 + n3);
34. }
35.
36. Arr3D& Arr3D::operator=(const Arr3D& t) {
37.     if ( this != &t ) {
38.         delete [] arr;
39.         dim1 = t.dim1;
40.         dim2 = t.dim2;
41.         dim3 = t.dim3;
42.         arr  = (int*)memcpy(new int[dim1*dim2*dim3],
43.                             t.arr,
44.                             dim1*dim2*dim3*sizeof(int));
45.     }
46.     return *this;
47. }
48.
49. int main() {
50.     int dim1 = 1000, dim2 = 1000, dim3 = 50;
51.
52.     Arr3D T1(dim1,dim2,dim3), T2;
53.
54.     for (int i = 0; i < dim1; i++)
55.         for (int j = 0; j < dim2; j++)
56.             for (int k = 0; k < dim3; k++)
57.                 T1(i,j,k) = i+j+k;
58.     T2 = T1;
59.
60.     cout << "T2(999, 999 , 2) = "
61.          << setw(4) << T2(999,999,2) << endl;
62.
63.     cout << "T2(  0,    0, 9) = "
64.          << setw(4) << T2(  0,  0,9) << endl;
65. }
```

Note that we have defined a copy-constructor, destructor and assignment operator overloading: this is normal for classes with pointer fields. Objects of class **Arr3D** represent 3-dimensional matrices (with three indices). Actually, the class member ***arr*** is a pointer to a normal *one*-dimensional array of size equal to the product of dimensions ***dim1***, ***dim2*** and ***dim3***. However, the only way the array can be accessed is through the **operator()** method (lines 32-34), i.e., with the help of expressions like `T(i,j,k)`, where ***T*** is the name of an object of class **Arr3D**. The method treats the three arguments as matrix indices and calculates which element of the one-dimensional array would correspond to element of 3-dimensional array with these indices (note that the first dimension, ***dim1***, is not needed for that). The method returns reference to this element, so `T(i,j,k)` behaves like `T[i][j][k]` would behave for a "normal" 3-dimensional array. In particular, it is an l-value and can appear on the left-hand side of assignments (as on line 57).

We create two objects of class **Arr3D**. The first, ***T1***, corresponds to a 3-dimensional array with dimensions $1000\times1000\times50$; the second, ***T2***, is created by the default constructor and has dimensions $1\times1\times1$ (line 52). The array in ***T1*** is represented by one-dimensional array of 50 million elements; it is a private member, so the user does not have to know it. We asign values to elements in lines 54-57 — for the purpose of this example these values are equal to the sum of indices of each element. On line 58 we make an assignment of objects of our class. We then print two values

```
T2( 999, 999 , 2) = 2000
T2(   0,    0, 9) =    9
```

to check the correctness of assignment and function call overloading. One can see that indeed `T(i,j,k)` can be treated as elements of a 3-dimensional array.

Operating on large matrices often requires careful considerations and experience; for example, changing the order of loops in lines 54-56 would yield equivalent program, but on most machines it would be 3-6 times slower.

19.4.4 Overloading indirect member selection operator

Operator of indirect member selection ('`->`') can be overloaded as a *one*-argument (unary) operator. Therefore, one overloads it with a parameterless method. It will be called for an object which appears on the left-hand side of the operator, without passing any arguments. Note that to the left of '`->`' operator we have an *object*, not a pointer! Let us recall here that the direct member selection operator, "dot" operator, cannot be overloaded at all).

Any method overloading the '`->`' oparator must return a pointer value (an address), which is then used as the left operand of "normal" '`->`' operator. It is, of course, possible, that what will be returned will be again an object of a class with overloaded `->` operator, which in turn returns a pointer (or again an object...).

Suppose ***obj*** is the name of an object of a class with overloaded '`->`' operator. Then an expression

```
obj->b
```

is equivalent to

```
temp = obj.operator->(), temp->b
```

In other words:

- method **operator->** is invoked for the object ***obj***;
- a value returned, denoted here by ***temp***, must be of a pointer type, i.e., is the address of an object of a class (not necessarily the same as that of object ***obj***!);
- from the object pointed to by ***temp***, a member ***b*** is selected (a member of that name must, of course, exist in the class of ****temp***). The value of this member will be the value of the whole expression.

Let us look at the following example. Variable ***AB*** is an object of class **Segment**, while `AB->x` will return x-coordinate of one of the two endpoints — objects of class **Point** — which define the segment. The return type of **operator->** method in class **Segment** is *pointer to constant object of class **Point*** (and *not* of class **Segment**):

P163: ovrlskl.cpp Overloading indirect member selection operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      int x, y;
6.
7.      Point(int x = 0, int y = 0) : x(x), y(y)
8.      { }
9.
10.     double r2() const { return x*x + y*y; }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.
16.     Segment(Point A = Point(), Point B = Point())
17.         : A(A), B(B)
18.     { }
19.
20.     const Point* operator->() const {
21.         return (A.r2() < B.r2()) ? &A : &B;
22.     }
23. };
24.
25. ostream& operator<<(ostream& str, const Point& A) {
26.     return str << "P[" << A.x << "," << A.y << "]";
27. }
28.
29. ostream& operator<<(ostream& str, const Segment& AB) {
30.     return str << AB.A << "--" << AB.B;
31. }
32.
33. int main() {
34.
35.     Point    A(1,0),  B(8,6),  C(4,3);
36.     Segment AB(A,B), BC(B,C), CA(C,A);
37.
38.     cout << "AB = " << AB << ": AB->x = "
39.                    << AB->x << endl;
40.
41.     cout << "BC = " << BC << ": BC->y = "
42.                    << BC->y << endl;
43.
44.     cout << "CA = " << CA << ": CA->x = "
45.                    << CA->x << endl;
46. }
```

The overloading method returns the pointer to (address of) the endpoint which lies closer to the origin of the coordinate system (line 21). From the object (point) pointed to by this pointer, member ***x*** or ***y*** is then selected by “ordinary” ' `->`' operator, as one can see on

lines 38-45, which print

```
AB = P[1,0]--P[8,6]: AB->x = 1
BC = P[8,6]--P[4,3]: BC->y = 3
CA = P[4,3]--P[1,0]: CA->x = 1
```

For example, ***BC*** on lines 41-42 is the name of an object of class **Segment** representing a segment with endpoints corresponding to objects ***B*** and ***C*** of class **Point**. In class **Segment**, the '`->`' operator is overloaded (lines 20-22) and returns the address of one of points which are members of the class — the one closer to the origin, in our case it is point $[4, 3]$. Then, from this object, the member ***y*** is selected. Note that in the class of ***BC***, which is class **Segment**, there is *no* field named ***y***. Additionally, ***BC*** is the name of an object, not of a pointer, so normally one would have to use dot operator instead of the “arrow”.

Prev: 19.3 Overloading with methods of **Up:** 19. Operator overloading. Move semantics
Next: 19.5 Move semantics

T.R. Werner, September 27, 2023; 22:32

Subsections

- 19.4.1 Assignment operator
- 19.4.2 Indexing operator
- 19.4.3 Function call operator
- 19.4.4 Overloading indirect member selection operator

19.4 “Special” operators

The '=', '->', '()' and '[]' operators are in a way special, so we will discuss them in separate subsections. They all can be overloaded only by defining methods, never with global functions.

19.4.1 Assignment operator

Assignment operator '=' should be overloaded almost always when there are pointer fields in a class (as we already know, also destructors and copy-constructors are usually needed in such situation). If it is not overloaded, the system will provide a default implementation of assignment, which will assign field by field of the two objects: this is often not what we would want to happen for pointer members. We would rather want to assign another value to data which is pointed to by a pointer, not the value of the pointer itself.

No default assignment operator will be generated for classes containing const fields, fields of a reference type, or fields which are objects of classes with private assignment operator, or for which, for the same reasons, assignment operator is not defined at all.

In such situations, if we really need assignments, we not only can but have to overload assignment opeartor ourselves.

Let us see what can go wrong if a class with a pointer field does not overload assignment operator properly:

P157: ovrlderr.cpp Lack of proper assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
```

```
 8.     A(const char* s)
 9.         : name(strcpy(new char[strlen(s)+1],s)) {
10.         cerr << "  ctor: " << (void*)name << endl;
11.     }
12.
13.     A(const A& k)
14.         : name(strcpy(new char[strlen(k.name)+1],k.name)) {
15.         cerr << "cpctor: " << (void*)name << endl;
16.     }
17.
18.     ~A() {
19.         cerr << "  dtor: "  << (void*)name << endl;
20.         delete [] name;
21.     }
22. };
23.
24. A ob1("ob1");
25.
26. int main() {
27.     cerr << "MAIN" << endl;
28.     A ob2(ob1);
29.     A ob3 = ob2; // copy-ctor
30.
31.     ob1 = ob3;
32.
33.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
34.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
35.
36.     cerr << "THE END" << endl;
37. }
```

There is a constructor, defined on lines 8-11, which fabricates a new object given a C-string: it allocates new region of memory to store the C-string and correctly makes a copy of the string (line 9) — this is similar to what we know from program `person3.cpp`. All this can be placed in the initialization list (see section on initialization lists). The expression

```
new char[strlen(s)+1]
```

allocates enough memory for the C-string ***s*** including terminating '\0' character. Then **strcpy** copies the string from ***s*** to the newly allocated segment of memory, and returns its address — this address is then used to initialize the member ***name*** of the object being constructed (functions **strlen** and **strcpy** have been described in chapter on strings).

The copy-constructor (lines 13-16) works in a similar way (see also section on copy-constructors).

Lines 18-21 define destructor, which releases memeory allocated in constructors. It seems that everything should work correctly; however, running the program reveals some problems:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
  ob1.name: 0x804b028
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b028
```

After returning from the **main** dstructor is called three times: this is not a surprise, as three objects have been created. However, two of them release the same segment of memory (starting at address 0x804b028). On the other hand, memory allocated for ***name*** belonging originally to object ***ob1*** (at address 0x804b008) has not been released at all!

One can see why it happened. During assignment from line 31, the content of object ***ob3*** was copied to ***ob1***. In particular, the address in member ***name*** of ***ob3***, allocated during creation of this object by the copy-constructor (line 29), was copied to the member ***name*** of ***ob1*** erasing what was there before, i.e., the address 0x804b008. This address is therefore completely lost and memory at this location will never be released. After the assignment, the two objects, ***ob1*** and ***ob3*** are distinct objects, but hold in ***name*** the address of the same region of memory (0x804b028 in our example).

It is now clear that when objects are destroyed after leaving **main**, destructor will try to release the same memory twice: first when destroying ***ob3*** and then when destroying ***ob1***. This is illegal and can crash the program — even if it does not, the state of the program becomes unpredictable.

Hence, we have to redefine assignment operation for objects of our class. We have to do it by overloading a method: asignment operator cannot be overloaded with global functions. The method should sensibly assign data corresponding to the object on the right-hand side of ' =' operator (passed as an argument) to data belonging to the object on the left-hand side (which will be *this* object). Arrays and objects pointed to by pointer members of *this* object have to be first released and then reallocated and populated with data from corresponding arrays or objects pointed to by pointer members of the other object. As an assignment `a=a` should always be legal, we have to take care not to release memory holding data which still will be needed! Moreover, for `a=b=c` to be valid, the operator (method) should return a reference to *this* object (' `return *this`'), so its value can be used in next assignment (`b=c` returns reference to ***b***, and the value of ***b*** *after* this assignment can then be assigned to ***a***). The argument (right operand of the operator) is usually passed by reference, not by value, to save on unnecessary copying. Therefore, the prototype of a method overloading assignment opearator for a class **A** will be

```
A& operator=(const A&);
```

Let us see all this in practice:

P158: ovrldeq.cpp Overloading assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
8.     A() : name(new char[1]) {
9.         cerr << "dfctor: " << (void*)name << endl;
10.        name[0] = '\0';
11.    }
12.
13.    A(const char* s)
14.        : name(strcpy(new char[strlen(s)+1],s)) {
15.        cerr << "  ctor: " << (void*)name << endl;
16.    }
17.
18.    A(const A& k)
19.        : name(strcpy(new char[strlen(k.name)+1],k.name)) {
20.        cerr << "cpctor: " << (void*)name << endl;
21.    }
22.
23.    A& operator=(const A& k) {
24.
25.        if (this == &k) return *this;
26.
27.        cerr << "delete: " << (void*)name << endl;
```

```
28.         delete [] name;
29.         name = strcpy(new char[strlen(k.name)+1],k.name);
30.         cerr << "   op=: " << (void*)name << endl;
31.         return *this;
32.     }
33.
34.     ~A() {
35.         cerr << "  dtor: "  << (void*)name << endl;
36.         delete [] name;
37.     }
38. };
39.
40. A ob1("ob1");
41.
42. int main() {
43.     cerr << "MAIN" << endl;
44.     A ob2(ob1);
45.     A ob3 = ob2; // copy-ctor
46.
47.     ob1 = ob3;
48.
49.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
50.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
51.
52.     cerr << "THE END" << endl;
53. }
```

Class **A** is almost identical to the one from the previous program. On lines 8-11, we have added a default (parameterless) constructor to make the class more complete. Note that even allocating an empty C-string, which takes one byte for '\0' character, we use the array form of **new** operator (with square brackets), although in principle we could have allocated a single **char** without using square brackets. However, the array form is necessary here, because the destructor always uses array form of **delete**.

We define a method overloading the assignment operator on lines 23-32. Its structure is typical for this kind of methods:

- it returns by reference the object for which it was invoked (`return *this`);
- it first checks (line 25), if *this* object is not the same as the object passed by argument. This can be done by comparing addresses of the two object: one is the value of **this** and the second is `&k`, the address of the variable passed by reference to the method. Equality of addresses means that the two objects are in fact the same object, so there is nothing to do and the methods returns;
- knowing that this is not the '`a=a`' case, we can delete the array (C-string) belonging to *this* object (line 28);
- we allocate a new region of memory for the array and we copy the contents of the corresponding array from the object `k`. Note that in '`a=a`' case, the source array would not exist after deleteing the array from *this* object, because these two arrays would be in fact the same array;
- we always return ***this***.

The main program itself does not differ from the previous one, but the results *do* differ:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
delete: 0x804b008
   op=: 0x804b008
  ob1.name: 0x804b008
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
```

```
dtor: 0x804b018
dtor: 0x804b008
```

When the program is executing the assignment from line 47 ('`ob1=ob3`'), memory for the C-string belonging to object ***ob1*** (under address 0x804b008) is first released, and then a new segment is allocated — in our example, the system allocated in fact the same region under address 0x804b008, but this does not matter: equally well it could have been any other segment. What is important is the fact that now every region allocated by **new[]** is eventually released by a corresponding **delete[]** and the destructors call **delete[]** for different addresses.

Let us recall here that an expression

```
A a = b;
```

does *not* denote an assignment: it is just another form of creating a new object using the copy-constructor — it is essentially equivalent to '`A a(b)`'. Symbol '`=`' denotes an assignment only when an l-value designating an *existing* object appears on the left-hand side of the operator.

Another example of overloading assignment operator can be found in the following program:

P159: ope.cpp An array-like class with assignment operator

```
1. #include <iostream>
2. #include <cstring>   // memcpy
3. using namespace std;
4.
5. class ArrInt {
6.     static int ID;
7.     int        id;
8.     int      size;
9.     int      *arr;
10. public:
11.     ArrInt(const int *t, int size)
12.         : id(++ID), size(size),
13.           arr((int*)memcpy(new int[size], t,
14.                            size*sizeof(int))) {
15.         cout << "      ctor: id = " << id << endl;
16.     }
17.
18.     ArrInt(const ArrInt& t)
19.         : id(++ID), size(t.size),
20.           arr((int*)memcpy(new int[size], t.arr,
21.                            size*sizeof(int))) {
22.         cout << " copy-ctor: id = " << t.id
23.              << "-->" << id << endl;
24.     }
25.
26.     ~ArrInt() {
27.         cout << "  Deleting: id = " << id << endl;
28.         delete [] arr;
29.     }
30.
31.     ArrInt& operator=(const ArrInt& t) {
32.         cout << "Assignment: id = " << id
33.              << "<--" << t.id << endl;
34.         if (this != &t) {
35.             delete [] arr;
36.             size = t.size;
37.             arr = (int*)memcpy(new int[size], t.arr,
38.                                size*sizeof(int));
39.         }
40.         return *this;
41.     }
42. };
```

```
43. int ArrInt::ID = 0;
44.
45. int main() {
46.     int arr[] = {1,2,3};
47.     int size  = sizeof(arr)/sizeof(int);
48.
49.     ArrInt *pt1 = new ArrInt(arr,size);
50.     ArrInt   t2 = *pt1;
51.     ArrInt   t3(t2);
52.
53.     *pt1 = t2;
54. }
```

Note that we have used **memcpy** instead of **strcpy**. It copies segments of memory without copying element by element in a loop and is very efficient. From the output

```
      ctor: id = 1
 copy-ctor: id = 1-->2
 copy-ctor: id = 2-->3
Assignment: id = 1<--2
  Deleting: id = 3
  Deleting: id = 2
```

we can see that the first element (created on line 49) has not been deleted. This is because it was created on the heap (with **new** operator) and not explicitely removed; the other objects were created on the stack, so they are deleted (and their destructor is called) automatically when the flow of control leaves the **main**.

19.4.2 Indexing operator

The indexing operator '`[]`' is a binary operator: in an expression '`a[i]`' the name ***a*** denotes the first argument (operand) and ***i*** the second one. A method overloading this operator must be a one-parameter method — it will be called for ***a*** with ***i*** passed as an argument. The "index" ***i*** does *not* have to be of an integer type; also the return type is arbitrary.

The prototype of a method overloading '`[]`' operator has the form

```
Type operator[](Type_arg);
```

with **Type** and **Type_arg** being return type and argument type, respectively. The method is invoked for object ***a*** when an expression

```
a[i]
```

is encounterd and ***a*** is the name of an object (or a reference to an object) of the class where the operator '`[]`' has been overloaded; of course the type of ***i*** must correspond to the declared type of the parameter of the method. The invocation will have the form

```
a.operator[](i)
```

Of course, the method may do whatever we want it to do, but it is reasonable if it resembles some sort of "indexing". If this is the rôle of the method, then, not to surprise the user, it should return an l-value (by reference), because normally indexing can appear both on the right and on the left-hand side of an assignment:

```
x = arr[i];
arr[j] = y;
```

A simple class **Letter** in the following program illustrates overloading of indexing operator, but also copy-constructor, destructor and oveloading of assignment — three elements ususally needed when a class contains a pointer field.

P160: letter.cpp Overloading indexing operator

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Letter {
6.     char* name;
7. public:
8.     Letter(const Letter& k)
9.         : name(strcpy(new char[strlen(k.name)+1],
10.                       k.name))
11.     { }
12.
13.     Letter(const char* name)
14.         : name(strcpy(new char[strlen(name)+1],
15.                       name))
16.     { }
17.
18.     Letter& operator=(const Letter&);
19.     char& operator[](int);
20.
21.     ~Letter() { delete [] name; }
22.
23.     friend ostream& operator<<(ostream&,const Letter&);
24. };
25.
26. char& Letter::operator[](int n) {
27.     int len = strlen(name);
28.     if ( n < 0 || n >= len )
29.         // reference to NUL if index is wrong
30.         return name[len];
31.     else
32.         return name[n];
33. }
34.
35. Letter& Letter::operator=(const Letter& k) {
36.     if (this == &k ) return *this;
37.     delete [] name;
38.     name = strcpy(new char[strlen(k.name)+1],k.name);
39.     return *this;
40. }
41.
42. ostream& operator<<(ostream& str, const Letter& k) {
43.     return str << k.name;
44. }
45.
46. int main() {
47.     Letter a("Benny");
48.     cout << "a=" << a << endl;
49.
50.     char c = a[100];
51.     if (c == '\0') cout << "Out of range!" << endl;
52.     else           cout << "character: "   << c << endl;
53.
54.     c = a[4];
55.     if (c == '\0') cout << "Out of range!" << endl;
56.     else           cout << "character: "   << c << endl;
57.
58.     a[0] = 'J';
59.     cout << "a=" << a << endl;
60. }
```

As we can see from lines 26-33, indexing returns a reference to the ***n***-th character of a C-

string ***name***. Defining such a method makes sense, because it can check if the value of an index is valid and, if not, return a reference to ' \0' character rather than some random value, as it would be the case for “normal” arrays. The printout

```
a=Benny
Out of range!
character: y
a=Jenny
```

shows that indeed the indexed name of an object of the class can be treated as the name of an array: it can appear on both sides of assignments.

However, the “index” does not have to be an *int*:

P161: ranges.cpp Floating point “index”

```
1. #include <iostream>
2. using namespace std;
3.
4. class Ranges {
5.     int    cntr[3];
6.     double min, max;
7. public:
8.     Ranges(double min, double max)
9.         : min(min), max(max) {
10.        cntr[0]=cntr[1]=cntr[2]=0;
11.    }
12.
13.    int& operator[](double x) {
14.        int i;
15.
16.        if      ( x >  max ) i = 2;
17.        else if ( x >= min ) i = 1;
18.        else                 i = 0;
19.
20.        return cntr[i];
21.    }
22.
23.    friend ostream& operator<<(ostream&, const Ranges&);
24. };
25.
26. ostream& operator<<(ostream& str, const Ranges& r) {
27.    return str << "Range = [" << r.min << ", " << r.max
28.               << "]:  " << "Below " << r.cntr[0]
29.               << "; inside " << r.cntr[1] << "; above "
30.               << r.cntr[2];
31. }
32.
33. int main() {
34.    Ranges range(3.0, 5.5);
35.    double x;
36.    cout << "Enter numbers; 0 terminates:" << endl;
37.
38.    while ( ( cin >> x) && x ) range[x]++;
39.
40.    cout << range << endl;
41.    cout << "x = 4.7 -> cntr = " << range[4.7] << endl;
42. }
```

The class **Ranges** defines fields ***min***, ***max*** and a three-element array of **int** s whose elements play the rôle of counters for data below ***min***, in the range [***min***, ***max***], and above ***max***, respectively. Indexing operator is overloaded with a method whose name is **operator[]** and which takes one argument of type **double**. If ***range*** is an object of class **Ranges** and ***x*** is of type **double**, expression `range[x]` is a reference to an element of

member array ***cntr***: `cntr[0]`, if the value of ***x*** is below ***min***, `cntr[1]`, it it is in the range [***min***, ***max***], and `licz[2]` if it is above ***max***. Therefore, expression on line 38 (`zakres[x]++`) increments an appropriate element of ***cntr*** according to the value of ***x***. Similarily, value of `zakres[4.7]` from line 41 is the value of this element of the array which corresponds to the value of the argument (4.7 lies in the range [3,5.5], so in our case it will be the current value of counter ***cntr[1]***). Th eoutput of the program

```
Enter numbers; 0 terminates:
8 7 6 5 4 6.8 9.4 1.2
3 4 5 1 6   3   2   7  7.9  3.1  0
Range = [3, 5.5]:  Below 3; inside 7; above 8
x = 4.7 -> cntr = 7
```

shows the effect of such overloading in action. On line 38 we used a logical conjuction in the condition part of a **while** loop. The loop terminates if either the stream ***cin*** has become *bad* (because an error ocurred or end of file had been reached) or number zero has been read.

In this example, indexing operator returns reference to a private member of the object: generally, such practice cannot be recommended, as it completely breaks privacy of a class.

19.4.3 Function call operator

Function call operator '`()`' is the only one which can take an arbitraty number of arguments. It can be overloaded only with a method, whose name is therefore **operator()** — here parentheses are part of the name, which in turn is followed by a list, perhaps empty, of parameters enclosed in another pair of parentheses. Therefore, the declaration of such a method looks like this:

```
Type operator()(Type_arg1, Type_arg2, Type_arg3);
```

where the number of parameters is arbitrary. The method will be called when an expression like

```
obj(a,b,c)
```

is encountered, where ***obj*** is the name of an object of a class with overloaded function call operator and types of ***a***, ***b*** and ***c*** match the types of parameters of the overloading method. The call will be equivalent to

```
obj.operator()(a,b,c)
```

Note that the form `obj(a,b,c)` looks like a “normal” function invocation, but here ***obj*** in the name of an *object*, not of a function. Such objects, which “can be called”, are often referred to as **function objects** or **callable objects**.

Invocation of such a method does not have to return an l-value — exactly as it is for “normal” functions, which most often return their results by value. However, we are not limited to such methods — equally well an overloading method may return a reference (which *is* an l-value).

The example below shows how to use overloading of the function call operator to get objects which behave as three-dimensional, dynamically allocated matrices (i.e., with dimensions which can be calculated or read in during execution). The method **operator()**

returns reference to a number of type **int** which is an element of the matrix represented by object. A call with three arguments plays the rôle of triple indexing of a matrix:

P162: arr3dim.cpp Overloading function call operator

```
1. #include <iostream>
2. #include <cstring>
3. #include <iomanip>
4. using namespace std;
5.
6. class Arr3D {
7.     int  dim1, dim2, dim3;
8.     int* arr;
9. public:
10.     Arr3D() : dim1(1),dim2(1),dim3(1),
11.               arr(new int[1])
12.     { }
13.
14.     Arr3D(int dim1, int dim2, int dim3)
15.         : dim1(dim1), dim2(dim2), dim3(dim3),
16.           arr(new int[dim1*dim2*dim3])
17.     { }
18.
19.     Arr3D(const Arr3D& t)
20.         : dim1(t.dim1), dim2(t.dim2), dim3(t.dim3),
21.           arr((int*)memcpy(new int[dim1*dim2*dim3],
22.                            t.arr,
23.                            dim1*dim2*dim3*sizeof(int)))
24.     { }
25.
26.     ~Arr3D() { delete [] arr; }
27.
28.     Arr3D& operator=(const Arr3D&);
29.     int& operator()(int,int,int);
30. };
31.
32. int& Arr3D::operator()(int n1, int n2, int n3)  {
33.     return *(arr + n1*dim2*dim3 + n2*dim3 + n3);
34. }
35.
36. Arr3D& Arr3D::operator=(const Arr3D& t) {
37.     if ( this != &t ) {
38.         delete [] arr;
39.         dim1 = t.dim1;
40.         dim2 = t.dim2;
41.         dim3 = t.dim3;
42.         arr  = (int*)memcpy(new int[dim1*dim2*dim3],
43.                             t.arr,
44.                             dim1*dim2*dim3*sizeof(int));
45.     }
46.     return *this;
47. }
48.
49. int main() {
50.     int dim1 = 1000, dim2 = 1000, dim3 = 50;
51.
52.     Arr3D T1(dim1,dim2,dim3), T2;
53.
54.     for (int i = 0; i < dim1; i++)
55.         for (int j = 0; j < dim2; j++)
56.             for (int k = 0; k < dim3; k++)
57.                 T1(i,j,k) = i+j+k;
58.     T2 = T1;
59.
60.     cout << "T2(999, 999 , 2) = "
61.          << setw(4) << T2(999,999,2) << endl;
62.
63.     cout << "T2(  0,    0, 9) = "
64.          << setw(4) << T2(  0,  0,9) << endl;
65. }
```

Note that we have defined a copy-constructor, destructor and assignment operator overloading: this is normal for classes with pointer fields. Objects of class **Arr3D** represent 3-dimensional matrices (with three indices). Actually, the class member ***arr*** is a pointer to a normal *one*-dimensional array of size equal to the product of dimensions ***dim1***, ***dim2*** and ***dim3***. However, the only way the array can be accessed is through the **operator()** method (lines 32-34), i.e., with the help of expressions like `T(i,j,k)`, where ***T*** is the name of an object of class **Arr3D**. The method treats the three arguments as matrix indices and calculates which element of the one-dimensional array would correspond to element of 3-dimensional array with these indices (note that the first dimension, ***dim1***, is not needed for that). The method returns reference to this element, so `T(i,j,k)` behaves like `T[i][j][k]` would behave for a "normal" 3-dimensional array. In particular, it is an l-value and can appear on the left-hand side of assignments (as on line 57).

We create two objects of class **Arr3D**. The first, ***T1***, corresponds to a 3-dimensional array with dimensions $1000\times1000\times50$; the second, ***T2***, is created by the default constructor and has dimensions $1\times1\times1$ (line 52). The array in ***T1*** is represented by one-dimensional array of 50 million elements; it is a private member, so the user does not have to know it. We asign values to elements in lines 54-57 — for the purpose of this example these values are equal to the sum of indices of each element. On line 58 we make an assignment of objects of our class. We then print two values

```
T2( 999, 999 , 2) = 2000
T2(   0,    0, 9) =    9
```

to check the correctness of assignment and function call overloading. One can see that indeed `T(i,j,k)` can be treated as elements of a 3-dimensional array.

Operating on large matrices often requires careful considerations and experience; for example, changing the order of loops in lines 54-56 would yield equivalent program, but on most machines it would be 3-6 times slower.

19.4.4 Overloading indirect member selection operator

Operator of indirect member selection ('`->`') can be overloaded as a *one*-argument (unary) operator. Therefore, one overloads it with a parameterless method. It will be called for an object which appears on the left-hand side of the operator, without passing any arguments. Note that to the left of '`->`' operator we have an *object*, not a pointer! Let us recall here that the direct member selection operator, "dot" operator, cannot be overloaded at all).

Any method overloading the '`->`' oparator must return a pointer value (an address), which is then used as the left operand of "normal" '`->`' operator. It is, of course, possible, that what will be returned will be again an object of a class with overloaded `->` operator, which in turn returns a pointer (or again an object...).

Suppose ***obj*** is the name of an object of a class with overloaded '`->`' operator. Then an expression

```
obj->b
```

is equivalent to

```
temp = obj.operator->(), temp->b
```

In other words:

- method **operator->** is invoked for the object ***obj***;
- a value returned, denoted here by ***temp***, must be of a pointer type, i.e., is the address of an object of a class (not necessarily the same as that of object ***obj***!);
- from the object pointed to by ***temp***, a member ***b*** is selected (a member of that name must, of course, exist in the class of ****temp***). The value of this member will be the value of the whole expression.

Let us look at the following example. Variable ***AB*** is an object of class **Segment**, while `AB->x` will return x-coordinate of one of the two endpoints — objects of class **Point** — which define the segment. The return type of **operator->** method in class **Segment** is *pointer to constant object of class **Point*** (and *not* of class **Segment**):

P163: ovrlskl.cpp Overloading indirect member selection operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      int x, y;
6.
7.      Point(int x = 0, int y = 0) : x(x), y(y)
8.      { }
9.
10.     double r2() const { return x*x + y*y; }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.
16.     Segment(Point A = Point(), Point B = Point())
17.         : A(A), B(B)
18.     { }
19.
20.     const Point* operator->() const {
21.         return (A.r2() < B.r2()) ? &A : &B;
22.     }
23. };
24.
25. ostream& operator<<(ostream& str, const Point& A) {
26.     return str << "P[" << A.x << "," << A.y << "]";
27. }
28.
29. ostream& operator<<(ostream& str, const Segment& AB) {
30.     return str << AB.A << "--" << AB.B;
31. }
32.
33. int main() {
34.
35.     Point    A(1,0),  B(8,6),  C(4,3);
36.     Segment AB(A,B), BC(B,C), CA(C,A);
37.
38.     cout << "AB = " << AB << ": AB->x = "
39.                    << AB->x << endl;
40.
41.     cout << "BC = " << BC << ": BC->y = "
42.                    << BC->y << endl;
43.
44.     cout << "CA = " << CA << ": CA->x = "
45.                    << CA->x << endl;
46. }
```

The overloading method returns the pointer to (address of) the endpoint which lies closer to the origin of the coordinate system (line 21). From the object (point) pointed to by this pointer, member ***x*** or ***y*** is then selected by “ordinary” ' `->`' operator, as one can see on

lines 38-45, which print

```
AB = P[1,0]--P[8,6]: AB->x = 1
BC = P[8,6]--P[4,3]: BC->y = 3
CA = P[4,3]--P[1,0]: CA->x = 1
```

For example, ***BC*** on lines 41-42 is the name of an object of class **Segment** representing a segment with endpoints corresponding to objects ***B*** and ***C*** of class **Point**. In class **Segment**, the '`->`' operator is overloaded (lines 20-22) and returns the address of one of points which are members of the class — the one closer to the origin, in our case it is point $[4, 3]$. Then, from this object, the member ***y*** is selected. Note that in the class of ***BC***, which is class **Segment**, there is *no* field named ***y***. Additionally, ***BC*** is the name of an object, not of a pointer, so normally one would have to use dot operator instead of the “arrow”.

Prev: 19.3 Overloading with methods of **Up:** 19. Operator overloading. Move semantics
Next: 19.5 Move semantics

T.R. Werner, September 27, 2023; 22:32

Subsections

- 19.4.1 Assignment operator
- 19.4.2 Indexing operator
- 19.4.3 Function call operator
- 19.4.4 Overloading indirect member selection operator

19.4 “Special” operators

The '=', '->', '()' and '[]' operators are in a way special, so we will discuss them in separate subsections. They all can be overloaded only by defining methods, never with global functions.

19.4.1 Assignment operator

Assignment operator '=' should be overloaded almost always when there are pointer fields in a class (as we already know, also destructors and copy-constructors are usually needed in such situation). If it is not overloaded, the system will provide a default implementation of assignment, which will assign field by field of the two objects: this is often not what we would want to happen for pointer members. We would rather want to assign another value to data which is pointed to by a pointer, not the value of the pointer itself.

No default assignment operator will be generated for classes containing const fields, fields of a reference type, or fields which are objects of classes with private assignment operator, or for which, for the same reasons, assignment operator is not defined at all.

In such situations, if we really need assignments, we not only can but have to overload assignment opeartor ourselves.

Let us see what can go wrong if a class with a pointer field does not overload assignment operator properly:

P157: ovrlderr.cpp Lack of proper assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
```

```
 8.     A(const char* s)
 9.         : name(strcpy(new char[strlen(s)+1],s)) {
10.         cerr << "  ctor: " << (void*)name << endl;
11.     }
12.
13.     A(const A& k)
14.         : name(strcpy(new char[strlen(k.name)+1],k.name)) {
15.         cerr << "cpctor: " << (void*)name << endl;
16.     }
17.
18.     ~A() {
19.         cerr << "  dtor: "  << (void*)name << endl;
20.         delete [] name;
21.     }
22. };
23.
24. A ob1("ob1");
25.
26. int main() {
27.     cerr << "MAIN" << endl;
28.     A ob2(ob1);
29.     A ob3 = ob2; // copy-ctor
30.
31.     ob1 = ob3;
32.
33.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
34.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
35.
36.     cerr << "THE END" << endl;
37. }
```

There is a constructor, defined on lines 8-11, which fabricates a new object given a C-string: it allocates new region of memory to store the C-string and correctly makes a copy of the string (line 9) — this is similar to what we know from program `person3.cpp`. All this can be placed in the initialization list (see section on initialization lists). The expression

```
new char[strlen(s)+1]
```

allocates enough memory for the C-string ***s*** including terminating '\0' character. Then **strcpy** copies the string from ***s*** to the newly allocated segment of memory, and returns its address — this address is then used to initialize the member ***name*** of the object being constructed (functions **strlen** and **strcpy** have been described in chapter on strings).

The copy-constructor (lines 13-16) works in a similar way (see also section on copy-constructors).

Lines 18-21 define destructor, which releases memeory allocated in constructors. It seems that everything should work correctly; however, running the program reveals some problems:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
  ob1.name: 0x804b028
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b028
```

After returning from the **main** dstructor is called three times: this is not a surprise, as three objects have been created. However, two of them release the same segment of memory (starting at address 0x804b028). On the other hand, memory allocated for ***name*** belonging originally to object ***ob1*** (at address 0x804b008) has not been released at all!

One can see why it happened. During assignment from line 31, the content of object ***ob3*** was copied to ***ob1***. In particular, the address in member ***name*** of ***ob3***, allocated during creation of this object by the copy-constructor (line 29), was copied to the member ***name*** of ***ob1*** erasing what was there before, i.e., the address 0x804b008. This address is therefore completely lost and memory at this location will never be released. After the assignment, the two objects, ***ob1*** and ***ob3*** are distinct objects, but hold in ***name*** the address of the same region of memory (0x804b028 in our example).

It is now clear that when objects are destroyed after leaving **main**, destructor will try to release the same memory twice: first when destroying ***ob3*** and then when destroying ***ob1***. This is illegal and can crash the program — even if it does not, the state of the program becomes unpredictable.

Hence, we have to redefine assignment operation for objects of our class. We have to do it by overloading a method: asignment operator cannot be overloaded with global functions. The method should sensibly assign data corresponding to the object on the right-hand side of ' =' operator (passed as an argument) to data belonging to the object on the left-hand side (which will be *this* object). Arrays and objects pointed to by pointer members of *this* object have to be first released and then reallocated and populated with data from corresponding arrays or objects pointed to by pointer members of the other object. As an assignment `a=a` should always be legal, we have to take care not to release memory holding data which still will be needed! Moreover, for `a=b=c` to be valid, the operator (method) should return a reference to *this* object (' `return *this`'), so its value can be used in next assignment (`b=c` returns reference to ***b***, and the value of ***b*** *after* this assignment can then be assigned to ***a***). The argument (right operand of the operator) is usually passed by reference, not by value, to save on unnecessary copying. Therefore, the prototype of a method overloading assignment opearator for a class **A** will be

```
A& operator=(const A&);
```

Let us see all this in practice:

P158: ovrldeq.cpp Overloading assignment operator

```
1. #include <iostream>
2. #include <cstring>   // strcpy, strlen
3. using namespace std;
4.
5. struct A {
6.     char* name;
7.
8.     A() : name(new char[1]) {
9.         cerr << "dfctor: " << (void*)name << endl;
10.        name[0] = '\0';
11.    }
12.
13.    A(const char* s)
14.        : name(strcpy(new char[strlen(s)+1],s)) {
15.        cerr << "  ctor: " << (void*)name << endl;
16.    }
17.
18.    A(const A& k)
19.        : name(strcpy(new char[strlen(k.name)+1],k.name)) {
20.        cerr << "cpctor: " << (void*)name << endl;
21.    }
22.
23.    A& operator=(const A& k) {
24.
25.        if (this == &k) return *this;
26.
27.        cerr << "delete: " << (void*)name << endl;
```

```
28.         delete [] name;
29.         name = strcpy(new char[strlen(k.name)+1],k.name);
30.         cerr << "   op=: " << (void*)name << endl;
31.         return *this;
32.     }
33.
34.     ~A() {
35.         cerr << "  dtor: "  << (void*)name << endl;
36.         delete [] name;
37.     }
38. };
39.
40. A ob1("ob1");
41.
42. int main() {
43.     cerr << "MAIN" << endl;
44.     A ob2(ob1);
45.     A ob3 = ob2; // copy-ctor
46.
47.     ob1 = ob3;
48.
49.     cerr << "  ob1.name: " << (void*)ob1.name << endl;
50.     cerr << "  ob3.name: " << (void*)ob3.name << endl;
51.
52.     cerr << "THE END" << endl;
53. }
```

Class **A** is almost identical to the one from the previous program. On lines 8-11, we have added a default (parameterless) constructor to make the class more complete. Note that even allocating an empty C-string, which takes one byte for '\0' character, we use the array form of **new** operator (with square brackets), although in principle we could have allocated a single **char** without using square brackets. However, the array form is necessary here, because the destructor always uses array form of **delete**.

We define a method overloading the assignment operator on lines 23-32. Its structure is typical for this kind of methods:

- it returns by reference the object for which it was invoked (`return *this`);
- it first checks (line 25), if *this* object is not the same as the object passed by argument. This can be done by comparing addresses of the two object: one is the value of **this** and the second is `&k`, the address of the variable passed by reference to the method. Equality of addresses means that the two objects are in fact the same object, so there is nothing to do and the methods returns;
- knowing that this is not the '`a=a`' case, we can delete the array (C-string) belonging to *this* object (line 28);
- we allocate a new region of memory for the array and we copy the contents of the corresponding array from the object `k`. Note that in '`a=a`' case, the source array would not exist after deleteing the array from *this* object, because these two arrays would be in fact the same array;
- we always return ***this***.

The main program itself does not differ from the previous one, but the results *do* differ:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
delete: 0x804b008
   op=: 0x804b008
  ob1.name: 0x804b008
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
```

```
dtor: 0x804b018
dtor: 0x804b008
```

When the program is executing the assignment from line 47 ('`ob1=ob3`'), memory for the C-string belonging to object ***ob1*** (under address 0x804b008) is first released, and then a new segment is allocated — in our example, the system allocated in fact the same region under address 0x804b008, but this does not matter: equally well it could have been any other segment. What is important is the fact that now every region allocated by **new[]** is eventually released by a corresponding **delete[]** and the destructors call **delete[]** for different addresses.

Let us recall here that an expression

```
A a = b;
```

does *not* denote an assignment: it is just another form of creating a new object using the copy-constructor — it is essentially equivalent to '`A a(b)`'. Symbol '`=`' denotes an assignment only when an l-value designating an *existing* object appears on the left-hand side of the operator.

Another example of overloading assignment operator can be found in the following program:

P159: ope.cpp An array-like class with assignment operator

```
1. #include <iostream>
2. #include <cstring>   // memcpy
3. using namespace std;
4.
5. class ArrInt {
6.     static int ID;
7.     int        id;
8.     int      size;
9.     int      *arr;
10. public:
11.     ArrInt(const int *t, int size)
12.         : id(++ID), size(size),
13.           arr((int*)memcpy(new int[size], t,
14.                            size*sizeof(int))) {
15.         cout << "      ctor: id = " << id << endl;
16.     }
17.
18.     ArrInt(const ArrInt& t)
19.         : id(++ID), size(t.size),
20.           arr((int*)memcpy(new int[size], t.arr,
21.                            size*sizeof(int))) {
22.         cout << " copy-ctor: id = " << t.id
23.              << "-->" << id << endl;
24.     }
25.
26.     ~ArrInt() {
27.         cout << "  Deleting: id = " << id << endl;
28.         delete [] arr;
29.     }
30.
31.     ArrInt& operator=(const ArrInt& t) {
32.         cout << "Assignment: id = " << id
33.              << "<--" << t.id << endl;
34.         if (this != &t) {
35.             delete [] arr;
36.             size = t.size;
37.             arr = (int*)memcpy(new int[size], t.arr,
38.                                size*sizeof(int));
39.         }
40.         return *this;
41.     }
42. };
```

```
43. int ArrInt::ID = 0;
44.
45. int main() {
46.     int arr[] = {1,2,3};
47.     int size  = sizeof(arr)/sizeof(int);
48.
49.     ArrInt *pt1 = new ArrInt(arr,size);
50.     ArrInt   t2 = *pt1;
51.     ArrInt   t3(t2);
52.
53.     *pt1 = t2;
54. }
```

Note that we have used **memcpy** instead of **strcpy**. It copies segments of memory without copying element by element in a loop and is very efficient. From the output

```
      ctor: id = 1
 copy-ctor: id = 1-->2
 copy-ctor: id = 2-->3
Assignment: id = 1<--2
  Deleting: id = 3
  Deleting: id = 2
```

we can see that the first element (created on line 49) has not been deleted. This is because it was created on the heap (with **new** operator) and not explicitely removed; the other objects were created on the stack, so they are deleted (and their destructor is called) automatically when the flow of control leaves the **main**.

19.4.2 Indexing operator

The indexing operator '`[]`' is a binary operator: in an expression '`a[i]`' the name ***a*** denotes the first argument (operand) and ***i*** the second one. A method overloading this operator must be a one-parameter method — it will be called for ***a*** with ***i*** passed as an argument. The "index" ***i*** does *not* have to be of an integer type; also the return type is arbitrary.

The prototype of a method overloading '`[]`' operator has the form

```
Type operator[](Type_arg);
```

with **Type** and **Type_arg** being return type and argument type, respectively. The method is invoked for object ***a*** when an expression

```
a[i]
```

is encounterd and ***a*** is the name of an object (or a reference to an object) of the class where the operator '`[]`' has been overloaded; of course the type of ***i*** must correspond to the declared type of the parameter of the method. The invocation will have the form

```
a.operator[](i)
```

Of course, the method may do whatever we want it to do, but it is reasonable if it resembles some sort of "indexing". If this is the rôle of the method, then, not to surprise the user, it should return an l-value (by reference), because normally indexing can appear both on the right and on the left-hand side of an assignment:

```
x = arr[i];
arr[j] = y;
```

A simple class **Letter** in the following program illustrates overloading of indexing operator, but also copy-constructor, destructor and oveloading of assignment — three elements ususally needed when a class contains a pointer field.

P160: letter.cpp Overloading indexing operator

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Letter {
6.     char* name;
7. public:
8.     Letter(const Letter& k)
9.         : name(strcpy(new char[strlen(k.name)+1],
10.                       k.name))
11.     { }
12.
13.     Letter(const char* name)
14.         : name(strcpy(new char[strlen(name)+1],
15.                       name))
16.     { }
17.
18.     Letter& operator=(const Letter&);
19.     char& operator[](int);
20.
21.     ~Letter() { delete [] name; }
22.
23.     friend ostream& operator<<(ostream&,const Letter&);
24. };
25.
26. char& Letter::operator[](int n) {
27.     int len = strlen(name);
28.     if ( n < 0 || n >= len )
29.         // reference to NUL if index is wrong
30.         return name[len];
31.     else
32.         return name[n];
33. }
34.
35. Letter& Letter::operator=(const Letter& k) {
36.     if (this == &k ) return *this;
37.     delete [] name;
38.     name = strcpy(new char[strlen(k.name)+1],k.name);
39.     return *this;
40. }
41.
42. ostream& operator<<(ostream& str, const Letter& k) {
43.     return str << k.name;
44. }
45.
46. int main() {
47.     Letter a("Benny");
48.     cout << "a=" << a << endl;
49.
50.     char c = a[100];
51.     if (c == '\0') cout << "Out of range!" << endl;
52.     else           cout << "character: "   << c << endl;
53.
54.     c = a[4];
55.     if (c == '\0') cout << "Out of range!" << endl;
56.     else           cout << "character: "   << c << endl;
57.
58.     a[0] = 'J';
59.     cout << "a=" << a << endl;
60. }
```

As we can see from lines 26-33, indexing returns a reference to the ***n***-th character of a C-

string ***name***. Defining such a method makes sense, because it can check if the value of an index is valid and, if not, return a reference to ' \0' character rather than some random value, as it would be the case for “normal” arrays. The printout

```
a=Benny
Out of range!
character: y
a=Jenny
```

shows that indeed the indexed name of an object of the class can be treated as the name of an array: it can appear on both sides of assignments.

However, the “index” does not have to be an *int*:

P161: ranges.cpp Floating point “index”

```
1. #include <iostream>
2. using namespace std;
3.
4. class Ranges {
5.     int    cntr[3];
6.     double min, max;
7. public:
8.     Ranges(double min, double max)
9.         : min(min), max(max) {
10.        cntr[0]=cntr[1]=cntr[2]=0;
11.    }
12.
13.    int& operator[](double x) {
14.        int i;
15.
16.        if      ( x >  max ) i = 2;
17.        else if ( x >= min ) i = 1;
18.        else                 i = 0;
19.
20.        return cntr[i];
21.    }
22.
23.    friend ostream& operator<<(ostream&, const Ranges&);
24. };
25.
26. ostream& operator<<(ostream& str, const Ranges& r) {
27.    return str << "Range = [" << r.min << ", " << r.max
28.               << "]:  " << "Below " << r.cntr[0]
29.               << "; inside " << r.cntr[1] << "; above "
30.               << r.cntr[2];
31. }
32.
33. int main() {
34.    Ranges range(3.0, 5.5);
35.    double x;
36.    cout << "Enter numbers; 0 terminates:" << endl;
37.
38.    while ( ( cin >> x) && x ) range[x]++;
39.
40.    cout << range << endl;
41.    cout << "x = 4.7 -> cntr = " << range[4.7] << endl;
42. }
```

The class **Ranges** defines fields ***min***, ***max*** and a three-element array of **int** s whose elements play the rôle of counters for data below ***min***, in the range [***min***, ***max***], and above ***max***, respectively. Indexing operator is overloaded with a method whose name is **operator[]** and which takes one argument of type **double**. If ***range*** is an object of class **Ranges** and ***x*** is of type **double**, expression `range[x]` is a reference to an element of

member array ***cntr***: `cntr[0]`, if the value of ***x*** is below ***min***, `cntr[1]`, it it is in the range [***min***, ***max***], and `licz[2]` if it is above ***max***. Therefore, expression on line 38 (`zakres[x]++`) increments an appropriate element of ***cntr*** according to the value of ***x***. Similarily, value of `zakres[4.7]` from line 41 is the value of this element of the array which corresponds to the value of the argument (4.7 lies in the range [3,5.5], so in our case it will be the current value of counter ***cntr[1]***). Th eoutput of the program

```
Enter numbers; 0 terminates:
8 7 6 5 4 6.8 9.4 1.2
3 4 5 1 6   3   2   7  7.9  3.1  0
Range = [3, 5.5]:  Below 3; inside 7; above 8
x = 4.7 -> cntr = 7
```

shows the effect of such overloading in action. On line 38 we used a logical conjuction in the condition part of a **while** loop. The loop terminates if either the stream ***cin*** has become *bad* (because an error ocurred or end of file had been reached) or number zero has been read.

In this example, indexing operator returns reference to a private member of the object: generally, such practice cannot be recommended, as it completely breaks privacy of a class.

19.4.3 Function call operator

Function call operator '`()`' is the only one which can take an arbitraty number of arguments. It can be overloaded only with a method, whose name is therefore **operator()** — here parentheses are part of the name, which in turn is followed by a list, perhaps empty, of parameters enclosed in another pair of parentheses. Therefore, the declaration of such a method looks like this:

```
Type operator()(Type_arg1, Type_arg2, Type_arg3);
```

where the number of parameters is arbitrary. The method will be called when an expression like

```
obj(a,b,c)
```

is encountered, where ***obj*** is the name of an object of a class with overloaded function call operator and types of ***a***, ***b*** and ***c*** match the types of parameters of the overloading method. The call will be equivalent to

```
obj.operator()(a,b,c)
```

Note that the form `obj(a,b,c)` looks like a “normal” function invocation, but here ***obj*** in the name of an *object*, not of a function. Such objects, which “can be called”, are often referred to as **function objects** or **callable objects**.

Invocation of such a method does not have to return an l-value — exactly as it is for “normal” functions, which most often return their results by value. However, we are not limited to such methods — equally well an overloading method may return a reference (which *is* an l-value).

The example below shows how to use overloading of the function call operator to get objects which behave as three-dimensional, dynamically allocated matrices (i.e., with dimensions which can be calculated or read in during execution). The method **operator()**

returns reference to a number of type **int** which is an element of the matrix represented by object. A call with three arguments plays the rôle of triple indexing of a matrix:

P162: arr3dim.cpp Overloading function call operator

```
1. #include <iostream>
2. #include <cstring>
3. #include <iomanip>
4. using namespace std;
5.
6. class Arr3D {
7.     int  dim1, dim2, dim3;
8.     int* arr;
9. public:
10.     Arr3D() : dim1(1),dim2(1),dim3(1),
11.               arr(new int[1])
12.     { }
13.
14.     Arr3D(int dim1, int dim2, int dim3)
15.         : dim1(dim1), dim2(dim2), dim3(dim3),
16.           arr(new int[dim1*dim2*dim3])
17.     { }
18.
19.     Arr3D(const Arr3D& t)
20.         : dim1(t.dim1), dim2(t.dim2), dim3(t.dim3),
21.           arr((int*)memcpy(new int[dim1*dim2*dim3],
22.                            t.arr,
23.                            dim1*dim2*dim3*sizeof(int)))
24.     { }
25.
26.     ~Arr3D() { delete [] arr; }
27.
28.     Arr3D& operator=(const Arr3D&);
29.     int& operator()(int,int,int);
30. };
31.
32. int& Arr3D::operator()(int n1, int n2, int n3)  {
33.     return *(arr + n1*dim2*dim3 + n2*dim3 + n3);
34. }
35.
36. Arr3D& Arr3D::operator=(const Arr3D& t) {
37.     if ( this != &t ) {
38.         delete [] arr;
39.         dim1 = t.dim1;
40.         dim2 = t.dim2;
41.         dim3 = t.dim3;
42.         arr  = (int*)memcpy(new int[dim1*dim2*dim3],
43.                             t.arr,
44.                             dim1*dim2*dim3*sizeof(int));
45.     }
46.     return *this;
47. }
48.
49. int main() {
50.     int dim1 = 1000, dim2 = 1000, dim3 = 50;
51.
52.     Arr3D T1(dim1,dim2,dim3), T2;
53.
54.     for (int i = 0; i < dim1; i++)
55.         for (int j = 0; j < dim2; j++)
56.             for (int k = 0; k < dim3; k++)
57.                 T1(i,j,k) = i+j+k;
58.     T2 = T1;
59.
60.     cout << "T2(999, 999 , 2) = "
61.          << setw(4) << T2(999,999,2) << endl;
62.
63.     cout << "T2(  0,    0, 9) = "
64.          << setw(4) << T2(  0,  0,9) << endl;
65. }
```

Note that we have defined a copy-constructor, destructor and assignment operator overloading: this is normal for classes with pointer fields. Objects of class **Arr3D** represent 3-dimensional matrices (with three indices). Actually, the class member ***arr*** is a pointer to a normal *one*-dimensional array of size equal to the product of dimensions ***dim1***, ***dim2*** and ***dim3***. However, the only way the array can be accessed is through the **operator()** method (lines 32-34), i.e., with the help of expressions like `T(i,j,k)`, where ***T*** is the name of an object of class **Arr3D**. The method treats the three arguments as matrix indices and calculates which element of the one-dimensional array would correspond to element of 3-dimensional array with these indices (note that the first dimension, ***dim1***, is not needed for that). The method returns reference to this element, so `T(i,j,k)` behaves like `T[i][j][k]` would behave for a "normal" 3-dimensional array. In particular, it is an l-value and can appear on the left-hand side of assignments (as on line 57).

We create two objects of class **Arr3D**. The first, ***T1***, corresponds to a 3-dimensional array with dimensions $1000\times1000\times50$; the second, ***T2***, is created by the default constructor and has dimensions $1\times1\times1$ (line 52). The array in ***T1*** is represented by one-dimensional array of 50 million elements; it is a private member, so the user does not have to know it. We asign values to elements in lines 54-57 — for the purpose of this example these values are equal to the sum of indices of each element. On line 58 we make an assignment of objects of our class. We then print two values

```
T2( 999, 999 , 2) = 2000
T2(   0,    0, 9) =    9
```

to check the correctness of assignment and function call overloading. One can see that indeed `T(i,j,k)` can be treated as elements of a 3-dimensional array.

Operating on large matrices often requires careful considerations and experience; for example, changing the order of loops in lines 54-56 would yield equivalent program, but on most machines it would be 3-6 times slower.

19.4.4 Overloading indirect member selection operator

Operator of indirect member selection ('`->`') can be overloaded as a *one*-argument (unary) operator. Therefore, one overloads it with a parameterless method. It will be called for an object which appears on the left-hand side of the operator, without passing any arguments. Note that to the left of '`->`' operator we have an *object*, not a pointer! Let us recall here that the direct member selection operator, "dot" operator, cannot be overloaded at all).

Any method overloading the '`->`' oparator must return a pointer value (an address), which is then used as the left operand of "normal" '`->`' operator. It is, of course, possible, that what will be returned will be again an object of a class with overloaded `->` operator, which in turn returns a pointer (or again an object...).

Suppose ***obj*** is the name of an object of a class with overloaded '`->`' operator. Then an expression

```
obj->b
```

is equivalent to

```
temp = obj.operator->(), temp->b
```

In other words:

- method **operator->** is invoked for the object ***obj***;
- a value returned, denoted here by ***temp***, must be of a pointer type, i.e., is the address of an object of a class (not necessarily the same as that of object ***obj***!);
- from the object pointed to by ***temp***, a member ***b*** is selected (a member of that name must, of course, exist in the class of ****temp***). The value of this member will be the value of the whole expression.

Let us look at the following example. Variable ***AB*** is an object of class **Segment**, while `AB->x` will return x-coordinate of one of the two endpoints — objects of class **Point** — which define the segment. The return type of **operator->** method in class **Segment** is *pointer to constant object of class **Point*** (and *not* of class **Segment**):

P163: ovrlskl.cpp Overloading indirect member selection operator

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  struct Point {
5.      int x, y;
6.
7.      Point(int x = 0, int y = 0) : x(x), y(y)
8.      { }
9.
10.     double r2() const { return x*x + y*y; }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.
16.     Segment(Point A = Point(), Point B = Point())
17.         : A(A), B(B)
18.     { }
19.
20.     const Point* operator->() const {
21.         return (A.r2() < B.r2()) ? &A : &B;
22.     }
23. };
24.
25. ostream& operator<<(ostream& str, const Point& A) {
26.     return str << "P[" << A.x << "," << A.y << "]";
27. }
28.
29. ostream& operator<<(ostream& str, const Segment& AB) {
30.     return str << AB.A << "--" << AB.B;
31. }
32.
33. int main() {
34.
35.     Point    A(1,0),  B(8,6),  C(4,3);
36.     Segment AB(A,B), BC(B,C), CA(C,A);
37.
38.     cout << "AB = " << AB << ": AB->x = "
39.                    << AB->x << endl;
40.
41.     cout << "BC = " << BC << ": BC->y = "
42.                    << BC->y << endl;
43.
44.     cout << "CA = " << CA << ": CA->x = "
45.                    << CA->x << endl;
46. }
```

The overloading method returns the pointer to (address of) the endpoint which lies closer to the origin of the coordinate system (line 21). From the object (point) pointed to by this pointer, member ***x*** or ***y*** is then selected by “ordinary” ' `->`' operator, as one can see on

lines 38-45, which print

```
AB = P[1,0]--P[8,6]: AB->x = 1
BC = P[8,6]--P[4,3]: BC->y = 3
CA = P[4,3]--P[1,0]: CA->x = 1
```

For example, ***BC*** on lines 41-42 is the name of an object of class **Segment** representing a segment with endpoints corresponding to objects ***B*** and ***C*** of class **Point**. In class **Segment**, the '`->`' operator is overloaded (lines 20-22) and returns the address of one of points which are members of the class — the one closer to the origin, in our case it is point $[4, 3]$. Then, from this object, the member ***y*** is selected. Note that in the class of ***BC***, which is class **Segment**, there is *no* field named ***y***. Additionally, ***BC*** is the name of an object, not of a pointer, so normally one would have to use dot operator instead of the “arrow”.

Prev: 19.3 Overloading with methods of **Up:** 19. Operator overloading. Move semantics
Next: 19.5 Move semantics

T.R. Werner, September 27, 2023; 22:32

19.5 Move semantics

Starting from the C++11 version of the standard, there is another type of references (denoted not by one, but by two ampersands '&&'). They are called **r-references** or **r-value references**.

R-references must be bound to (be `another name' of) r-values only, i.e., temporary objects that do not have any name and we cannot take their addresses; l-values have values, but also a well defined and accessible for the programmer location in memory, while r-values are just values, so, for example, one cannot assign anything to them.

Let us consider the following statements

```
1.      int i = 2;
2.      int &r1 = i;            // 'normal' l-value reference
3.      int &&r2 = i;           // WRONG - rhs is an l-value
4.      int &r3 = i + 2;        // WRONG - rhs is an r-value
5.      const int &r4 = i + 2; // OK - const-reference bound to an rvalue
6.      int &&r5 = i + 2;       // OK
```

Line 2 is valid, as the variable ***i*** is an l-value and ***r1*** is a `normal', l-value reference. However, line 3 is illegal, as variable ***i*** is an l-value – by no means a temporary! – so an r-reference cannot be bound to it. Also line 4 is not valid, as `i+2` is not an l-value, so l-reference cannot be bound to it; however, in line 5, we have a reference to **const** and such references, as we remember, *may* be bound to temporaries. The last line is valid, as the right hand side is a temporary and on the left we have an r-reference.

When dealing with l- and r-values it's important to know which functions (or operators) return l-values and r-values. L-values are returned by assignment operator, subscript operator, dereference of pointers, *prefix* increment and decrement operators (but not postfix ones). On the other hand, r-values are returned by arithmetic, relational, bitwise, and *postfix* increment/decrement operators. We can bind l-reference to **const** or r-reference to to the result of these expressions.

The r-references are mainly used as parameters of functions. When a parameter is declared as an r-reference, only temporaries can be passed as the corresponding arguments. Therefore, we know that the referred-to object is about to be destroyed and it cannot be accessed by any other user, so it is safe to `steal' its resources without making any copies. A typical example would be a field of pointer type — it points to, for example, an array which logically belongs to the object, but physically does not. Normally, in such situations, we have to define copy constructor, override the assignment operator and define the destructor, so they take care of the array: instead of copying the pointer, we have to create new arrays and copy the content of arrays, not the pointers (and delete the array in the destructor). However, if we know that the `original' (in the copy constructor or

the assignment operator) will not be used any more, we *can* just copy the pointers, only taking care that the original is left in a valid state and nothing wrong will happen when it's being deleted. In this way we ensure that our class supports the so called **move semantics**.

It sometimes happens that we have an l-value that, for example, we want to pass to a function as an argument, but we know that after that we will not need it any more. Then we can explicitly ask the compiler to treat our l-value as an r-value: it can make copying trivial and much more effective. Such casting can be performed by calling the **std::move** function (from the `utility` header). Of course, we have to remember that our object is now in an unspecified (although valid and destructible) state.

How can we ensure that a class supports the move semantics? Firstly, we have to define a special constructor, **move constructor**, with its parameter declared as r-reference. Of course *not* **const**, as we *do* want to modify the original passed to the constructor — we want to `steal' resources from this object (by copying just pointers to them) and then modify the original object in such a way that it doesn't `own' them anymore (for example by nullifying its pointers).
Secondly, we override, in a similar way, **move-assignment operator.**

In the following example, we define a class **Arr** which just wraps an array of integers. The only members are size of the array and the *pointer* to it — the array itself is a `resource' which logically belongs to the object, but physically is allocated somewhere on the heap.
In line ❶, we define `normal' constructor, and in line ❷, the copy constructor. It takes an object of the same type by **const** reference, because it cannot modify it — therefore it has to allocate a new array and copy elements from the array owned by the other object to the one belonging to *this* one. Only in this way the two objects will remain independent one from the other.

P164: rmoveassign.cpp Move semantics

```
1.  #include <cstring>
2.  #include <iostream>
3.  #include <utility>   // move
4.
5.  using std::cout; using std::endl;
6.
7.  class Arr {
8.      size_t size;
9.      int*    arr;
10. public:
11.     Arr(size_t s, const int* a)                ❶
12.         : size(s),
13.           arr(static_cast<int*>(
14.               std::memcpy(new int[size],a,
15.                           size*sizeof(int))))
16.     {
17.         cout << "ctor from array\n";
18.     }
19.     Arr(const Arr& other)                      ❷
20.         : size(other.size),
21.           arr(static_cast<int*>(
```

```
22.                 std::memcpy(new int[size],other.arr,
23.                             size*sizeof(int))))
24.     {
25.         cout << "copy-ctor\n";
26.     }
27.     Arr(Arr&& other) noexcept                    ❸
28.         : size(other.size), arr(other.arr)
29.     {
30.         other.size = 0;
31.         other.arr = nullptr;
32.         cout << "move-ctor\n";
33.     }
34.     Arr& operator=(const Arr& other) {           ❹
35.         if (this == &other) return *this;
36.         int* a = new int[other.size];
37.         memcpy(a,other.arr,other.size*sizeof(int));
38.         delete [] arr;
39.         size = other.size;
40.         arr = a;
41.         cout << "copy-assign\n";
42.         return *this;
43.     }
44.     Arr& operator=(Arr&& other) noexcept {       ❺
45.         if (this == &other) return *this;
46.         delete [] arr;
47.         size = other.size;
48.         arr  = other.arr;
49.         other.size = 0;
50.         other.arr  = nullptr;
51.         cout << "move-assign\n";
52.         return *this;
53.     }
54.     ~Arr() {
55.         delete [] arr;
56.     }
57.     friend std::ostream& operator<<(std::ostream& str,
58.                                     const Arr& a) {
59.         if (a.size == 0) return cout << "Empty";
60.         str << "[ ";
61.         for (size_t i = 0; i < a.size; ++i)
62.             str << a.arr[i] << " ";
63.         return str << "]";
64.     }
65. };
66.
67. Arr replicate(Arr a) {                           ❻
68.     cout << "In replicate\n";
69.     return a;
70. }
71.
72. int main() {
73.     cout << "**** 0 ****\n";
74.     int a[]{1,2,3,4};
75.     Arr arr(std::size(a),a);
76.     cout << "arr : " << arr  << endl;
77.
78.     cout << "**** 1 ****\n";
79.     Arr arr1 = replicate(arr);
80.     cout << "arr1: " << arr1 << endl;
81.     cout << "arr : " << arr  << endl;
82.
83.     cout << "\n**** 2 ****\n";
84.     arr = arr1;
85.     cout << "arr : " << arr  << endl;
86.     cout << "arr1: " << arr1 << endl;
87.
88.     cout << "\n**** 3 ****\n";
89.     Arr arr2 = replicate(std::move(arr));
90.     cout << "arr2: " << arr2 << endl;
91.     cout << "arr : " << arr  << endl;
92.
93.     cout << "\n**** 4 ****\n";
94.     arr  = replicate(std::move(arr2));
95.     cout << "arr : " << arr  << endl;
96.     cout << "arr2: " << arr2 << endl;
97.
```

```
98.     cout << "\n**** 5 ****\n";
99.     arr2 = std::move(arr);
100.    cout << "arr2: " << arr2 << endl;
101.    cout << "arr : " << arr  << endl;
102. }
```

In line ❸, we define the move constructor, which will be invoked when the ***other*** object is a temporary. It just `steals' both fields, so in *this* object the pointer ***arr*** will point to the array owned by the ***other***. No array is created, no memory allocation takes place. However, we have to ensure that the ***other*** object will remain in a valid state and its destructor will *not* delete the array that we have just taken over. Nullifying ***arr*** in the *other* object will be sufficient, as invoking **delete** on **nullptr** in the destructor will be a safe no-op.
The move-assignment operator defined in line ❺ works in a similar fashion — we take over the resources and nullify the ***other*** object.

Note that both `moving functions' (constructor and assignment) are declared with **noexcept**: in this way we `promise' that they will never throw — they shouldn't, as they just copy values of primitive types. This is important; there are many functions in the Standard Library that will *not* use moving, what would be beneficial for efficiency, if these functions don't promise that they won't throw.
Note also the function **replicate** (❻): it just takes and returns objects of the **Arr** class.

Let us now analyze the program.
***** 0 *****: we create an object of type **Arr** using the first constructor and print it. The output is, of course

```
**** 0 ****
ctor from array
arr : [ 1 2 3 4 ]
```

***** 1 *****: we call **replicate** passing ***arr*** by value. As ***arr*** is an l-value, normal copy constructor will be used to create its copy to be pushed onto the stack. However, the functions returns a temporary that will be used to create ***arr1*** — as this is a temporary, move constructor is used

```
**** 1 ****
copy-ctor
In replicate
move-ctor
arr1: [ 1 2 3 4 ]
arr : [ 1 2 3 4 ]
```

***** 2 *****: in the assignment `arr=arr1` the object on the right is an l-value, so normal copy-assignment operator will be used

```
**** 2 ****
copy-assign
arr : [ 1 2 3 4 ]
arr1: [ 1 2 3 4 ]
```

***** 3 *****: now we pass ***arr*** to the **replicate** function as an r-value (cast by **std::move**). Therefore, move constructor will be used to make a copy which is pushed onto the stack. The returned temporary is then used to initialize ***arr2***, so again the move constructor is used. Note that ***arr*** has been nullified!

```
**** 3 ****
move-ctor
In replicate
move-ctor
arr2: [ 1 2 3 4 ]
```

```
arr : Empty
```

*** **4** ***: now we pass ***arr2*** as an r-value and assign the returned temporary to an existing object ***arr***. Therefore, move constructor will be used to create the temporary and move-assignment operator to assign it to ***arr***; now ***arr2*** has been nullified

```
**** 4 ****
move-ctor
In replicate
move-ctor
move-assign
arr : [ 1 2 3 4 ]
arr2: Empty
```

*** **5** ***: now we assign ***arr*** but cast to an r-value (by **move**) to ***arr2***, so move-assignment operator is used and ***arr*** is nullified

```
**** 5 ****
move-assign
arr2: [ 1 2 3 4 ]
arr : Empty
```

As we remember, copy constructor and copy-assignment operator are automatically synthesized by the compiler (if they are not **delete** d). With move constructor and move-assignment operator, the situation is a little different. If a class defines a copy constructor and/or copy-assignment operator and/or destructor, the move constructor and move-assignment operator are *not* synthesized automatically. Then, when moving is requested, the corresponding copy operation will be used instead.

If a class doesn't define any of its copy-control members – copy constructor, copy-assignment operator, destructor – the compiler will synthesize required move operation if all members can be moved: member of built in types can (this is equivalent to copying), members of object types must be themselves movable (for example, **string** s are.)

If, however, a class defines a move constructor and/or a move-assignment operator, then copy constructor and copy-assignment operator will be declared as deleted — we have to define them, if they are needed.

Generally, if any of the copy/move-control functions is needed, we should define all five (copy and move constructors, copy- and move- assignment operators, and the destructor).

It is even possible to overload methods of a class in such a way that different implementations will be invoked depending on whether the call was on an l-value or r-value. The overload designed for calling on l-values is marked with a single ampersand after the parameter list while the one for r-values — by a double ampersand, as in the following example

P165: RrefMet.cpp Overloading methods to be called on r- and l-values

```
1. #include <iostream>
2.
3. struct X {
4.     void fun()  & { std::cout << "L-value\n"; }
5.     void fun() && { std::cout << "R-value\n"; }
6. };
7.
```

```
int main() {
    X x{};
    x.fun();     // calling fun() on l-value
    X{}.fun();   // calling fun() on r-value
}
```

which prints

```
L-value
R-value
```

Prev: 19.4 “Special” operators **Up:** 19. Operator overloading. Move semantics **Next:** 19.6 Smart pointers

Subsections

- 19.6.1 Smart pointers of the *unique_ptr* type
- 19.6.2 Smart pointers of the *shared_ptr* type

19.6 Smart pointers

Allocating and releasing memory (with **new**/ **delete**) can be very tedious and error prone. New C++ features, in particular r-values and move semantics, have made it possible to develop an improved versions of the so called smart pointers. These are objects of classes concretized from the **shared_ptr** and **unique_ptr** class templates. Internally, they hold pointers to objects (of any type). Due to appropriate operator overloadings, they syntactically and semantically behave (at least to some extend, and this in particular applies to the **unique_ptr** pointers) as pointers to the objects they refer to. What is important, when variables of these types go out of scope, the appropriate actions are undertaken to release the resources (objects) `owned' by them.

19.6.1 Smart pointers of the *unique_ptr* type

An object of the **unique_ptr** type (let's call them u-pointers, for short) is supposed to be the sole `owner' of the object it manages. When this owner gets destroyed, or is released, or starts owning another object, the object previously owned gets destroyed, and any associated resources are released. Therefore, objects of this type cannot be copied or copy-assigned, as that would lead to a situation when two such smart pointers manage the same resource. However, they can be moved. Then the one from which we move looses ownership (is “nullified”), while the one to which we move gains ownership (and responsibility) of the managed resources (and releases the resources it owned previously).

Smart pointers of the **unique_ptr** type can be created in several ways; for example:

```
std::unique_ptr<int>    empty; // holds nullptr
std::unique_ptr<int>    pi(new int(1));
std::unique_ptr<Person> pp(new Person("Mary", 2001));
std::unique_ptr<int[]>  pa(new int[4]{1,2,2}); // array
*pi = 21;
pp->setName("Kate");
pa[2] = 3;
```

Note that ***pa*** represents a pointer to an array, so **operator[]** is properly overloaded and can be used to access individual elements — the dereference operator ***** and the arrow operator **->**, however, are then not defined. Otherwise, for non-array objects, the smart pointer may be manipulated using syntax and having semantics of a `raw' pointer, but, as they cannot be assigned or copied, they provide neither full valuelike nor pointerlike behaviour. In any case, when the managed array or object is to be destroyed, the correct version of deleting operator will be used — **delete[]** for arrays and **delete** for non-array

objects.
However, if just **delete** or **delete[]** is not appropriate, we can define and pass a custom **deleter**. It has to be a function (generally, a callable) returning nothing and taking a raw pointer of the appropriate type, as in the example below. Note, that the type of a custom deleter must be passed as the second type argument to the **unique_pointer** template to concretize it! In the program below, we use as a deleter an object of our own type **Del** in the first case, and a lambda in the second case

P166: delunique.cpp Custom deleter of *unique_ptr*

```
1. #include <functional>
2. #include <iostream>
3. #include <memory>
4. #include <string>
5.
6. using std::unique_ptr; using std::string;
7.
8. template <typename T>
9. struct Del {
10.     void operator()(T* p) {
11.         std::cout << "Del deleting " << *p << '\n';
12.         delete p;
13.     }
14. };
15.
16. int main() {
17.     {
18.         unique_ptr<string, Del<string>>
19.             us{new string{"Hello"}, Del<string>{}};
20.     }
21.     std::cout << "us is now out of scope\n";
22.
23.     {
24.         unique_ptr<double, std::function<void(double*)>>
25.             ud{new double{7.5},
26.                [](double* p) {
27.                    std::cout << "Deleting " << *p << '\n';
28.                    delete p;
29.                }
30.             };
31.     }
32.     std::cout << "ud is now out of scope\n";
33. }
```

The program prints

```
Del deleting Hello
us is now out of scope
Deleting 7.5
ud is now out of scope
```

To create a **unique_ptr** object, one can also use the **make_unique** function. The objects to be managed by the smart pointer will be created by the function, we only pass initializing values or arguments to a constructor. In case of arrays, only their size can be passed; there is no way to initialize them. Also, a small disadvantage of the **make_unique** function is that you cannot pass your own custom deleter — this is, however, rather seldom useful anyway.

```
std::unique_ptr<int>    pi = std::make_unique<int>(19);
std::unique_ptr<Person> pp =
                std::make_unique<Person>("Mary", 2001);
std::unique_ptr<int[]>  pa = std::make_unique<int[]>(3);
for (int i = 0; i < 3; ++i) pa[i] = i;
```

Smart pointers of the **unique_ptr** type cannot be copied or assigned, as that would violate the uniqueness of ownership. They *can* be moved, though. The output of the following program

P167: ownunique.cpp Passing ownership

```
1.  #include <iostream>
2.  #include <memory>      // smart pointers
3.  #include <string>
4.  #include <utility>     // move
5.
6.  using std::unique_ptr; using std::string;
7.
8.  template <typename T>
9.  struct Del {
10.     void operator()(T* p) {
11.         std::cout << "Del deleting " << *p << '\n';
12.         delete p;
13.     }
14. };
15.
16. template <typename T>
17. void print(const T* p) {
18.     if (p) std::cout << *p << " ";
19.     else   std::cout << "null ";
20. }
21.
22. int main() {
23.     unique_ptr<string, Del<string>>
24.             p1{new string{"abcde"}, Del<string>{}},
25.             p2{new string{"vwxyz"}, Del<string>{}};
26.
27.     print(p1.get()); print(p2.get()); std::cout << '\n'; ❶
28.     std::cout << "Now moving\n";
29.     p1 = std::move(p2);                                  ❷
30.     std::cout << "After moving\n";
31.     print(p1.get()); print(p2.get()); std::cout << '\n'; ❸
32.     std::cout << "Exiting from main\n";
33. }
```

is

```
abcde vwxyz
Now moving
Del deleting abcde
After moving
vwxyz null
Exiting from main
Del deleting vwxyz
```

Note that the **get** method (lines ❶ and ❸) returns the `raw' pointer held by the smart pointer: it shouldn't be assigned to a variable because then the principle of single ownership could easily be violated.
After the move-assignment (line ❷), as we can see from the output, object managed by the ***p1*** pointer is destroyed, the pointer assumes the responsibility for the object managed by ***p2***, and ***p2*** is `nullified'.

Other important methods of **unique_ptr** pointers include (**T** is the type of the object managed by the pointer):

T* release() — releases the ownership of the managed object, so the receiver object is `nullified'; returns the raw pointer to the managed object (or **nullptr** if the pointer was `empty') – now the caller is responsible for deleting the object in due course;

T* reset(a_pointer = nullptr) — deletes the previously managed object (if there was any) and starts managing the pointer passed as the argument (perhaps, or by default, **nullptr**).

The **reset** method is illustrated by the following program

P168: uniquereset.cpp Resetting *unique_ptr* pointers

```
1. #include <iostream>
2. #include <memory>
3.
4. using std::unique_ptr; using std::ostream; using std::cout;
5.
6. template <typename T>
7. struct Del {
8.     void operator()(T* p) {
9.         cout << "Del deleting " << *p << '\n';
10.        delete p;
11.    }
12. };
13.
14. struct Klazz {
15.     Klazz()  { cout << "Ctor Klazz\n"; }
16.     ~Klazz() { cout << "Dtor Klazz\n"; }
17.     friend ostream& operator<<(ostream& s, const Klazz& k) {
18.         return s << "object of type Klazz";
19.     }
20. };
21.
22. int main() {
23.     std::cout << "Creating  u-pointer\n";
24.     std::unique_ptr<Klazz, Del<Klazz>>
25.                             p(new Klazz{}, Del<Klazz>{});
26.     std::cout << "Resetting u-pointer\n";
27.     p.reset(new Klazz{});
28.     std::cout << "Releasing and deleting\n";
29.     p.reset(); // or reset(nullptr)
30. }
```

As we can see, when an u-pointer is being reset, a new object is constructed and only then the `old' object is destroyed by calling the deleter; for object types, also the destructor will, of course, be invoked.

The u-pointers are often used as elements of collections. The following program demonstrates how we can populate a vector. It also illustrates a very important fact: smart pointers of base classes, exactly as `raw' pointers, may point to objects of derived class and polymorphic invocations work as expected; here, we create a vector with four elements, one of which is an object of the base class **B** and the remaining three of the derived class **D**:

P169: uniquederiv.cpp U-pointers and polymorphism

```
1. #include <iostream>
2. #include <vector>
3. #include <memory>
4.
5. struct B {
6.     virtual void f() { std::cout << "f from B\n"; }
7.     virtual ~B() { }
8. };
9. struct D : B {
```

```
    D() { std::cout << "Ctor D\n"; }
    void f() override { std::cout << "f from D\n"; }
    ~D(){ std::cout << "Dtor D\n"; }
};

int main() {
    {
        std::vector<std::unique_ptr<B>> vec;
        vec.push_back(std::make_unique<B>());
        vec.push_back(std::make_unique<D>());
        vec.emplace_back(std::make_unique<D>());
        std::unique_ptr<B> d{new D};
        vec.push_back(std::move(d));
        for (const auto& up : vec) up->f();
    }
    std::cout << "now vec is out of scope\n";
}
```

Note that when the vector goes out of scope, all its elements release their managed objects, avoiding memory leakage; the program prints

```
Ctor D
Ctor D
Ctor D
f from B
f from D
f from D
f from D
Dtor D
Dtor D
Dtor D
vec out of scope
```

As raw pointers, the u-pointers may also be used in a logical context: if the pointer doesn't manage any object, so `up.get() == nullptr` is true, then it is interpreted as **false**, otherwise – as **true**.

19.6.2 Smart pointers of the *shared_ptr* type

Smart pointers of the **shared_ptr** type (s-pointers for short) can share ownership of the same resource, represented internally by a `raw' pointer. They are equipped with the `reference counting' mechanism — there exist a special data structure associated with s-pointers which keeps record of the number of s-pointers sharing the same managed object. When an s-pointer goes out of scope, the counter is decremented by one and only if it reaches zero, the managed resource is destroyed. Similarly, if we assign two such pointers, `p = q`, the counter associated with the resource managed by ***p*** is decremented (as ***p*** doesn't refer to this object any more), while that associated with the resource managed by ***q*** is incremented, as now also ***p*** refers to it. We can look at the current value of a counter invoking the **use_count** method.

Some of these features are illustrated by the following program:

P170: sharedcount.cpp Counters associated with *shared_ptr* smart pointers.

```
#include <iostream>
#include <memory>
using std::shared_ptr; using std::cout; using std::ostream;

```

```
class Klazz {
    char c;
public:
    Klazz(char c)
        : c{c} { cout << "Ctor " << c << '\n'; }
    ~Klazz()   { cout << "Dtor " << c << '\n'; }
    friend ostream& operator<<(ostream& s, const Klazz& k) {
        return s << k.c;
    }
};

void f(shared_ptr<Klazz> p) {
    cout << "In f: p=" << *p << ", count="
         << p.use_count() << '\n';
}

int main() {
    shared_ptr<Klazz> p = std::make_shared<Klazz>('A');
    shared_ptr<Klazz> q{new Klazz{'B'}};
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    f(p);
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "Now assigning p = q\n";
    p = q;
    cout << "After assignment\n";
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    cout << "Exiting from main\n";
}
```

The output is

```
Ctor A
Ctor B
p=A, count=1
q=B, count=1
In f: p=A, count=2
p=A, count=1
Now assigning p = q
Dtor A
After assignment
p=B, count=2
q=B, count=2
Exiting from main
Dtor B
```

As we can see, we can create an s-pointers by passing to its constructor an ordinary pointer. The default constructor creates an empty s-pointer, analogously to u-pointers. There is also a **make_shared** function, analogous to **make_unique** for u-pointers.
After creating ***p*** and ***q***, they both refer to two different objects, so the counters are both 1. Then we pass ***p*** to the **f** function by value — a copy has to be made which refers to the same object 'A' that ***p*** refers to. Therefore, inside the function, the counter associated with object 'A' is 2. After returning, the copy is removed from the stack, so the counter is decremented back to 1.
Now we assign ***q*** to ***p***: as ***p*** refers to 'A', the counter associated with this object is decremented and reaches zero: it is therefore deleted. Now both ***p*** and ***q*** refer to the same object 'B', so the counter is 2. After exiting from the **main** function, first ***q*** is removed from the stack (decrementing the counter to 1), and then ***p*** — the counter now reaches 0 and the managed object is deleted.

S-pointers support, as do u-pointers, custom deleters provided by the user. Unlike u-pointers, the type of an s-pointer does not depend on the type of the deleter — we can pass a deleter as the additional argument to the constructor. In particular, prior to C++17, if the resource managed by an s-pointer was an array, the default deleter did *not* use **delete[]**

as it should. We had to take care of this ourselves:

P171: shareddelete.cpp Array deleters

```
1. #include <iostream>
2. #include <memory>
3. using std::shared_ptr;
4.
5. template< typename T >
6. struct arrdel {
7.     void operator ()(T const *p) { delete[] p; }
8. };
9.
10. int main() {
11.     shared_ptr<int> sp(new int(1));
12.
13.       // pointer to int[] array - custom deleter
14.     shared_ptr<int> p1(new int[10], arrdel<int>());
15.       // ... or lambda
16.     shared_ptr<int> p2(new int[10'000'000],
17.                        [](int *p) { delete[] p; });
18.       // ... or the one from the library
19.     shared_ptr<int> p3(new int[3]{1, 2, 3},
20.                        std::default_delete<int[]>());
21.     std::cout << p3.get()[2] << " " << *p3 << std::endl; ❶
22.
23.       // since c++17 this will work
24.     shared_ptr<int[]> p4(new int[3]{4, 5, 6});
25.     std::cout << p4[2] << std::endl;
26. }
```

Also, prior to C++17 standard, even for s-pointers managing arrays, the subscript operator (**[]**) was *not* overridden; that's why it was impossible to use it in the line ❶ of the above program. Since the C++17 standard, though, s-pointers may be created to manage arrays and the correct deleter (using **delete[]**) will be used without the user having to provide a custom one. The program, compiled by a compiler supporting C++17 standard, prints

```
3 1
6
```

As the u-pointers, the s-pointers may be used in a logical context (**false** if they are empty, **true** otherwise). They also have the **get** method and overridden ***** and **->** operators.

Prev: 19.5 Move semantics **Up:** 19. Operator overloading. Move semantics **Next:** 20. More on conversions

Subsections

- 19.6.1 Smart pointers of the *unique_ptr* type
- 19.6.2 Smart pointers of the *shared_ptr* type

19.6 Smart pointers

Allocating and releasing memory (with **new**/ **delete**) can be very tedious and error prone. New C++ features, in particular r-values and move semantics, have made it possible to develop an improved versions of the so called smart pointers. These are objects of classes concretized from the **shared_ptr** and **unique_ptr** class templates. Internally, they hold pointers to objects (of any type). Due to appropriate operator overloadings, they syntactically and semantically behave (at least to some extend, and this in particular applies to the **unique_ptr** pointers) as pointers to the objects they refer to. What is important, when variables of these types go out of scope, the appropriate actions are undertaken to release the resources (objects) `owned' by them.

19.6.1 Smart pointers of the *unique_ptr* type

An object of the **unique_ptr** type (let's call them u-pointers, for short) is supposed to be the sole `owner' of the object it manages. When this owner gets destroyed, or is released, or starts owning another object, the object previously owned gets destroyed, and any associated resources are released. Therefore, objects of this type cannot be copied or copy-assigned, as that would lead to a situation when two such smart pointers manage the same resource. However, they can be moved. Then the one from which we move looses ownership (is “nullified”), while the one to which we move gains ownership (and responsibility) of the managed resources (and releases the resources it owned previously).

Smart pointers of the **unique_ptr** type can be created in several ways; for example:

```
std::unique_ptr<int>    empty; // holds nullptr
std::unique_ptr<int>    pi(new int(1));
std::unique_ptr<Person> pp(new Person("Mary", 2001));
std::unique_ptr<int[]>  pa(new int[4]{1,2,2}); // array
*pi = 21;
pp->setName("Kate");
pa[2] = 3;
```

Note that ***pa*** represents a pointer to an array, so **operator[]** is properly overloaded and can be used to access individual elements — the dereference operator ***** and the arrow operator **->**, however, are then not defined. Otherwise, for non-array objects, the smart pointer may be manipulated using syntax and having semantics of a `raw' pointer, but, as they cannot be assigned or copied, they provide neither full valuelike nor pointerlike behaviour. In any case, when the managed array or object is to be destroyed, the correct version of deleting operator will be used — **delete[]** for arrays and **delete** for non-array

objects.
However, if just **delete** or **delete[]** is not appropriate, we can define and pass a custom **deleter**. It has to be a function (generally, a callable) returning nothing and taking a raw pointer of the appropriate type, as in the example below. Note, that the type of a custom deleter must be passed as the second type argument to the **unique_pointer** template to concretize it! In the program below, we use as a deleter an object of our own type **Del** in the first case, and a lambda in the second case

P166: delunique.cpp Custom deleter of *unique_ptr*

```
1. #include <functional>
2. #include <iostream>
3. #include <memory>
4. #include <string>
5.
6. using std::unique_ptr; using std::string;
7.
8. template <typename T>
9. struct Del {
10.     void operator()(T* p) {
11.         std::cout << "Del deleting " << *p << '\n';
12.         delete p;
13.     }
14. };
15.
16. int main() {
17.     {
18.         unique_ptr<string, Del<string>>
19.             us{new string{"Hello"}, Del<string>{}};
20.     }
21.     std::cout << "us is now out of scope\n";
22.
23.     {
24.         unique_ptr<double, std::function<void(double*)>>
25.             ud{new double{7.5},
26.                [](double* p) {
27.                    std::cout << "Deleting " << *p << '\n';
28.                    delete p;
29.                }
30.             };
31.     }
32.     std::cout << "ud is now out of scope\n";
33. }
```

The program prints

```
Del deleting Hello
us is now out of scope
Deleting 7.5
ud is now out of scope
```

To create a **unique_ptr** object, one can also use the **make_unique** function. The objects to be managed by the smart pointer will be created by the function, we only pass initializing values or arguments to a constructor. In case of arrays, only their size can be passed; there is no way to initialize them. Also, a small disadvantage of the **make_unique** function is that you cannot pass your own custom deleter — this is, however, rather seldom useful anyway.

```
std::unique_ptr<int>    pi = std::make_unique<int>(19);
std::unique_ptr<Person> pp =
                std::make_unique<Person>("Mary", 2001);
std::unique_ptr<int[]>  pa = std::make_unique<int[]>(3);
for (int i = 0; i < 3; ++i) pa[i] = i;
```

Smart pointers of the **unique_ptr** type cannot be copied or assigned, as that would violate the uniqueness of ownership. They *can* be moved, though. The output of the following program

P167: ownunique.cpp Passing ownership

```
1.  #include <iostream>
2.  #include <memory>      // smart pointers
3.  #include <string>
4.  #include <utility>     // move
5.
6.  using std::unique_ptr; using std::string;
7.
8.  template <typename T>
9.  struct Del {
10.     void operator()(T* p) {
11.         std::cout << "Del deleting " << *p << '\n';
12.         delete p;
13.     }
14. };
15.
16. template <typename T>
17. void print(const T* p) {
18.     if (p) std::cout << *p << " ";
19.     else   std::cout << "null ";
20. }
21.
22. int main() {
23.     unique_ptr<string, Del<string>>
24.             p1{new string{"abcde"}, Del<string>{}},
25.             p2{new string{"vwxyz"}, Del<string>{}};
26.
27.     print(p1.get()); print(p2.get()); std::cout << '\n'; ❶
28.     std::cout << "Now moving\n";
29.     p1 = std::move(p2);                                  ❷
30.     std::cout << "After moving\n";
31.     print(p1.get()); print(p2.get()); std::cout << '\n'; ❸
32.     std::cout << "Exiting from main\n";
33. }
```

is

```
abcde vwxyz
Now moving
Del deleting abcde
After moving
vwxyz null
Exiting from main
Del deleting vwxyz
```

Note that the **get** method (lines ❶ and ❸) returns the `raw' pointer held by the smart pointer: it shouldn't be assigned to a variable because then the principle of single ownership could easily be violated.
After the move-assignment (line ❷), as we can see from the output, object managed by the ***p1*** pointer is destroyed, the pointer assumes the responsibility for the object managed by ***p2***, and ***p2*** is `nullified'.

Other important methods of **unique_ptr** pointers include (**T** is the type of the object managed by the pointer):

T* release() — releases the ownership of the managed object, so the receiver object is `nullified'; returns the raw pointer to the managed object (or **nullptr** if the pointer was `empty') – now the caller is responsible for deleting the object in due course;

T* reset(a_pointer = nullptr) — deletes the previously managed object (if there was any) and starts managing the pointer passed as the argument (perhaps, or by default, **nullptr**).

The **reset** method is illustrated by the following program

P168: uniquereset.cpp Resetting *unique_ptr* pointers

```
1. #include <iostream>
2. #include <memory>
3.
4. using std::unique_ptr; using std::ostream; using std::cout;
5.
6. template <typename T>
7. struct Del {
8.     void operator()(T* p) {
9.         cout << "Del deleting " << *p << '\n';
10.        delete p;
11.    }
12. };
13.
14. struct Klazz {
15.     Klazz()  { cout << "Ctor Klazz\n"; }
16.     ~Klazz() { cout << "Dtor Klazz\n"; }
17.     friend ostream& operator<<(ostream& s, const Klazz& k) {
18.         return s << "object of type Klazz";
19.     }
20. };
21.
22. int main() {
23.     std::cout << "Creating  u-pointer\n";
24.     std::unique_ptr<Klazz, Del<Klazz>>
25.                             p(new Klazz{}, Del<Klazz>{});
26.     std::cout << "Resetting u-pointer\n";
27.     p.reset(new Klazz{});
28.     std::cout << "Releasing and deleting\n";
29.     p.reset(); // or reset(nullptr)
30. }
```

As we can see, when an u-pointer is being reset, a new object is constructed and only then the `old' object is destroyed by calling the deleter; for object types, also the destructor will, of course, be invoked.

The u-pointers are often used as elements of collections. The following program demonstrates how we can populate a vector. It also illustrates a very important fact: smart pointers of base classes, exactly as `raw' pointers, may point to objects of derived class and polymorphic invocations work as expected; here, we create a vector with four elements, one of which is an object of the base class **B** and the remaining three of the derived class **D**:

P169: uniquederiv.cpp U-pointers and polymorphism

```
1. #include <iostream>
2. #include <vector>
3. #include <memory>
4.
5. struct B {
6.     virtual void f() { std::cout << "f from B\n"; }
7.     virtual ~B() { }
8. };
9. struct D : B {
```

```
    D() { std::cout << "Ctor D\n"; }
    void f() override { std::cout << "f from D\n"; }
    ~D(){ std::cout << "Dtor D\n"; }
};

int main() {
    {
        std::vector<std::unique_ptr<B>> vec;
        vec.push_back(std::make_unique<B>());
        vec.push_back(std::make_unique<D>());
        vec.emplace_back(std::make_unique<D>());
        std::unique_ptr<B> d{new D};
        vec.push_back(std::move(d));
        for (const auto& up : vec) up->f();
    }
    std::cout << "now vec is out of scope\n";
}
```

Note that when the vector goes out of scope, all its elements release their managed objects, avoiding memory leakage; the program prints

```
Ctor D
Ctor D
Ctor D
f from B
f from D
f from D
f from D
Dtor D
Dtor D
Dtor D
vec out of scope
```

As raw pointers, the u-pointers may also be used in a logical context: if the pointer doesn't manage any object, so `up.get() == nullptr` is true, then it is interpreted as **false**, otherwise – as **true**.

19.6.2 Smart pointers of the *shared_ptr* type

Smart pointers of the **shared_ptr** type (s-pointers for short) can share ownership of the same resource, represented internally by a `raw' pointer. They are equipped with the `reference counting' mechanism — there exist a special data structure associated with s-pointers which keeps record of the number of s-pointers sharing the same managed object. When an s-pointer goes out of scope, the counter is decremented by one and only if it reaches zero, the managed resource is destroyed. Similarly, if we assign two such pointers, `p = q`, the counter associated with the resource managed by ***p*** is decremented (as ***p*** doesn't refer to this object any more), while that associated with the resource managed by ***q*** is incremented, as now also ***p*** refers to it. We can look at the current value of a counter invoking the **use_count** method.

Some of these features are illustrated by the following program:

P170: sharedcount.cpp Counters associated with *shared_ptr* smart pointers.

```
#include <iostream>
#include <memory>
using std::shared_ptr; using std::cout; using std::ostream;

```

```
class Klazz {
    char c;
public:
    Klazz(char c)
        : c{c} { cout << "Ctor " << c << '\n'; }
    ~Klazz()   { cout << "Dtor " << c << '\n'; }
    friend ostream& operator<<(ostream& s, const Klazz& k) {
        return s << k.c;
    }
};

void f(shared_ptr<Klazz> p) {
    cout << "In f: p=" << *p << ", count="
         << p.use_count() << '\n';
}

int main() {
    shared_ptr<Klazz> p = std::make_shared<Klazz>('A');
    shared_ptr<Klazz> q{new Klazz{'B'}};
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    f(p);
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "Now assigning p = q\n";
    p = q;
    cout << "After assignment\n";
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    cout << "Exiting from main\n";
}
```

The output is

```
Ctor A
Ctor B
p=A, count=1
q=B, count=1
In f: p=A, count=2
p=A, count=1
Now assigning p = q
Dtor A
After assignment
p=B, count=2
q=B, count=2
Exiting from main
Dtor B
```

As we can see, we can create an s-pointers by passing to its constructor an ordinary pointer. The default constructor creates an empty s-pointer, analogously to u-pointers. There is also a **make_shared** function, analogous to **make_unique** for u-pointers.
After creating ***p*** and ***q***, they both refer to two different objects, so the counters are both 1. Then we pass ***p*** to the **f** function by value — a copy has to be made which refers to the same object 'A' that ***p*** refers to. Therefore, inside the function, the counter associated with object 'A' is 2. After returning, the copy is removed from the stack, so the counter is decremented back to 1.
Now we assign ***q*** to ***p***: as ***p*** refers to 'A', the counter associated with this object is decremented and reaches zero: it is therefore deleted. Now both ***p*** and ***q*** refer to the same object 'B', so the counter is 2. After exiting from the **main** function, first ***q*** is removed from the stack (decrementing the counter to 1), and then ***p*** — the counter now reaches 0 and the managed object is deleted.

S-pointers support, as do u-pointers, custom deleters provided by the user. Unlike u-pointers, the type of an s-pointer does not depend on the type of the deleter — we can pass a deleter as the additional argument to the constructor. In particular, prior to C++17, if the resource managed by an s-pointer was an array, the default deleter did *not* use **delete[]**

as it should. We had to take care of this ourselves:

P171: shareddelete.cpp Array deleters

```
1. #include <iostream>
2. #include <memory>
3. using std::shared_ptr;
4.
5. template< typename T >
6. struct arrdel {
7.     void operator ()(T const *p) { delete[] p; }
8. };
9.
10. int main() {
11.     shared_ptr<int> sp(new int(1));
12.
13.       // pointer to int[] array - custom deleter
14.     shared_ptr<int> p1(new int[10], arrdel<int>());
15.       // ... or lambda
16.     shared_ptr<int> p2(new int[10'000'000],
17.                        [](int *p) { delete[] p; });
18.       // ... or the one from the library
19.     shared_ptr<int> p3(new int[3]{1, 2, 3},
20.                        std::default_delete<int[]>());
21.     std::cout << p3.get()[2] << " " << *p3 << std::endl; ❶
22.
23.       // since c++17 this will work
24.     shared_ptr<int[]> p4(new int[3]{4, 5, 6});
25.     std::cout << p4[2] << std::endl;
26. }
```

Also, prior to C++17 standard, even for s-pointers managing arrays, the subscript operator (**[]**) was *not* overridden; that's why it was impossible to use it in the line ❶ of the above program. Since the C++17 standard, though, s-pointers may be created to manage arrays and the correct deleter (using **delete[]**) will be used without the user having to provide a custom one. The program, compiled by a compiler supporting C++17 standard, prints

```
3 1
6
```

As the u-pointers, the s-pointers may be used in a logical context (**false** if they are empty, **true** otherwise). They also have the **get** method and overridden ***** and **->** operators.

Prev: 19.5 Move semantics **Up:** 19. Operator overloading. Move semantics **Next:** 20. More on conversions

Subsections

- 19.6.1 Smart pointers of the *unique_ptr* type
- 19.6.2 Smart pointers of the *shared_ptr* type

19.6 Smart pointers

Allocating and releasing memory (with **new**/ **delete**) can be very tedious and error prone. New C++ features, in particular r-values and move semantics, have made it possible to develop an improved versions of the so called smart pointers. These are objects of classes concretized from the **shared_ptr** and **unique_ptr** class templates. Internally, they hold pointers to objects (of any type). Due to appropriate operator overloadings, they syntactically and semantically behave (at least to some extend, and this in particular applies to the **unique_ptr** pointers) as pointers to the objects they refer to. What is important, when variables of these types go out of scope, the appropriate actions are undertaken to release the resources (objects) `owned' by them.

19.6.1 Smart pointers of the *unique_ptr* type

An object of the **unique_ptr** type (let's call them u-pointers, for short) is supposed to be the sole `owner' of the object it manages. When this owner gets destroyed, or is released, or starts owning another object, the object previously owned gets destroyed, and any associated resources are released. Therefore, objects of this type cannot be copied or copy-assigned, as that would lead to a situation when two such smart pointers manage the same resource. However, they can be moved. Then the one from which we move looses ownership (is “nullified”), while the one to which we move gains ownership (and responsibility) of the managed resources (and releases the resources it owned previously).

Smart pointers of the **unique_ptr** type can be created in several ways; for example:

```
std::unique_ptr<int>    empty; // holds nullptr
std::unique_ptr<int>    pi(new int(1));
std::unique_ptr<Person> pp(new Person("Mary", 2001));
std::unique_ptr<int[]>  pa(new int[4]{1,2,2}); // array
*pi = 21;
pp->setName("Kate");
pa[2] = 3;
```

Note that ***pa*** represents a pointer to an array, so **operator[]** is properly overloaded and can be used to access individual elements — the dereference operator ***** and the arrow operator **->**, however, are then not defined. Otherwise, for non-array objects, the smart pointer may be manipulated using syntax and having semantics of a `raw' pointer, but, as they cannot be assigned or copied, they provide neither full valuelike nor pointerlike behaviour. In any case, when the managed array or object is to be destroyed, the correct version of deleting operator will be used — **delete[]** for arrays and **delete** for non-array

objects.
However, if just **delete** or **delete[]** is not appropriate, we can define and pass a custom **deleter**. It has to be a function (generally, a callable) returning nothing and taking a raw pointer of the appropriate type, as in the example below. Note, that the type of a custom deleter must be passed as the second type argument to the **unique_pointer** template to concretize it! In the program below, we use as a deleter an object of our own type **Del** in the first case, and a lambda in the second case

P166: delunique.cpp Custom deleter of *unique_ptr*

```
1. #include <functional>
2. #include <iostream>
3. #include <memory>
4. #include <string>
5.
6. using std::unique_ptr; using std::string;
7.
8. template <typename T>
9. struct Del {
10.     void operator()(T* p) {
11.         std::cout << "Del deleting " << *p << '\n';
12.         delete p;
13.     }
14. };
15.
16. int main() {
17.     {
18.         unique_ptr<string, Del<string>>
19.             us{new string{"Hello"}, Del<string>{}};
20.     }
21.     std::cout << "us is now out of scope\n";
22.
23.     {
24.         unique_ptr<double, std::function<void(double*)>>
25.             ud{new double{7.5},
26.                [](double* p) {
27.                    std::cout << "Deleting " << *p << '\n';
28.                    delete p;
29.                }
30.             };
31.     }
32.     std::cout << "ud is now out of scope\n";
33. }
```

The program prints

```
Del deleting Hello
us is now out of scope
Deleting 7.5
ud is now out of scope
```

To create a **unique_ptr** object, one can also use the **make_unique** function. The objects to be managed by the smart pointer will be created by the function, we only pass initializing values or arguments to a constructor. In case of arrays, only their size can be passed; there is no way to initialize them. Also, a small disadvantage of the **make_unique** function is that you cannot pass your own custom deleter — this is, however, rather seldom useful anyway.

```
std::unique_ptr<int>    pi = std::make_unique<int>(19);
std::unique_ptr<Person> pp =
                std::make_unique<Person>("Mary", 2001);
std::unique_ptr<int[]>  pa = std::make_unique<int[]>(3);
for (int i = 0; i < 3; ++i) pa[i] = i;
```

Smart pointers of the **unique_ptr** type cannot be copied or assigned, as that would violate the uniqueness of ownership. They *can* be moved, though. The output of the following program

P167: ownunique.cpp Passing ownership

```
1.  #include <iostream>
2.  #include <memory>      // smart pointers
3.  #include <string>
4.  #include <utility>     // move
5.
6.  using std::unique_ptr; using std::string;
7.
8.  template <typename T>
9.  struct Del {
10.     void operator()(T* p) {
11.         std::cout << "Del deleting " << *p << '\n';
12.         delete p;
13.     }
14. };
15.
16. template <typename T>
17. void print(const T* p) {
18.     if (p) std::cout << *p << " ";
19.     else   std::cout << "null ";
20. }
21.
22. int main() {
23.     unique_ptr<string, Del<string>>
24.             p1{new string{"abcde"}, Del<string>{}},
25.             p2{new string{"vwxyz"}, Del<string>{}};
26.
27.     print(p1.get()); print(p2.get()); std::cout << '\n'; ❶
28.     std::cout << "Now moving\n";
29.     p1 = std::move(p2);                                  ❷
30.     std::cout << "After moving\n";
31.     print(p1.get()); print(p2.get()); std::cout << '\n'; ❸
32.     std::cout << "Exiting from main\n";
33. }
```

is

```
abcde vwxyz
Now moving
Del deleting abcde
After moving
vwxyz null
Exiting from main
Del deleting vwxyz
```

Note that the **get** method (lines ❶ and ❸) returns the `raw' pointer held by the smart pointer: it shouldn't be assigned to a variable because then the principle of single ownership could easily be violated.
After the move-assignment (line ❷), as we can see from the output, object managed by the ***p1*** pointer is destroyed, the pointer assumes the responsibility for the object managed by ***p2***, and ***p2*** is `nullified'.

Other important methods of **unique_ptr** pointers include (**T** is the type of the object managed by the pointer):

T* release() — releases the ownership of the managed object, so the receiver object is `nullified'; returns the raw pointer to the managed object (or **nullptr** if the pointer was `empty') – now the caller is responsible for deleting the object in due course;

T* reset(a_pointer = nullptr) — deletes the previously managed object (if there was any) and starts managing the pointer passed as the argument (perhaps, or by default, **nullptr**).

The **reset** method is illustrated by the following program

P168: uniquereset.cpp Resetting *unique_ptr* pointers

```
1. #include <iostream>
2. #include <memory>
3.
4. using std::unique_ptr; using std::ostream; using std::cout;
5.
6. template <typename T>
7. struct Del {
8.     void operator()(T* p) {
9.         cout << "Del deleting " << *p << '\n';
10.        delete p;
11.    }
12. };
13.
14. struct Klazz {
15.     Klazz()  { cout << "Ctor Klazz\n"; }
16.     ~Klazz() { cout << "Dtor Klazz\n"; }
17.     friend ostream& operator<<(ostream& s, const Klazz& k) {
18.         return s << "object of type Klazz";
19.     }
20. };
21.
22. int main() {
23.     std::cout << "Creating  u-pointer\n";
24.     std::unique_ptr<Klazz, Del<Klazz>>
25.                             p(new Klazz{}, Del<Klazz>{});
26.     std::cout << "Resetting u-pointer\n";
27.     p.reset(new Klazz{});
28.     std::cout << "Releasing and deleting\n";
29.     p.reset(); // or reset(nullptr)
30. }
```

As we can see, when an u-pointer is being reset, a new object is constructed and only then the `old' object is destroyed by calling the deleter; for object types, also the destructor will, of course, be invoked.

The u-pointers are often used as elements of collections. The following program demonstrates how we can populate a vector. It also illustrates a very important fact: smart pointers of base classes, exactly as `raw' pointers, may point to objects of derived class and polymorphic invocations work as expected; here, we create a vector with four elements, one of which is an object of the base class **B** and the remaining three of the derived class **D**:

P169: uniquederiv.cpp U-pointers and polymorphism

```
1. #include <iostream>
2. #include <vector>
3. #include <memory>
4.
5. struct B {
6.     virtual void f() { std::cout << "f from B\n"; }
7.     virtual ~B() { }
8. };
9. struct D : B {
```

```
    D() { std::cout << "Ctor D\n"; }
    void f() override { std::cout << "f from D\n"; }
    ~D(){ std::cout << "Dtor D\n"; }
};

int main() {
    {
        std::vector<std::unique_ptr<B>> vec;
        vec.push_back(std::make_unique<B>());
        vec.push_back(std::make_unique<D>());
        vec.emplace_back(std::make_unique<D>());
        std::unique_ptr<B> d{new D};
        vec.push_back(std::move(d));
        for (const auto& up : vec) up->f();
    }
    std::cout << "now vec is out of scope\n";
}
```

Note that when the vector goes out of scope, all its elements release their managed objects, avoiding memory leakage; the program prints

```
Ctor D
Ctor D
Ctor D
f from B
f from D
f from D
f from D
Dtor D
Dtor D
Dtor D
vec out of scope
```

As raw pointers, the u-pointers may also be used in a logical context: if the pointer doesn't manage any object, so `up.get() == nullptr` is true, then it is interpreted as **false**, otherwise – as **true**.

19.6.2 Smart pointers of the *shared_ptr* type

Smart pointers of the **shared_ptr** type (s-pointers for short) can share ownership of the same resource, represented internally by a `raw' pointer. They are equipped with the `reference counting' mechanism — there exist a special data structure associated with s-pointers which keeps record of the number of s-pointers sharing the same managed object. When an s-pointer goes out of scope, the counter is decremented by one and only if it reaches zero, the managed resource is destroyed. Similarly, if we assign two such pointers, `p = q`, the counter associated with the resource managed by ***p*** is decremented (as ***p*** doesn't refer to this object any more), while that associated with the resource managed by ***q*** is incremented, as now also ***p*** refers to it. We can look at the current value of a counter invoking the **use_count** method.

Some of these features are illustrated by the following program:

P170: sharedcount.cpp Counters associated with *shared_ptr* smart pointers.

```
#include <iostream>
#include <memory>
using std::shared_ptr; using std::cout; using std::ostream;

```

```
class Klazz {
    char c;
public:
    Klazz(char c)
        : c{c} { cout << "Ctor " << c << '\n'; }
    ~Klazz()   { cout << "Dtor " << c << '\n'; }
    friend ostream& operator<<(ostream& s, const Klazz& k) {
        return s << k.c;
    }
};

void f(shared_ptr<Klazz> p) {
    cout << "In f: p=" << *p << ", count="
         << p.use_count() << '\n';
}

int main() {
    shared_ptr<Klazz> p = std::make_shared<Klazz>('A');
    shared_ptr<Klazz> q{new Klazz{'B'}};
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    f(p);
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "Now assigning p = q\n";
    p = q;
    cout << "After assignment\n";
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    cout << "Exiting from main\n";
}
```

The output is

```
Ctor A
Ctor B
p=A, count=1
q=B, count=1
In f: p=A, count=2
p=A, count=1
Now assigning p = q
Dtor A
After assignment
p=B, count=2
q=B, count=2
Exiting from main
Dtor B
```

As we can see, we can create an s-pointers by passing to its constructor an ordinary pointer. The default constructor creates an empty s-pointer, analogously to u-pointers. There is also a **make_shared** function, analogous to **make_unique** for u-pointers.
After creating ***p*** and ***q***, they both refer to two different objects, so the counters are both 1. Then we pass ***p*** to the **f** function by value — a copy has to be made which refers to the same object 'A' that ***p*** refers to. Therefore, inside the function, the counter associated with object 'A' is 2. After returning, the copy is removed from the stack, so the counter is decremented back to 1.
Now we assign ***q*** to ***p***: as ***p*** refers to 'A', the counter associated with this object is decremented and reaches zero: it is therefore deleted. Now both ***p*** and ***q*** refer to the same object 'B', so the counter is 2. After exiting from the **main** function, first ***q*** is removed from the stack (decrementing the counter to 1), and then ***p*** — the counter now reaches 0 and the managed object is deleted.

S-pointers support, as do u-pointers, custom deleters provided by the user. Unlike u-pointers, the type of an s-pointer does not depend on the type of the deleter — we can pass a deleter as the additional argument to the constructor. In particular, prior to C++17, if the resource managed by an s-pointer was an array, the default deleter did *not* use **delete[]**

as it should. We had to take care of this ourselves:

P171: shareddelete.cpp Array deleters

```
1. #include <iostream>
2. #include <memory>
3. using std::shared_ptr;
4.
5. template< typename T >
6. struct arrdel {
7.     void operator ()(T const *p) { delete[] p; }
8. };
9.
10. int main() {
11.     shared_ptr<int> sp(new int(1));
12.
13.       // pointer to int[] array - custom deleter
14.     shared_ptr<int> p1(new int[10], arrdel<int>());
15.       // ... or lambda
16.     shared_ptr<int> p2(new int[10'000'000],
17.                        [](int *p) { delete[] p; });
18.       // ... or the one from the library
19.     shared_ptr<int> p3(new int[3]{1, 2, 3},
20.                        std::default_delete<int[]>());
21.     std::cout << p3.get()[2] << " " << *p3 << std::endl; ❶
22.
23.       // since c++17 this will work
24.     shared_ptr<int[]> p4(new int[3]{4, 5, 6});
25.     std::cout << p4[2] << std::endl;
26. }
```

Also, prior to C++17 standard, even for s-pointers managing arrays, the subscript operator (**[]**) was *not* overridden; that's why it was impossible to use it in the line ❶ of the above program. Since the C++17 standard, though, s-pointers may be created to manage arrays and the correct deleter (using **delete[]**) will be used without the user having to provide a custom one. The program, compiled by a compiler supporting C++17 standard, prints

```
3 1
6
```

As the u-pointers, the s-pointers may be used in a logical context (**false** if they are empty, **true** otherwise). They also have the **get** method and overridden ***** and **->** operators.

Prev: 19.5 Move semantics **Up:** 19. Operator overloading. Move semantics **Next:** 20. More on conversions

20. More on conversions

We have been mentioning conversions many times in this course. In section on conversions we described standard conversions which are performed automatically between built-in types; now we will be talking about conversions between types defined in a program (or a library) and about explicit conversions, forced programatically.

Subsections

- 20.1 Conversions to and from a user-defined type
  - 20.1.1 Conversions *to* a defined type
  - 20.1.2 Conversions *from* user-defined type
- 20.2 Explicit conversions
  - 20.2.1 *const_cast* conversions
  - 20.2.2 Static casts
  - 20.2.3 Dynamic conversions
  - 20.2.4 Forced conversions

Subsections

- 20.1.1 Conversions *to* a defined type
- 20.1.2 Conversions *from* user-defined type

20.1 Conversions to and from a user-defined type

Defining classes, we define new data types. It is then sometimes convenient to define ways in which values of variables of these new types can be converted into values of other types — built-in or also of our own production — and the other way around: from values of other types to a value of a type that we are defining.

20.1.1 Conversions *to* a defined type

Suppose we have a class **A** with a constructor which can be invoked with one argument of type **B**. It can be a one-parameter constructor, or many parameter one, but with default values defined for the rest of arguments. Type **B** can be a built-in or a user defined type.

Now suppose we have used an object ***b*** of type **B** in a context where an object of type **A** is required. If there is a constructor in class **A** which can be called with one argument of type **B**, then conversion **B** → **A** will be performed by creating an object of type **A** using this constructor with argument ***b***.

For example, if there is a function with a parameter of type **A**

```
void fun(A a) { ... }
```

and we call it with a **double** as an argument, `fun(4.25)`, then the types do not agree and the program is in error. However, there will be no error if class **A** defines a constructor which can be invoked with one argument of type **double**. This constructor will be used as a **conversion constructor**: an object of type **A** will be created using this constructor with 4.25 as argument and passed to the function **fun**.

What will happen if we replace argument 4.25 by an integer value, e.g., if we call `fun(4)`? This call will also be valid! True, there is no direct conversion **int** → **A**, because there is no constructor in **A** which can take one argument of type **int**. However, there *is* a standard conversion **int** → **double**, and there is a non-standard (defined by a conversion constructor) conversion **double** → **A**. Therefore, a two-step conversion will take place here: first a temporary **double** will be created from the **int** value, and then this **double** will be used in conversion constructor of **A** to produce a value of type **A**. A sequence of conversions leading to a target type could be even longer; however

only one of the conversions in such a sequence can be a user-

defined conversion; all the remaining have to be standard conversions.

Allowing more than one user-defined conversion in such sequences could leave us with so many possible paths of conversions leading to the same type that it would be impossible to keep control over the program.

Sometimes even one such conversion is too much. Most of the times, if we define in a class a one-parameter constructor, we do it because it is convenient as a normal constructor: we do not want it to play the rôle of a conversion constructor. However, if a one parameter constructor exists, it *will* be used for conversions, sometimes unexpectedly and not at all according to our intentions. To avoid such situations, there is a special keyword, **explicit**, which can be used as a modifier in declaration of a constructor which can be invoked with one argument. A constructor which is **explicit** can be used to create objects in a usual way, but it will never be used as a conversion constructor.

Let us look at an example:

P172: convto.cpp Conversion to user-defined type

```
1. #include <iostream>
2. using namespace std;
3.
4. struct Point {
5.     int x, y;
6.     Point(int x = 0, int y = 0) : x(x), y(y) { }
7. };
8.
9. struct Segment {
10.     Point A, B;
11.     // explicit
12.     Segment(Point A = Point(), Point B = Point())
13.         : A(A), B(B)
14.     { }
15. };
16.
17. void showPoint(Point A) {
18.     cout << "Point[" << A.x << "," << A.y << "]";
19. }
20.
21. void showSegment(Segment AB) {
22.     cout << "Segment: ";
23.     showPoint(AB.A);
24.     cout << "--";
25.     showPoint(AB.B);
26.     cout << endl;
27. }
28.
29. int main() {
30.     int k = 7;
31.     showPoint(k);
32.
33.     cout << endl;
34.
35.     Point A(1,1);
36.     showSegment(A);
37.     // showSegment(k);
38. }
```

Two classes are defined here: **Point** and **Segment**. Both have constructors which can be called with one argument. Actually, they are not one-parameter constructors, but the

presence of default arguments (see section on default arguments) makes them suitable for playing the rôle of conversion constructors. Class **Point** has a constructor which can be called with one **int** argument, while **Segment** — with one argument of type **Point**.

Now look at functions **showPoint** and **showSegment**: they accept arguments of types **Point** and **Segment**, respectively. In function **main**, on line 31, we call **showPoint** with argument of type **int**. There is no function of this name which could accept an **int**. However, there is a function named **showPoint** which accepts arguments of type **Point**. Therefore, the compiler will check if an **int** can be converted into **Point** by using a conversion constructor **int** → **Point**. Indeed, this is possible, because such a constructor exists; consequently, the call is valid and the conversion will be performed yielding a temporary object of type **Point** which in turn will be passed to the function **showPoint**, as can be seen from the first line of the output:

```
Point[7,0]
Segment: Point[1,1]--Point[0,0]
```

Now, on line 35, we create an object ***A*** of type **Point** and pass it to function **showSegment** which expects an object of class **Segment**. Again, the call is valid because there *is* a conversion constructor **Point** → **Segment** defined in class **Segment**. This can be seen from the second line of the output.

However, uncommenting the call on line 37 would lead to an erroneous program. Here we pass an **int** to a function which expects a **Segment**. Such a call to be valid would require a two-step conversion: first **int** → **Point**, and then **Point** → **Segment**. Both conversions exist, provided by constructors in classes **Point** and **Segment**, but cannot be used here, because it would lead to a sequence of conversions containing more than one non-standard conversion, what is illegal.

Let us now try to uncomment line 11. That will make the constructor of **Segment** explicit. Hence, it cannot play the rôle of a conversion constructor any more. The call from line 36, which requires a conversion **Point** → **Segment**, becomes illegal:

```
cpp> g++ -Wall -pedantic-errors convto.cpp
convto.cpp: In function `int main()':
convto.cpp:36: error: conversion from `Point' to
       non-scalar type `Segment' requested
```

As the second example, let us consider another version of class **Modulo** which has already been introduced in programs **modsev.cpp** and **modsev1.cpp**. We will see that a conversion constructor can be useful for operator overloading. It will be used here for overloading addition operator of numbers of type **Modulo**.

P173: modcon.cpp Using conversions for operator overloading

```
1. #include <iostream>
2. using namespace std;
3.
4. class Modulo {
5.     int numb;
6. public:
7.     static int modul;
8.
9.     Modulo() : numb(0) { }
10.
11.     Modulo(int numb) : numb(numb%modul) { }
12.
```

```
13.     friend Modulo operator+(Modulo,Modulo);
14.     friend ostream& operator<<(ostream&,const Modulo&);
15. };
16. int Modulo::modul = 7;
17.
18. Modulo operator+(Modulo m, Modulo n) {
19.     return Modulo(m.numb + n.numb);
20. }
21.
22. ostream& operator<<(ostream& str, const Modulo& m) {
23.     return str << m.numb;
24. }
25.
26. int main() {
27.
28.     Modulo m(5), n(6), k;
29.
30.     k = m + n;
31.     cout << "m + n (mod " << Modulo::modul
32.          << ") = "        << k << endl;
33.
34.     k = m + 6;
35.     cout << "m + 6 (mod " << Modulo::modul
36.          << ") = "        << k << endl;
37.
38.     k = 6 + m;
39.     cout << "6 + m (mod " << Modulo::modul
40.          << ") = "        << k << endl;
41. }
```

This time we overload addition of objects of type **Modulo** by a friend global function, not a method of the class. Note that we have defined only one such function (lines 18-20): with both parameters of type **Modulo**. But we can see in **main** that the function can be used for adding two objects of class **Modulo** (line 30), for adding an **int** to object of class **Modulo** (line 34) and also for adding an object of class **Modulo** to an **int** (line 38). In all cases the addition works as expected:

```
m + n (mod 7) = 4
m + 6 (mod 7) = 4
6 + m (mod 7) = 4
```

Note that the last form, *number+object*, would be impossible to implement as a method, because we have a number on the left-hand side of the operator. How can it be that one function, with both parameters of type **Modulo**, can handle all three cases? The class has a conversion constructor **int** → **Modulo** (line 11). It is therefore sufficient to have only one argument of type **Modulo** — the second, be it the left or the right one, can be an **int**: it will be converted to type **Modulo** automatically.

20.1.2 Conversions *from* user-defined type

Sometimes we would like to define a conversions in the opposite direction: from a user-defined type to another type. Suppose, for example, that we would like to have conversion **Modulo** → **int**. We cannot do it by defining a conversion constructor in class **int** for several reasons: we do not have access to the definition of the target type and, last but not least, the target type is not a class at all...Even if the target type is a class, it can be defined in a library that we do not want or are not able to modify.

In these situations, we can define a **conversion method** in a class *from* which a conversion is required. It should be a parameterless method named ' **operator Type**', where **Type** is the name of a target type — built-in or also a user-defined one.

When declaring/defining a conversion method, one *must not* specify any return type. This is logical, because a return type is already determined by the name of the method itself. Still, the method has to terminate with a **return** statement returning a value of the target type. As all other methods, a conversion method is always invoked on an object: its task is to return a value of the target type corresponding to *this* object. Of couse, the source object should not be modified, so usually we declare such methods as **const**. We can also declare them as **explicit** (as converting constructors) to avoid accidental, unwanted conversions, but then we have to use explicit conversions where they are needed.

Let us consider an example similar to that from program convto.cpp:

P174: convfrom.cpp Conversion from user-defined type

```
1.  #include <iostream>
2.  #include <cmath>
3.  using std::cout; using std::endl;
4.
5.  struct Point {
6.      double x, y;
7.      Point(double x = 0, double y = 0) : x(x), y(y) { }
8.      operator double() const {          ❶
9.          return std::sqrt(x*x+y*y);
10.     }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.     Segment(Point A = Point(), Point B = Point())
16.         : A(A), B(B)
17.     { }
18.     operator Point() const {           ❷
19.         return Point( (A.x+B.x)/2, (A.y+B.y)/2 );
20.     }
21. };
22.
23. void showPoint(Point A) {
24.     cout << "Point[" << A.x << "," << A.y << "]";
25. }
26.
27. void showSegment(Segment AB) {
28.     cout << "Segment: ";
29.     showPoint(AB.A);
30.     cout << "--";
31.     showPoint(AB.B);
32.     cout << endl;
33. }
34.
35. void showDouble(double d) {
36.     cout << "Double " << d;
37. }
38.
39. int main() {
40.     Point A(3,4);
41.     showPoint(A);                      ❸
42.     cout << endl;
43.     showDouble(A);                     ❹
44.
45.     cout << endl;
46.
47.     Segment BC(Point(1,1),Point(3,3));
48.     showSegment(BC);
49.     showPoint(BC);                     ❺
50.
51.     cout << endl;
52. }
```

Class **Point** has been equipped with a conversion method **operator double()** (❶). It returns the distance of *this* point from the origin as a value of type **double** (i.e., of a built-in type). Similarly, we have added to class **Segment** a method converting segments to points (❷). It returns an object representing the geometrical center of *this* segment. Finally, we have added a function **showDouble**, which just prints the value passed to it through the argument.

In line ❸, function **showPoint** is called with argument of type **Point**, i.e., of the declared type of its parameter. Immediately after that we call function **showDouble** with the same argument of type **Point** (❹). The function expects a **double**, so a conversion **Point** → **double** is needed — such conversion is indeed defined by the conversion method in class **Point**. Therefore, the call is valid, what can be seen from the second line of the output:

```
Point[3,4]
Double 5
Segment: Point[1,1]--Point[3,3]
Point[2,2]
```

In line ❺, the situation is similar. This time we pass a segment to a function accepting points. Again, a conversion will take place, this time it will be a conversion **Segment** → **Point** (❷).

Note that in the second case we could have defined a conversion *constructor* in class **Point** taking a point as an argument: the effect would be equivalent to that of defining conversion *method* in class **Segment**. However, in the first case we had no choice: it is impossible to define a conversion constructor in class **double** (mainly because there is no such class...).

Conversion methods can be inherited and can be virtual — the meaning of this statement will become clear in one of the next chapters.

Prev: 20. More on conversions **Up:** 20. More on conversions **Next:** 20.2 Explicit conversions

Subsections

- 20.1.1 Conversions *to* a defined type
- 20.1.2 Conversions *from* user-defined type

20.1 Conversions to and from a user-defined type

Defining classes, we define new data types. It is then sometimes convenient to define ways in which values of variables of these new types can be converted into values of other types — built-in or also of our own production — and the other way around: from values of other types to a value of a type that we are defining.

20.1.1 Conversions *to* a defined type

Suppose we have a class **A** with a constructor which can be invoked with one argument of type **B**. It can be a one-parameter constructor, or many parameter one, but with default values defined for the rest of arguments. Type **B** can be a built-in or a user defined type.

Now suppose we have used an object ***b*** of type **B** in a context where an object of type **A** is required. If there is a constructor in class **A** which can be called with one argument of type **B**, then conversion **B** → **A** will be performed by creating an object of type **A** using this constructor with argument ***b***.

For example, if there is a function with a parameter of type **A**

```
void fun(A a) { ... }
```

and we call it with a **double** as an argument, `fun(4.25)`, then the types do not agree and the program is in error. However, there will be no error if class **A** defines a constructor which can be invoked with one argument of type **double**. This constructor will be used as a **conversion constructor**: an object of type **A** will be created using this constructor with 4.25 as argument and passed to the function **fun**.

What will happen if we replace argument 4.25 by an integer value, e.g., if we call `fun(4)`? This call will also be valid! True, there is no direct conversion **int** → **A**, because there is no constructor in **A** which can take one argument of type **int**. However, there *is* a standard conversion **int** → **double**, and there is a non-standard (defined by a conversion constructor) conversion **double** → **A**. Therefore, a two-step conversion will take place here: first a temporary **double** will be created from the **int** value, and then this **double** will be used in conversion constructor of **A** to produce a value of type **A**. A sequence of conversions leading to a target type could be even longer; however

only one of the conversions in such a sequence can be a user-

defined conversion; all the remaining have to be standard conversions.

Allowing more than one user-defined conversion in such sequences could leave us with so many possible paths of conversions leading to the same type that it would be impossible to keep control over the program.

Sometimes even one such conversion is too much. Most of the times, if we define in a class a one-parameter constructor, we do it because it is convenient as a normal constructor: we do not want it to play the rôle of a conversion constructor. However, if a one parameter constructor exists, it *will* be used for conversions, sometimes unexpectedly and not at all according to our intentions. To avoid such situations, there is a special keyword, **explicit**, which can be used as a modifier in declaration of a constructor which can be invoked with one argument. A constructor which is **explicit** can be used to create objects in a usual way, but it will never be used as a conversion constructor.

Let us look at an example:

P172: convto.cpp Conversion to user-defined type

```
1. #include <iostream>
2. using namespace std;
3.
4. struct Point {
5.     int x, y;
6.     Point(int x = 0, int y = 0) : x(x), y(y) { }
7. };
8.
9. struct Segment {
10.     Point A, B;
11.     // explicit
12.     Segment(Point A = Point(), Point B = Point())
13.         : A(A), B(B)
14.     { }
15. };
16.
17. void showPoint(Point A) {
18.     cout << "Point[" << A.x << "," << A.y << "]";
19. }
20.
21. void showSegment(Segment AB) {
22.     cout << "Segment: ";
23.     showPoint(AB.A);
24.     cout << "--";
25.     showPoint(AB.B);
26.     cout << endl;
27. }
28.
29. int main() {
30.     int k = 7;
31.     showPoint(k);
32.
33.     cout << endl;
34.
35.     Point A(1,1);
36.     showSegment(A);
37.     // showSegment(k);
38. }
```

Two classes are defined here: **Point** and **Segment**. Both have constructors which can be called with one argument. Actually, they are not one-parameter constructors, but the

presence of default arguments (see section on default arguments) makes them suitable for playing the rôle of conversion constructors. Class **Point** has a constructor which can be called with one **int** argument, while **Segment** — with one argument of type **Point**.

Now look at functions **showPoint** and **showSegment**: they accept arguments of types **Point** and **Segment**, respectively. In function **main**, on line 31, we call **showPoint** with argument of type **int**. There is no function of this name which could accept an **int**. However, there is a function named **showPoint** which accepts arguments of type **Point**. Therefore, the compiler will check if an **int** can be converted into **Point** by using a conversion constructor **int** → **Point**. Indeed, this is possible, because such a constructor exists; consequently, the call is valid and the conversion will be performed yielding a temporary object of type **Point** which in turn will be passed to the function **showPoint**, as can be seen from the first line of the output:

```
Point[7,0]
Segment: Point[1,1]--Point[0,0]
```

Now, on line 35, we create an object ***A*** of type **Point** and pass it to function **showSegment** which expects an object of class **Segment**. Again, the call is valid because there *is* a conversion constructor **Point** → **Segment** defined in class **Segment**. This can be seen from the second line of the output.

However, uncommenting the call on line 37 would lead to an erroneous program. Here we pass an **int** to a function which expects a **Segment**. Such a call to be valid would require a two-step conversion: first **int** → **Point**, and then **Point** → **Segment**. Both conversions exist, provided by constructors in classes **Point** and **Segment**, but cannot be used here, because it would lead to a sequence of conversions containing more than one non-standard conversion, what is illegal.

Let us now try to uncomment line 11. That will make the constructor of **Segment** explicit. Hence, it cannot play the rôle of a conversion constructor any more. The call from line 36, which requires a conversion **Point** → **Segment**, becomes illegal:

```
cpp> g++ -Wall -pedantic-errors convto.cpp
convto.cpp: In function `int main()':
convto.cpp:36: error: conversion from `Point' to
       non-scalar type `Segment' requested
```

As the second example, let us consider another version of class **Modulo** which has already been introduced in programs **modsev.cpp** and **modsev1.cpp**. We will see that a conversion constructor can be useful for operator overloading. It will be used here for overloading addition operator of numbers of type **Modulo**.

P173: modcon.cpp Using conversions for operator overloading

```
1. #include <iostream>
2. using namespace std;
3.
4. class Modulo {
5.     int numb;
6. public:
7.     static int modul;
8.
9.     Modulo() : numb(0) { }
10.
11.     Modulo(int numb) : numb(numb%modul) { }
12.
```

```
13.     friend Modulo operator+(Modulo,Modulo);
14.     friend ostream& operator<<(ostream&,const Modulo&);
15. };
16. int Modulo::modul = 7;
17.
18. Modulo operator+(Modulo m, Modulo n) {
19.     return Modulo(m.numb + n.numb);
20. }
21.
22. ostream& operator<<(ostream& str, const Modulo& m) {
23.     return str << m.numb;
24. }
25.
26. int main() {
27.
28.     Modulo m(5), n(6), k;
29.
30.     k = m + n;
31.     cout << "m + n (mod " << Modulo::modul
32.          << ") = "        << k << endl;
33.
34.     k = m + 6;
35.     cout << "m + 6 (mod " << Modulo::modul
36.          << ") = "        << k << endl;
37.
38.     k = 6 + m;
39.     cout << "6 + m (mod " << Modulo::modul
40.          << ") = "        << k << endl;
41. }
```

This time we overload addition of objects of type **Modulo** by a friend global function, not a method of the class. Note that we have defined only one such function (lines 18-20): with both parameters of type **Modulo**. But we can see in **main** that the function can be used for adding two objects of class **Modulo** (line 30), for adding an **int** to object of class **Modulo** (line 34) and also for adding an object of class **Modulo** to an **int** (line 38). In all cases the addition works as expected:

```
m + n (mod 7) = 4
m + 6 (mod 7) = 4
6 + m (mod 7) = 4
```

Note that the last form, *number+object*, would be impossible to implement as a method, because we have a number on the left-hand side of the operator. How can it be that one function, with both parameters of type **Modulo**, can handle all three cases? The class has a conversion constructor **int** → **Modulo** (line 11). It is therefore sufficient to have only one argument of type **Modulo** — the second, be it the left or the right one, can be an **int**: it will be converted to type **Modulo** automatically.

20.1.2 Conversions *from* user-defined type

Sometimes we would like to define a conversions in the opposite direction: from a user-defined type to another type. Suppose, for example, that we would like to have conversion **Modulo** → **int**. We cannot do it by defining a conversion constructor in class **int** for several reasons: we do not have access to the definition of the target type and, last but not least, the target type is not a class at all...Even if the target type is a class, it can be defined in a library that we do not want or are not able to modify.

In these situations, we can define a **conversion method** in a class *from* which a conversion is required. It should be a parameterless method named ' **operator Type**', where **Type** is the name of a target type — built-in or also a user-defined one.

When declaring/defining a conversion method, one *must not* specify any return type. This is logical, because a return type is already determined by the name of the method itself. Still, the method has to terminate with a **return** statement returning a value of the target type. As all other methods, a conversion method is always invoked on an object: its task is to return a value of the target type corresponding to *this* object. Of couse, the source object should not be modified, so usually we declare such methods as **const**. We can also declare them as **explicit** (as converting constructors) to avoid accidental, unwanted conversions, but then we have to use explicit conversions where they are needed.

Let us consider an example similar to that from program convto.cpp:

P174: convfrom.cpp Conversion from user-defined type

```
1.  #include <iostream>
2.  #include <cmath>
3.  using std::cout; using std::endl;
4.
5.  struct Point {
6.      double x, y;
7.      Point(double x = 0, double y = 0) : x(x), y(y) { }
8.      operator double() const {          ❶
9.          return std::sqrt(x*x+y*y);
10.     }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.     Segment(Point A = Point(), Point B = Point())
16.         : A(A), B(B)
17.     { }
18.     operator Point() const {           ❷
19.         return Point( (A.x+B.x)/2, (A.y+B.y)/2 );
20.     }
21. };
22.
23. void showPoint(Point A) {
24.     cout << "Point[" << A.x << "," << A.y << "]";
25. }
26.
27. void showSegment(Segment AB) {
28.     cout << "Segment: ";
29.     showPoint(AB.A);
30.     cout << "--";
31.     showPoint(AB.B);
32.     cout << endl;
33. }
34.
35. void showDouble(double d) {
36.     cout << "Double " << d;
37. }
38.
39. int main() {
40.     Point A(3,4);
41.     showPoint(A);                      ❸
42.     cout << endl;
43.     showDouble(A);                     ❹
44.
45.     cout << endl;
46.
47.     Segment BC(Point(1,1),Point(3,3));
48.     showSegment(BC);
49.     showPoint(BC);                     ❺
50.
51.     cout << endl;
52. }
```

Class **Point** has been equipped with a conversion method **operator double()** (❶). It returns the distance of *this* point from the origin as a value of type **double** (i.e., of a built-in type). Similarly, we have added to class **Segment** a method converting segments to points (❷). It returns an object representing the geometrical center of *this* segment. Finally, we have added a function **showDouble**, which just prints the value passed to it through the argument.

In line ❸, function **showPoint** is called with argument of type **Point**, i.e., of the declared type of its parameter. Immediately after that we call function **showDouble** with the same argument of type **Point** (❹). The function expects a **double**, so a conversion **Point** → **double** is needed — such conversion is indeed defined by the conversion method in class **Point**. Therefore, the call is valid, what can be seen from the second line of the output:

```
Point[3,4]
Double 5
Segment: Point[1,1]--Point[3,3]
Point[2,2]
```

In line ❺, the situation is similar. This time we pass a segment to a function accepting points. Again, a conversion will take place, this time it will be a conversion **Segment** → **Point** (❷).

Note that in the second case we could have defined a conversion *constructor* in class **Point** taking a point as an argument: the effect would be equivalent to that of defining conversion *method* in class **Segment**. However, in the first case we had no choice: it is impossible to define a conversion constructor in class **double** (mainly because there is no such class...).

Conversion methods can be inherited and can be virtual — the meaning of this statement will become clear in one of the next chapters.

Prev: 20. More on conversions **Up:** 20. More on conversions **Next:** 20.2 Explicit conversions

Subsections

- 20.1.1 Conversions *to* a defined type
- 20.1.2 Conversions *from* user-defined type

20.1 Conversions to and from a user-defined type

Defining classes, we define new data types. It is then sometimes convenient to define ways in which values of variables of these new types can be converted into values of other types — built-in or also of our own production — and the other way around: from values of other types to a value of a type that we are defining.

20.1.1 Conversions *to* a defined type

Suppose we have a class **A** with a constructor which can be invoked with one argument of type **B**. It can be a one-parameter constructor, or many parameter one, but with default values defined for the rest of arguments. Type **B** can be a built-in or a user defined type.

Now suppose we have used an object ***b*** of type **B** in a context where an object of type **A** is required. If there is a constructor in class **A** which can be called with one argument of type **B**, then conversion **B** → **A** will be performed by creating an object of type **A** using this constructor with argument ***b***.

For example, if there is a function with a parameter of type **A**

```
void fun(A a) { ... }
```

and we call it with a **double** as an argument, `fun(4.25)`, then the types do not agree and the program is in error. However, there will be no error if class **A** defines a constructor which can be invoked with one argument of type **double**. This constructor will be used as a **conversion constructor**: an object of type **A** will be created using this constructor with 4.25 as argument and passed to the function **fun**.

What will happen if we replace argument 4.25 by an integer value, e.g., if we call `fun(4)`? This call will also be valid! True, there is no direct conversion **int** → **A**, because there is no constructor in **A** which can take one argument of type **int**. However, there *is* a standard conversion **int** → **double**, and there is a non-standard (defined by a conversion constructor) conversion **double** → **A**. Therefore, a two-step conversion will take place here: first a temporary **double** will be created from the **int** value, and then this **double** will be used in conversion constructor of **A** to produce a value of type **A**. A sequence of conversions leading to a target type could be even longer; however

only one of the conversions in such a sequence can be a user-

defined conversion; all the remaining have to be standard conversions.

Allowing more than one user-defined conversion in such sequences could leave us with so many possible paths of conversions leading to the same type that it would be impossible to keep control over the program.

Sometimes even one such conversion is too much. Most of the times, if we define in a class a one-parameter constructor, we do it because it is convenient as a normal constructor: we do not want it to play the rôle of a conversion constructor. However, if a one parameter constructor exists, it *will* be used for conversions, sometimes unexpectedly and not at all according to our intentions. To avoid such situations, there is a special keyword, **explicit**, which can be used as a modifier in declaration of a constructor which can be invoked with one argument. A constructor which is **explicit** can be used to create objects in a usual way, but it will never be used as a conversion constructor.

Let us look at an example:

P172: convto.cpp Conversion to user-defined type

```
1. #include <iostream>
2. using namespace std;
3.
4. struct Point {
5.     int x, y;
6.     Point(int x = 0, int y = 0) : x(x), y(y) { }
7. };
8.
9. struct Segment {
10.     Point A, B;
11.     // explicit
12.     Segment(Point A = Point(), Point B = Point())
13.         : A(A), B(B)
14.     { }
15. };
16.
17. void showPoint(Point A) {
18.     cout << "Point[" << A.x << "," << A.y << "]";
19. }
20.
21. void showSegment(Segment AB) {
22.     cout << "Segment: ";
23.     showPoint(AB.A);
24.     cout << "--";
25.     showPoint(AB.B);
26.     cout << endl;
27. }
28.
29. int main() {
30.     int k = 7;
31.     showPoint(k);
32.
33.     cout << endl;
34.
35.     Point A(1,1);
36.     showSegment(A);
37.     // showSegment(k);
38. }
```

Two classes are defined here: **Point** and **Segment**. Both have constructors which can be called with one argument. Actually, they are not one-parameter constructors, but the

presence of default arguments (see section on default arguments) makes them suitable for playing the rôle of conversion constructors. Class **Point** has a constructor which can be called with one **int** argument, while **Segment** — with one argument of type **Point**.

Now look at functions **showPoint** and **showSegment**: they accept arguments of types **Point** and **Segment**, respectively. In function **main**, on line 31, we call **showPoint** with argument of type **int**. There is no function of this name which could accept an **int**. However, there is a function named **showPoint** which accepts arguments of type **Point**. Therefore, the compiler will check if an **int** can be converted into **Point** by using a conversion constructor **int** → **Point**. Indeed, this is possible, because such a constructor exists; consequently, the call is valid and the conversion will be performed yielding a temporary object of type **Point** which in turn will be passed to the function **showPoint**, as can be seen from the first line of the output:

```
Point[7,0]
Segment: Point[1,1]--Point[0,0]
```

Now, on line 35, we create an object ***A*** of type **Point** and pass it to function **showSegment** which expects an object of class **Segment**. Again, the call is valid because there *is* a conversion constructor **Point** → **Segment** defined in class **Segment**. This can be seen from the second line of the output.

However, uncommenting the call on line 37 would lead to an erroneous program. Here we pass an **int** to a function which expects a **Segment**. Such a call to be valid would require a two-step conversion: first **int** → **Point**, and then **Point** → **Segment**. Both conversions exist, provided by constructors in classes **Point** and **Segment**, but cannot be used here, because it would lead to a sequence of conversions containing more than one non-standard conversion, what is illegal.

Let us now try to uncomment line 11. That will make the constructor of **Segment** explicit. Hence, it cannot play the rôle of a conversion constructor any more. The call from line 36, which requires a conversion **Point** → **Segment**, becomes illegal:

```
cpp> g++ -Wall -pedantic-errors convto.cpp
convto.cpp: In function `int main()':
convto.cpp:36: error: conversion from `Point' to
       non-scalar type `Segment' requested
```

As the second example, let us consider another version of class **Modulo** which has already been introduced in programs **modsev.cpp** and **modsev1.cpp**. We will see that a conversion constructor can be useful for operator overloading. It will be used here for overloading addition operator of numbers of type **Modulo**.

P173: modcon.cpp Using conversions for operator overloading

```
1. #include <iostream>
2. using namespace std;
3.
4. class Modulo {
5.     int numb;
6. public:
7.     static int modul;
8.
9.     Modulo() : numb(0) { }
10.
11.     Modulo(int numb) : numb(numb%modul) { }
12.
```

```
13.     friend Modulo operator+(Modulo,Modulo);
14.     friend ostream& operator<<(ostream&,const Modulo&);
15. };
16. int Modulo::modul = 7;
17.
18. Modulo operator+(Modulo m, Modulo n) {
19.     return Modulo(m.numb + n.numb);
20. }
21.
22. ostream& operator<<(ostream& str, const Modulo& m) {
23.     return str << m.numb;
24. }
25.
26. int main() {
27.
28.     Modulo m(5), n(6), k;
29.
30.     k = m + n;
31.     cout << "m + n (mod " << Modulo::modul
32.          << ") = "        << k << endl;
33.
34.     k = m + 6;
35.     cout << "m + 6 (mod " << Modulo::modul
36.          << ") = "        << k << endl;
37.
38.     k = 6 + m;
39.     cout << "6 + m (mod " << Modulo::modul
40.          << ") = "        << k << endl;
41. }
```

This time we overload addition of objects of type **Modulo** by a friend global function, not a method of the class. Note that we have defined only one such function (lines 18-20): with both parameters of type **Modulo**. But we can see in **main** that the function can be used for adding two objects of class **Modulo** (line 30), for adding an **int** to object of class **Modulo** (line 34) and also for adding an object of class **Modulo** to an **int** (line 38). In all cases the addition works as expected:

```
m + n (mod 7) = 4
m + 6 (mod 7) = 4
6 + m (mod 7) = 4
```

Note that the last form, *number+object*, would be impossible to implement as a method, because we have a number on the left-hand side of the operator. How can it be that one function, with both parameters of type **Modulo**, can handle all three cases? The class has a conversion constructor **int** → **Modulo** (line 11). It is therefore sufficient to have only one argument of type **Modulo** — the second, be it the left or the right one, can be an **int**: it will be converted to type **Modulo** automatically.

20.1.2 Conversions *from* user-defined type

Sometimes we would like to define a conversions in the opposite direction: from a user-defined type to another type. Suppose, for example, that we would like to have conversion **Modulo** → **int**. We cannot do it by defining a conversion constructor in class **int** for several reasons: we do not have access to the definition of the target type and, last but not least, the target type is not a class at all...Even if the target type is a class, it can be defined in a library that we do not want or are not able to modify.

In these situations, we can define a **conversion method** in a class *from* which a conversion is required. It should be a parameterless method named ' **operator Type**', where **Type** is the name of a target type — built-in or also a user-defined one.

When declaring/defining a conversion method, one *must not* specify any return type. This is logical, because a return type is already determined by the name of the method itself. Still, the method has to terminate with a **return** statement returning a value of the target type. As all other methods, a conversion method is always invoked on an object: its task is to return a value of the target type corresponding to *this* object. Of couse, the source object should not be modified, so usually we declare such methods as **const**. We can also declare them as **explicit** (as converting constructors) to avoid accidental, unwanted conversions, but then we have to use explicit conversions where they are needed.

Let us consider an example similar to that from program convto.cpp:

P174: convfrom.cpp Conversion from user-defined type

```
1.  #include <iostream>
2.  #include <cmath>
3.  using std::cout; using std::endl;
4.
5.  struct Point {
6.      double x, y;
7.      Point(double x = 0, double y = 0) : x(x), y(y) { }
8.      operator double() const {          ❶
9.          return std::sqrt(x*x+y*y);
10.     }
11. };
12.
13. struct Segment {
14.     Point A, B;
15.     Segment(Point A = Point(), Point B = Point())
16.         : A(A), B(B)
17.     { }
18.     operator Point() const {           ❷
19.         return Point( (A.x+B.x)/2, (A.y+B.y)/2 );
20.     }
21. };
22.
23. void showPoint(Point A) {
24.     cout << "Point[" << A.x << "," << A.y << "]";
25. }
26.
27. void showSegment(Segment AB) {
28.     cout << "Segment: ";
29.     showPoint(AB.A);
30.     cout << "--";
31.     showPoint(AB.B);
32.     cout << endl;
33. }
34.
35. void showDouble(double d) {
36.     cout << "Double " << d;
37. }
38.
39. int main() {
40.     Point A(3,4);
41.     showPoint(A);                      ❸
42.     cout << endl;
43.     showDouble(A);                     ❹
44.
45.     cout << endl;
46.
47.     Segment BC(Point(1,1),Point(3,3));
48.     showSegment(BC);
49.     showPoint(BC);                     ❺
50.
51.     cout << endl;
52. }
```

Class **Point** has been equipped with a conversion method **operator double()** (❶). It returns the distance of *this* point from the origin as a value of type **double** (i.e., of a built-in type). Similarly, we have added to class **Segment** a method converting segments to points (❷). It returns an object representing the geometrical center of *this* segment. Finally, we have added a function **showDouble**, which just prints the value passed to it through the argument.

In line ❸, function **showPoint** is called with argument of type **Point**, i.e., of the declared type of its parameter. Immediately after that we call function **showDouble** with the same argument of type **Point** (❹). The function expects a **double**, so a conversion **Point** → **double** is needed — such conversion is indeed defined by the conversion method in class **Point**. Therefore, the call is valid, what can be seen from the second line of the output:

```
Point[3,4]
Double 5
Segment: Point[1,1]--Point[3,3]
Point[2,2]
```

In line ❺, the situation is similar. This time we pass a segment to a function accepting points. Again, a conversion will take place, this time it will be a conversion **Segment** → **Point** (❷).

Note that in the second case we could have defined a conversion *constructor* in class **Point** taking a point as an argument: the effect would be equivalent to that of defining conversion *method* in class **Segment**. However, in the first case we had no choice: it is impossible to define a conversion constructor in class **double** (mainly because there is no such class...).

Conversion methods can be inherited and can be virtual — the meaning of this statement will become clear in one of the next chapters.

Prev: 20. More on conversions **Up:** 20. More on conversions **Next:** 20.2 Explicit conversions

Subsections

- 20.2.1 *const_cast* conversions
- 20.2.2 Static casts
- 20.2.3 Dynamic conversions
- 20.2.4 Forced conversions

20.2 Explicit conversions

In C (and in Java as well) one can explicitly convert a value of one type to a value of another type by **casting**. It has the form

```
(Type) expression
```

where **Type** is the name of a target type and ***expression*** is an r-value of another type. The result is an r-value of type **Type** representing, in some sense, the same value as that of ***expression***.

In Java such conversions are safe: the compiler (at compile time) or the virtual machine (at ryn time) will check if a cast is sensible. In C such casting can be performed “by force” leading to a nonsense result. It is then much better to use new forms of casting, introduced in C++, which give more control over the process of casting. All casting operators in C++ have form

```
kind_cast<Type>(expression)
```

where `kind' can be ***static***, ***dynamic***, ***const*** or ***reinterpret***. The result will be a value of type **Type** corresponding to the value of ***expression***.

20.2.1 *const_cast* conversions

This type of conversion, called **const_cast conversion** has form

```
const_cast<Type>(expression)
```

where ***expression*** must be of type **const Type** or **volatile Type**. The conversion (cast) removes the **const** or **volatile** attribute and can be used *only* for that purpose. On the other hand, other forms of casting (**static_cast**, **dynamic_cast** or **reinterpret_cast**) *cannot* be used in this context: the compiler would raise an error. An example:

P175: concast.cpp Removing *const* attribute

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. void changeFirst(char* str, char c) {
5.     str[0]=c;
6. }
7.
8. int main() {
9.     const char name[] = "Jenny";
10.     cout << name << endl;
11.
12.     // name[0]='K';
13.
14.     changeFirst(const_cast<char*>(name),'K');
15.
16.     // changeFirst(name,'K');
17.
18.     cout << name << endl;
19. }
```

Function **changeFirst** modifies the first character of the C-string passed as the argument. In the main program, we create a **const** C-string ***imie*** with content "Jenny" (line 9). An attempt to modify it, as in commented out line 12, would lead to a compilation error. On line 14 we pass the string to function **changeFirst**, but removing **const** attribute; as can be seen from the output

```
Jenny
Kenny
```

the string *was* modified. Note that it would be impossible without casting, e.g., as on line 16, which is illegal. The variable ***name*** is constant and the function **changeFirst** does not promise (by declaring the type of its parameter as **const**) that it will not modify the C-string passed to it (actually, the function cannot promise this, because it *does* modify the string).

When we declare variables as **const**, our intention is to protect them against even accidental modification. Use of **const_cast** usually means that there is a flaw in the design of our program. Such casts should therefore be used only in exceptional circumstances.

20.2.2 Static casts

Static cast has form

```
static_cast<Type>(expression)
```

and performs conversion of the value of ***expression*** to a value of type **Type**. The compiler "does its best" to verify if this is a legitimate conversion, but later, at run-time, no additional type check is made to ensure the safety of the conversion. This type of conversion is often used even if a conversion would take place anyway: explicit conversion prevents the compiler from issuing warnings about possible loss of information and makes intentions of the programmer more clear. For example,

```
double x = 4;
int i = x;
```

is legal but triggers some warnings during compilation

```
d.cpp:6: warning: initialization to `int' from `double'
d.cpp:6: warning: argument to `int' from `double'
```

which can be avoided by making the conversion explicit:

```
double x = 4;
int i = static_cast<int>(x);
```

Static conversion is also often used to convert from type **void*** to **Type*** — the safety of this kind of conversion cannot be checked during compilation and will not be checked at run-time, so the programmer has to know what he/she is doing...(conversion in the opposite direction is always safe, so it does not have to be checked or made explicit).

20.2.3 Dynamic conversions

Dynamic conversion (cast) has form

```
dynamic_cast<Type>(expression)
```

Dynamic conversions are used when the legibility of the conversion cannot be checked at compile-time, because it depends on dynamic type of an object of a polymorphic type. Type **Type** must be a pointer or reference type: in the former case ***expression*** has to be of pointer type as well, in the latter ***expression*** must be an l-value. We will say more about dynamic casts when we have learnt about polymorphism (section on dynamic casts).

20.2.4 Forced conversions

The **forced cast** has form

```
reinterpret_cast<Type>(expression)
```

It basically means that we do not want to check the legitimity of the conversion at all: neither at compile-time nor at run-time. The programmer takes full responsibility for the conversion to have any sense. The conversion can be used to cast, e.g., **char*** → **int*** or **classA*** → **classB*** with classes **classA** and **classB** completely unrelated: these conversions are inherently unsafe and their result can depend on the platform or compiler in use.

An example:

P176: dyncast.cpp Explicit forced casts

```
1. #include <iostream>
2. #include <cstring>
3. #include <fstream>
4. using namespace std;
5.
6. class Person {
7.     char nam[30];
8.     int  age;
9. public:
10.     Person(const char* n, int a) : age(a) {
11.         strcpy(nam,n);
12.     }
13.
14.     void info() {
15.         cout << nam << " (" << age << ")" << endl;
16.     }
17. };
18.
19. int main() {
20.     const size_t size = sizeof(Person);
21.
```

```
22.     Person john("John Brown",40);
23.     Person mary("Mary Wiles",26);
24.
25.     ofstream out("person.ob");
26.     out.write(reinterpret_cast<char*>(&john),size);
27.     out.write(                (char*) &mary ,size);
28.     out.close();
29.
30.     char* buff1 = new char[size];
31.     char* buff2 = new char[size];
32.     ifstream in("person.ob");
33.     in.read(buff1,size);
34.     in.read(buff2,size);
35.     in.close();
36.
37.     Person* p1 = reinterpret_cast<Person*>(buff1);
38.     Person* p2 =                (Person*) buff2 ;
39.
40.     p1->info();
41.     p2->info();
42.
43.     delete [] buff1;
44.     delete [] buff2;
45. }
```

On lines 22 and 23 we have defined two objects of type **Person**; each contains as members an array and an **int**. Then, on lines 26 and 27, we dump the objects on disk in binary format. Method **write** (see section on unformatted writing) has the first parameter of type **const char***, while ***&john*** is of type **Person***: therefore, we cast ***&john*** to type **char*** using **reinterpret_cast** and, in the next line, C-like casting (just to show that these two forms are equivalent here).

Two object of class **Person** are then read back to character buffers ***buff1*** and ***buff2*** (lines 33 and 34). After that they are just arrays of characters (bytes) — neither ar compile-time nor at run-time the system is able to check whether these sequences of bytes do actually correspond to any objects. However, we do know that this is the case. We can therefore force a conversion (lines 37 and 38) of buffers ***buff*** to type **Person*** — again **reinterpret_cast** and C-like casting are equivalent in this context. The output from lines 40 and 41

```
John Brown (40)
Mary Wiles (26)
```

shows that indeed the arrays can be treated as the byte representation of our original objects. One has to remember that this "trick" would not work with more coplicated objects (of, e.g., a polymorphic type). The result can depend on the platform and compiler used and we are entirely responsible for consequences of such conversions.

Prev: 20.1 Conversions to and from **Up:** 20. More on conversions **Next:** 21. Inheritance and polymorphism

Subsections

- 20.2.1 *const_cast* conversions
- 20.2.2 Static casts
- 20.2.3 Dynamic conversions
- 20.2.4 Forced conversions

20.2 Explicit conversions

In C (and in Java as well) one can explicitly convert a value of one type to a value of another type by **casting**. It has the form

```
(Type) expression
```

where **Type** is the name of a target type and ***expression*** is an r-value of another type. The result is an r-value of type **Type** representing, in some sense, the same value as that of ***expression***.

In Java such conversions are safe: the compiler (at compile time) or the virtual machine (at ryn time) will check if a cast is sensible. In C such casting can be performed “by force” leading to a nonsense result. It is then much better to use new forms of casting, introduced in C++, which give more control over the process of casting. All casting operators in C++ have form

```
kind_cast<Type>(expression)
```

where `kind' can be ***static***, ***dynamic***, ***const*** or ***reinterpret***. The result will be a value of type **Type** corresponding to the value of ***expression***.

20.2.1 *const_cast* conversions

This type of conversion, called **const_cast conversion** has form

```
const_cast<Type>(expression)
```

where ***expression*** must be of type **const Type** or **volatile Type**. The conversion (cast) removes the **const** or **volatile** attribute and can be used *only* for that purpose. On the other hand, other forms of casting (**static_cast**, **dynamic_cast** or **reinterpret_cast**) *cannot* be used in this context: the compiler would raise an error. An example:

P175: concast.cpp Removing *const* attribute

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. void changeFirst(char* str, char c) {
5.     str[0]=c;
6. }
7.
8. int main() {
9.     const char name[] = "Jenny";
10.     cout << name << endl;
11.
12.     // name[0]='K';
13.
14.     changeFirst(const_cast<char*>(name),'K');
15.
16.     // changeFirst(name,'K');
17.
18.     cout << name << endl;
19. }
```

Function **changeFirst** modifies the first character of the C-string passed as the argument. In the main program, we create a **const** C-string ***imie*** with content "Jenny" (line 9). An attempt to modify it, as in commented out line 12, would lead to a compilation error. On line 14 we pass the string to function **changeFirst**, but removing **const** attribute; as can be seen from the output

```
Jenny
Kenny
```

the string *was* modified. Note that it would be impossible without casting, e.g., as on line 16, which is illegal. The variable ***name*** is constant and the function **changeFirst** does not promise (by declaring the type of its parameter as **const**) that it will not modify the C-string passed to it (actually, the function cannot promise this, because it *does* modify the string).

When we declare variables as **const**, our intention is to protect them against even accidental modification. Use of **const_cast** usually means that there is a flaw in the design of our program. Such casts should therefore be used only in exceptional circumstances.

20.2.2 Static casts

Static cast has form

```
static_cast<Type>(expression)
```

and performs conversion of the value of ***expression*** to a value of type **Type**. The compiler "does its best" to verify if this is a legitimate conversion, but later, at run-time, no additional type check is made to ensure the safety of the conversion. This type of conversion is often used even if a conversion would take place anyway: explicit conversion prevents the compiler from issuing warnings about possible loss of information and makes intentions of the programmer more clear. For example,

```
double x = 4;
int i = x;
```

is legal but triggers some warnings during compilation

```
d.cpp:6: warning: initialization to `int' from `double'
d.cpp:6: warning: argument to `int' from `double'
```

which can be avoided by making the conversion explicit:

```
double x = 4;
int i = static_cast<int>(x);
```

Static conversion is also often used to convert from type **void*** to **Type*** — the safety of this kind of conversion cannot be checked during compilation and will not be checked at run-time, so the programmer has to know what he/she is doing...(conversion in the opposite direction is always safe, so it does not have to be checked or made explicit).

20.2.3 Dynamic conversions

Dynamic conversion (cast) has form

```
dynamic_cast<Type>(expression)
```

Dynamic conversions are used when the legibility of the conversion cannot be checked at compile-time, because it depends on dynamic type of an object of a polymorphic type. Type **Type** must be a pointer or reference type: in the former case ***expression*** has to be of pointer type as well, in the latter ***expression*** must be an l-value. We will say more about dynamic casts when we have learnt about polymorphism (section on dynamic casts).

20.2.4 Forced conversions

The **forced cast** has form

```
reinterpret_cast<Type>(expression)
```

It basically means that we do not want to check the legitimity of the conversion at all: neither at compile-time nor at run-time. The programmer takes full responsibility for the conversion to have any sense. The conversion can be used to cast, e.g., **char*** → **int*** or **classA*** → **classB*** with classes **classA** and **classB** completely unrelated: these conversions are inherently unsafe and their result can depend on the platform or compiler in use.

An example:

P176: dyncast.cpp Explicit forced casts

```
1. #include <iostream>
2. #include <cstring>
3. #include <fstream>
4. using namespace std;
5.
6. class Person {
7.     char nam[30];
8.     int  age;
9. public:
10.     Person(const char* n, int a) : age(a) {
11.         strcpy(nam,n);
12.     }
13.
14.     void info() {
15.         cout << nam << " (" << age << ")" << endl;
16.     }
17. };
18.
19. int main() {
20.     const size_t size = sizeof(Person);
21.
```

```
22.     Person john("John Brown",40);
23.     Person mary("Mary Wiles",26);
24.
25.     ofstream out("person.ob");
26.     out.write(reinterpret_cast<char*>(&john),size);
27.     out.write(                (char*) &mary ,size);
28.     out.close();
29.
30.     char* buff1 = new char[size];
31.     char* buff2 = new char[size];
32.     ifstream in("person.ob");
33.     in.read(buff1,size);
34.     in.read(buff2,size);
35.     in.close();
36.
37.     Person* p1 = reinterpret_cast<Person*>(buff1);
38.     Person* p2 =                (Person*) buff2 ;
39.
40.     p1->info();
41.     p2->info();
42.
43.     delete [] buff1;
44.     delete [] buff2;
45. }
```

On lines 22 and 23 we have defined two objects of type **Person**; each contains as members an array and an **int**. Then, on lines 26 and 27, we dump the objects on disk in binary format. Method **write** (see section on unformatted writing) has the first parameter of type **const char***, while ***&john*** is of type **Person***: therefore, we cast ***&john*** to type **char*** using **reinterpret_cast** and, in the next line, C-like casting (just to show that these two forms are equivalent here).

Two object of class **Person** are then read back to character buffers ***buff1*** and ***buff2*** (lines 33 and 34). After that they are just arrays of characters (bytes) — neither ar compile-time nor at run-time the system is able to check whether these sequences of bytes do actually correspond to any objects. However, we do know that this is the case. We can therefore force a conversion (lines 37 and 38) of buffers ***buff*** to type **Person*** — again **reinterpret_cast** and C-like casting are equivalent in this context. The output from lines 40 and 41

```
John Brown (40)
Mary Wiles (26)
```

shows that indeed the arrays can be treated as the byte representation of our original objects. One has to remember that this "trick" would not work with more coplicated objects (of, e.g., a polymorphic type). The result can depend on the platform and compiler used and we are entirely responsible for consequences of such conversions.

Prev: 20.1 Conversions to and from **Up:** 20. More on conversions **Next:** 21. Inheritance and polymorphism

Subsections

- 20.2.1 *const_cast* conversions
- 20.2.2 Static casts
- 20.2.3 Dynamic conversions
- 20.2.4 Forced conversions

20.2 Explicit conversions

In C (and in Java as well) one can explicitly convert a value of one type to a value of another type by **casting**. It has the form

```
(Type) expression
```

where **Type** is the name of a target type and ***expression*** is an r-value of another type. The result is an r-value of type **Type** representing, in some sense, the same value as that of ***expression***.

In Java such conversions are safe: the compiler (at compile time) or the virtual machine (at ryn time) will check if a cast is sensible. In C such casting can be performed “by force” leading to a nonsense result. It is then much better to use new forms of casting, introduced in C++, which give more control over the process of casting. All casting operators in C++ have form

```
kind_cast<Type>(expression)
```

where `kind' can be ***static***, ***dynamic***, ***const*** or ***reinterpret***. The result will be a value of type **Type** corresponding to the value of ***expression***.

20.2.1 *const_cast* conversions

This type of conversion, called **const_cast conversion** has form

```
const_cast<Type>(expression)
```

where ***expression*** must be of type **const Type** or **volatile Type**. The conversion (cast) removes the **const** or **volatile** attribute and can be used *only* for that purpose. On the other hand, other forms of casting (**static_cast**, **dynamic_cast** or **reinterpret_cast**) *cannot* be used in this context: the compiler would raise an error. An example:

P175: concast.cpp Removing *const* attribute

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. void changeFirst(char* str, char c) {
5.     str[0]=c;
6. }
7.
8. int main() {
9.     const char name[] = "Jenny";
10.     cout << name << endl;
11.
12.     // name[0]='K';
13.
14.     changeFirst(const_cast<char*>(name),'K');
15.
16.     // changeFirst(name,'K');
17.
18.     cout << name << endl;
19. }
```

Function **changeFirst** modifies the first character of the C-string passed as the argument. In the main program, we create a **const** C-string ***imie*** with content "Jenny" (line 9). An attempt to modify it, as in commented out line 12, would lead to a compilation error. On line 14 we pass the string to function **changeFirst**, but removing **const** attribute; as can be seen from the output

```
Jenny
Kenny
```

the string *was* modified. Note that it would be impossible without casting, e.g., as on line 16, which is illegal. The variable ***name*** is constant and the function **changeFirst** does not promise (by declaring the type of its parameter as **const**) that it will not modify the C-string passed to it (actually, the function cannot promise this, because it *does* modify the string).

When we declare variables as **const**, our intention is to protect them against even accidental modification. Use of **const_cast** usually means that there is a flaw in the design of our program. Such casts should therefore be used only in exceptional circumstances.

20.2.2 Static casts

Static cast has form

```
static_cast<Type>(expression)
```

and performs conversion of the value of ***expression*** to a value of type **Type**. The compiler "does its best" to verify if this is a legitimate conversion, but later, at run-time, no additional type check is made to ensure the safety of the conversion. This type of conversion is often used even if a conversion would take place anyway: explicit conversion prevents the compiler from issuing warnings about possible loss of information and makes intentions of the programmer more clear. For example,

```
double x = 4;
int i = x;
```

is legal but triggers some warnings during compilation

```
d.cpp:6: warning: initialization to `int' from `double'
d.cpp:6: warning: argument to `int' from `double'
```

which can be avoided by making the conversion explicit:

```
double x = 4;
int i = static_cast<int>(x);
```

Static conversion is also often used to convert from type **void*** to **Type*** — the safety of this kind of conversion cannot be checked during compilation and will not be checked at run-time, so the programmer has to know what he/she is doing...(conversion in the opposite direction is always safe, so it does not have to be checked or made explicit).

20.2.3 Dynamic conversions

Dynamic conversion (cast) has form

```
dynamic_cast<Type>(expression)
```

Dynamic conversions are used when the legibility of the conversion cannot be checked at compile-time, because it depends on dynamic type of an object of a polymorphic type. Type **Type** must be a pointer or reference type: in the former case ***expression*** has to be of pointer type as well, in the latter ***expression*** must be an l-value. We will say more about dynamic casts when we have learnt about polymorphism (section on dynamic casts).

20.2.4 Forced conversions

The **forced cast** has form

```
reinterpret_cast<Type>(expression)
```

It basically means that we do not want to check the legitimity of the conversion at all: neither at compile-time nor at run-time. The programmer takes full responsibility for the conversion to have any sense. The conversion can be used to cast, e.g., **char*** → **int*** or **classA*** → **classB*** with classes **classA** and **classB** completely unrelated: these conversions are inherently unsafe and their result can depend on the platform or compiler in use.

An example:

P176: dyncast.cpp Explicit forced casts

```
1. #include <iostream>
2. #include <cstring>
3. #include <fstream>
4. using namespace std;
5.
6. class Person {
7.     char nam[30];
8.     int  age;
9. public:
10.     Person(const char* n, int a) : age(a) {
11.         strcpy(nam,n);
12.     }
13.
14.     void info() {
15.         cout << nam << " (" << age << ")" << endl;
16.     }
17. };
18.
19. int main() {
20.     const size_t size = sizeof(Person);
21.
```

```
22.     Person john("John Brown",40);
23.     Person mary("Mary Wiles",26);
24.
25.     ofstream out("person.ob");
26.     out.write(reinterpret_cast<char*>(&john),size);
27.     out.write(                (char*) &mary ,size);
28.     out.close();
29.
30.     char* buff1 = new char[size];
31.     char* buff2 = new char[size];
32.     ifstream in("person.ob");
33.     in.read(buff1,size);
34.     in.read(buff2,size);
35.     in.close();
36.
37.     Person* p1 = reinterpret_cast<Person*>(buff1);
38.     Person* p2 =                (Person*) buff2 ;
39.
40.     p1->info();
41.     p2->info();
42.
43.     delete [] buff1;
44.     delete [] buff2;
45. }
```

On lines 22 and 23 we have defined two objects of type **Person**; each contains as members an array and an **int**. Then, on lines 26 and 27, we dump the objects on disk in binary format. Method **write** (see section on unformatted writing) has the first parameter of type **const char***, while ***&john*** is of type **Person***: therefore, we cast ***&john*** to type **char*** using **reinterpret_cast** and, in the next line, C-like casting (just to show that these two forms are equivalent here).

Two object of class **Person** are then read back to character buffers ***buff1*** and ***buff2*** (lines 33 and 34). After that they are just arrays of characters (bytes) — neither ar compile-time nor at run-time the system is able to check whether these sequences of bytes do actually correspond to any objects. However, we do know that this is the case. We can therefore force a conversion (lines 37 and 38) of buffers ***buff*** to type **Person*** — again **reinterpret_cast** and C-like casting are equivalent in this context. The output from lines 40 and 41

```
John Brown (40)
Mary Wiles (26)
```

shows that indeed the arrays can be treated as the byte representation of our original objects. One has to remember that this "trick" would not work with more coplicated objects (of, e.g., a polymorphic type). The result can depend on the platform and compiler used and we are entirely responsible for consequences of such conversions.

Prev: 20.1 Conversions to and from **Up:** 20. More on conversions **Next:** 21. Inheritance and polymorphism

Subsections

- 20.2.1 *const_cast* conversions
- 20.2.2 Static casts
- 20.2.3 Dynamic conversions
- 20.2.4 Forced conversions

20.2 Explicit conversions

In C (and in Java as well) one can explicitly convert a value of one type to a value of another type by **casting**. It has the form

```
(Type) expression
```

where **Type** is the name of a target type and ***expression*** is an r-value of another type. The result is an r-value of type **Type** representing, in some sense, the same value as that of ***expression***.

In Java such conversions are safe: the compiler (at compile time) or the virtual machine (at ryn time) will check if a cast is sensible. In C such casting can be performed “by force” leading to a nonsense result. It is then much better to use new forms of casting, introduced in C++, which give more control over the process of casting. All casting operators in C++ have form

```
kind_cast<Type>(expression)
```

where `kind' can be ***static***, ***dynamic***, ***const*** or ***reinterpret***. The result will be a value of type **Type** corresponding to the value of ***expression***.

20.2.1 *const_cast* conversions

This type of conversion, called **const_cast conversion** has form

```
const_cast<Type>(expression)
```

where ***expression*** must be of type **const Type** or **volatile Type**. The conversion (cast) removes the **const** or **volatile** attribute and can be used *only* for that purpose. On the other hand, other forms of casting (**static_cast**, **dynamic_cast** or **reinterpret_cast**) *cannot* be used in this context: the compiler would raise an error. An example:

P175: concast.cpp Removing *const* attribute

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. void changeFirst(char* str, char c) {
5.     str[0]=c;
6. }
7.
8. int main() {
9.     const char name[] = "Jenny";
10.     cout << name << endl;
11.
12.     // name[0]='K';
13.
14.     changeFirst(const_cast<char*>(name),'K');
15.
16.     // changeFirst(name,'K');
17.
18.     cout << name << endl;
19. }
```

Function **changeFirst** modifies the first character of the C-string passed as the argument. In the main program, we create a **const** C-string ***imie*** with content "Jenny" (line 9). An attempt to modify it, as in commented out line 12, would lead to a compilation error. On line 14 we pass the string to function **changeFirst**, but removing **const** attribute; as can be seen from the output

```
Jenny
Kenny
```

the string *was* modified. Note that it would be impossible without casting, e.g., as on line 16, which is illegal. The variable ***name*** is constant and the function **changeFirst** does not promise (by declaring the type of its parameter as **const**) that it will not modify the C-string passed to it (actually, the function cannot promise this, because it *does* modify the string).

When we declare variables as **const**, our intention is to protect them against even accidental modification. Use of **const_cast** usually means that there is a flaw in the design of our program. Such casts should therefore be used only in exceptional circumstances.

20.2.2 Static casts

Static cast has form

```
static_cast<Type>(expression)
```

and performs conversion of the value of ***expression*** to a value of type **Type**. The compiler "does its best" to verify if this is a legitimate conversion, but later, at run-time, no additional type check is made to ensure the safety of the conversion. This type of conversion is often used even if a conversion would take place anyway: explicit conversion prevents the compiler from issuing warnings about possible loss of information and makes intentions of the programmer more clear. For example,

```
double x = 4;
int i = x;
```

is legal but triggers some warnings during compilation

```
d.cpp:6: warning: initialization to `int' from `double'
d.cpp:6: warning: argument to `int' from `double'
```

which can be avoided by making the conversion explicit:

```
double x = 4;
int i = static_cast<int>(x);
```

Static conversion is also often used to convert from type **void*** to **Type*** — the safety of this kind of conversion cannot be checked during compilation and will not be checked at run-time, so the programmer has to know what he/she is doing...(conversion in the opposite direction is always safe, so it does not have to be checked or made explicit).

20.2.3 Dynamic conversions

Dynamic conversion (cast) has form

```
dynamic_cast<Type>(expression)
```

Dynamic conversions are used when the legibility of the conversion cannot be checked at compile-time, because it depends on dynamic type of an object of a polymorphic type. Type **Type** must be a pointer or reference type: in the former case ***expression*** has to be of pointer type as well, in the latter ***expression*** must be an l-value. We will say more about dynamic casts when we have learnt about polymorphism (section on dynamic casts).

20.2.4 Forced conversions

The **forced cast** has form

```
reinterpret_cast<Type>(expression)
```

It basically means that we do not want to check the legitimity of the conversion at all: neither at compile-time nor at run-time. The programmer takes full responsibility for the conversion to have any sense. The conversion can be used to cast, e.g., **char*** → **int*** or **classA*** → **classB*** with classes **classA** and **classB** completely unrelated: these conversions are inherently unsafe and their result can depend on the platform or compiler in use.

An example:

P176: dyncast.cpp Explicit forced casts

```
1. #include <iostream>
2. #include <cstring>
3. #include <fstream>
4. using namespace std;
5.
6. class Person {
7.     char nam[30];
8.     int  age;
9. public:
10.     Person(const char* n, int a) : age(a) {
11.         strcpy(nam,n);
12.     }
13.
14.     void info() {
15.         cout << nam << " (" << age << ")" << endl;
16.     }
17. };
18.
19. int main() {
20.     const size_t size = sizeof(Person);
21.
```

```
22.     Person john("John Brown",40);
23.     Person mary("Mary Wiles",26);
24.
25.     ofstream out("person.ob");
26.     out.write(reinterpret_cast<char*>(&john),size);
27.     out.write(                (char*) &mary ,size);
28.     out.close();
29.
30.     char* buff1 = new char[size];
31.     char* buff2 = new char[size];
32.     ifstream in("person.ob");
33.     in.read(buff1,size);
34.     in.read(buff2,size);
35.     in.close();
36.
37.     Person* p1 = reinterpret_cast<Person*>(buff1);
38.     Person* p2 =                (Person*) buff2 ;
39.
40.     p1->info();
41.     p2->info();
42.
43.     delete [] buff1;
44.     delete [] buff2;
45. }
```

On lines 22 and 23 we have defined two objects of type **Person**; each contains as members an array and an **int**. Then, on lines 26 and 27, we dump the objects on disk in binary format. Method **write** (see section on unformatted writing) has the first parameter of type **const char***, while ***&john*** is of type **Person***: therefore, we cast ***&john*** to type **char*** using **reinterpret_cast** and, in the next line, C-like casting (just to show that these two forms are equivalent here).

Two object of class **Person** are then read back to character buffers ***buff1*** and ***buff2*** (lines 33 and 34). After that they are just arrays of characters (bytes) — neither ar compile-time nor at run-time the system is able to check whether these sequences of bytes do actually correspond to any objects. However, we do know that this is the case. We can therefore force a conversion (lines 37 and 38) of buffers ***buff*** to type **Person*** — again **reinterpret_cast** and C-like casting are equivalent in this context. The output from lines 40 and 41

```
John Brown (40)
Mary Wiles (26)
```

shows that indeed the arrays can be treated as the byte representation of our original objects. One has to remember that this "trick" would not work with more coplicated objects (of, e.g., a polymorphic type). The result can depend on the platform and compiler used and we are entirely responsible for consequences of such conversions.

Prev: 20.1 Conversions to and from **Up:** 20. More on conversions **Next:** 21. Inheritance and polymorphism

Subsections

- 20.2.1 *const_cast* conversions
- 20.2.2 Static casts
- 20.2.3 Dynamic conversions
- 20.2.4 Forced conversions

20.2 Explicit conversions

In C (and in Java as well) one can explicitly convert a value of one type to a value of another type by **casting**. It has the form

```
(Type) expression
```

where **Type** is the name of a target type and ***expression*** is an r-value of another type. The result is an r-value of type **Type** representing, in some sense, the same value as that of ***expression***.

In Java such conversions are safe: the compiler (at compile time) or the virtual machine (at ryn time) will check if a cast is sensible. In C such casting can be performed “by force” leading to a nonsense result. It is then much better to use new forms of casting, introduced in C++, which give more control over the process of casting. All casting operators in C++ have form

```
kind_cast<Type>(expression)
```

where `kind' can be ***static***, ***dynamic***, ***const*** or ***reinterpret***. The result will be a value of type **Type** corresponding to the value of ***expression***.

20.2.1 *const_cast* conversions

This type of conversion, called **const_cast conversion** has form

```
const_cast<Type>(expression)
```

where ***expression*** must be of type **const Type** or **volatile Type**. The conversion (cast) removes the **const** or **volatile** attribute and can be used *only* for that purpose. On the other hand, other forms of casting (**static_cast**, **dynamic_cast** or **reinterpret_cast**) *cannot* be used in this context: the compiler would raise an error. An example:

P175: concast.cpp Removing *const* attribute

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4. void changeFirst(char* str, char c) {
5.     str[0]=c;
6. }
7.
8. int main() {
9.     const char name[] = "Jenny";
10.     cout << name << endl;
11.
12.     // name[0]='K';
13.
14.     changeFirst(const_cast<char*>(name),'K');
15.
16.     // changeFirst(name,'K');
17.
18.     cout << name << endl;
19. }
```

Function **changeFirst** modifies the first character of the C-string passed as the argument. In the main program, we create a **const** C-string ***imie*** with content "Jenny" (line 9). An attempt to modify it, as in commented out line 12, would lead to a compilation error. On line 14 we pass the string to function **changeFirst**, but removing **const** attribute; as can be seen from the output

```
Jenny
Kenny
```

the string *was* modified. Note that it would be impossible without casting, e.g., as on line 16, which is illegal. The variable ***name*** is constant and the function **changeFirst** does not promise (by declaring the type of its parameter as **const**) that it will not modify the C-string passed to it (actually, the function cannot promise this, because it *does* modify the string).

When we declare variables as **const**, our intention is to protect them against even accidental modification. Use of **const_cast** usually means that there is a flaw in the design of our program. Such casts should therefore be used only in exceptional circumstances.

20.2.2 Static casts

Static cast has form

```
static_cast<Type>(expression)
```

and performs conversion of the value of ***expression*** to a value of type **Type**. The compiler "does its best" to verify if this is a legitimate conversion, but later, at run-time, no additional type check is made to ensure the safety of the conversion. This type of conversion is often used even if a conversion would take place anyway: explicit conversion prevents the compiler from issuing warnings about possible loss of information and makes intentions of the programmer more clear. For example,

```
double x = 4;
int i = x;
```

is legal but triggers some warnings during compilation

```
d.cpp:6: warning: initialization to `int' from `double'
d.cpp:6: warning: argument to `int' from `double'
```

which can be avoided by making the conversion explicit:

```
double x = 4;
int i = static_cast<int>(x);
```

Static conversion is also often used to convert from type **void*** to **Type*** — the safety of this kind of conversion cannot be checked during compilation and will not be checked at run-time, so the programmer has to know what he/she is doing...(conversion in the opposite direction is always safe, so it does not have to be checked or made explicit).

20.2.3 Dynamic conversions

Dynamic conversion (cast) has form

```
dynamic_cast<Type>(expression)
```

Dynamic conversions are used when the legibility of the conversion cannot be checked at compile-time, because it depends on dynamic type of an object of a polymorphic type. Type **Type** must be a pointer or reference type: in the former case ***expression*** has to be of pointer type as well, in the latter ***expression*** must be an l-value. We will say more about dynamic casts when we have learnt about polymorphism (section on dynamic casts).

20.2.4 Forced conversions

The **forced cast** has form

```
reinterpret_cast<Type>(expression)
```

It basically means that we do not want to check the legitimity of the conversion at all: neither at compile-time nor at run-time. The programmer takes full responsibility for the conversion to have any sense. The conversion can be used to cast, e.g., **char*** → **int*** or **classA*** → **classB*** with classes **classA** and **classB** completely unrelated: these conversions are inherently unsafe and their result can depend on the platform or compiler in use.

An example:

P176: dyncast.cpp Explicit forced casts

```
1. #include <iostream>
2. #include <cstring>
3. #include <fstream>
4. using namespace std;
5.
6. class Person {
7.     char nam[30];
8.     int  age;
9. public:
10.     Person(const char* n, int a) : age(a) {
11.         strcpy(nam,n);
12.     }
13.
14.     void info() {
15.         cout << nam << " (" << age << ")" << endl;
16.     }
17. };
18.
19. int main() {
20.     const size_t size = sizeof(Person);
21.
```

```
22.     Person john("John Brown",40);
23.     Person mary("Mary Wiles",26);
24.
25.     ofstream out("person.ob");
26.     out.write(reinterpret_cast<char*>(&john),size);
27.     out.write(                (char*) &mary ,size);
28.     out.close();
29.
30.     char* buff1 = new char[size];
31.     char* buff2 = new char[size];
32.     ifstream in("person.ob");
33.     in.read(buff1,size);
34.     in.read(buff2,size);
35.     in.close();
36.
37.     Person* p1 = reinterpret_cast<Person*>(buff1);
38.     Person* p2 =                (Person*) buff2 ;
39.
40.     p1->info();
41.     p2->info();
42.
43.     delete [] buff1;
44.     delete [] buff2;
45. }
```

On lines 22 and 23 we have defined two objects of type **Person**; each contains as members an array and an **int**. Then, on lines 26 and 27, we dump the objects on disk in binary format. Method **write** (see section on unformatted writing) has the first parameter of type **const char***, while ***&john*** is of type **Person***: therefore, we cast ***&john*** to type **char*** using **reinterpret_cast** and, in the next line, C-like casting (just to show that these two forms are equivalent here).

Two object of class **Person** are then read back to character buffers ***buff1*** and ***buff2*** (lines 33 and 34). After that they are just arrays of characters (bytes) — neither ar compile-time nor at run-time the system is able to check whether these sequences of bytes do actually correspond to any objects. However, we do know that this is the case. We can therefore force a conversion (lines 37 and 38) of buffers ***buff*** to type **Person*** — again **reinterpret_cast** and C-like casting are equivalent in this context. The output from lines 40 and 41

```
John Brown (40)
Mary Wiles (26)
```

shows that indeed the arrays can be treated as the byte representation of our original objects. One has to remember that this "trick" would not work with more coplicated objects (of, e.g., a polymorphic type). The result can depend on the platform and compiler used and we are entirely responsible for consequences of such conversions.

Prev: 20.1 Conversions to and from **Up:** 20. More on conversions **Next:** 21. Inheritance and polymorphism

21. Inheritance and polymorphism

Inheritance is the essence of object programming and C++, being an object-oriented language, supports it, as do other languages, like Modula, Java or Smalltalk and many other. Classes can **inherit** their properties from other classes: they are then called **derived** classes (**subclasses**) while classes they inherit from are called their **base** classes. A class that does not inherit from any other class is a **primary** class — such classes exist in C++, but not necessarily in other languages (Java, C#). A class can inherit from many classes (**multibase inheritance**); this, however, often leads to a complications which are better avoided, if possible (for that reason, there is no multibase inheritance in Java or C#).

Subsections

- 21.1 Fundamentals of inheritance
- 21.2 Constructors and destructors of derived classes
- 21.3 Assignment operator in derived classes
- 21.4 Virtual methods and polymorphism
- 21.5 Abstract classes
- 21.6 Virtual destructors
- 21.7 Multiple inheritance

T.R. Werner, September 27, 2023; 22:32

21.1 Fundamentals of inheritance

Defining a **derived class** (or **subclass**), we define a new data type which extends the type determined by the **base class** (or **superclass**), i.e., the class from which our new class **inherits**. Objects of the subclass will contain, as their **subobject**, an object of the superclass (with all its members) and, most often but not necessarily, other, additional members specific for the subclass. A derived class can also define new methods or redefine methods inherited from its base class(es).

Thus one can say that a base class is more general — it models a fragment of the real world on a more abstract level. A subclass is more specific, i.e., less abstract: for example, we could think about a class describing any piece of furniture as a base class and a class describing chairs as its subclass; it would define additional members (like *seat*), which were absent in the superclass (because not all pieces of furniture have seats) and also additional actions that can be performed on them. Every chair *is* a piece of furniture, buy not *vice versa*. One denotes this kind of relation between classes by an arrow from the *derived* class to its *base* class (**Furniture** ← **Chair**).

When defining subclasses, one uses the following syntax:

```
class A {
    // ...
};

class B : public A {
    // ...
};

class C : B {
    // ...
};
```

In the definition of a subclass, one puts a comma-separated **inheritance list** of names of its base classes — after the colon following the name of the class being declared/defined but before the opening brace which begins the body of the declaration/definition. In the example above

- class **A** is a primary class; it does not inherit from any other class (there is no common ancestor of the whole class hierarchy, like class **Object** in Java);
- class **B** inherits from class **A**, i.e., **B** is a class derived form **A**; class **B** is a subclass here and **A** is its base class (superclass);
- class **C** inherits directly from **B** and, consequently, indirectly also from **A**.

In the definition of **B**, the base class **A** is declared as **public**. Other possibilities here would be **private** or **protected**. This keyword determines the upper limit of accessibility for the inherited members (see section on accessibility of class members). Members inherited from the base class will have the same accessibility as they had in the base class, but not higher then declared on the inheritance list. Private members of the base class are not directly visible in derived class, although they can be accessed by inherited non-private

member functions.

Private fields, eventhough not visible, are still physically contained in objects of derived classes.

Summarizing:

- The **public** specifier means that all public members from the base class are **public** in derived class and **protected** remain **protected**; **private** members are not visible at all;
- The **protected** specifier means that **public** members of the base class become **protected** in derived class and **protected** remain **protected**;
- The **private** specifier means that all **public** and **protected** members of the base class become **private** in derived classes.

Recall that

protected members are accessible (as if they were public) in derived classes, but inaccessible (as if they were private) in all other scopes.

If there is no accessibility specifier at all, it is assumed that the inheritance is **private**. Note that deriving classes from class **C** would not have much sense: in objects of such classes no members of **A** and **B** would be accessible.

If we narrowed accessibility of members by using **protected** or **private** (after a colon but before definition of a class), we can *restore* original accessibility of some members individually (restore, but not widen). We can do it by specifying their qualified names in appropriate sections of a class.

In the example below, fields ***x***, ***y*** and ***z*** of class **A** are public, while methods **fff**, **ggg** and **hhh** are protected. Field ***k*** is private and will ot be visible at all in derived classes.

```
class A {
    double k;

public:
    int x, y, z

protected:
    double fff(int);
    double ggg(int);
    double hhh(int);
    // ...
};
```

Now, class **B** is defined as follows:

```
1.    class B : private A {
2.    public:
3.        A::x;
4.        A::y;
5.
6.    protected:
7.        A::fff;
8.        // ...
9.    };
```

In class **B**, accessibility of inherited members has been narrowed to **private** (line 1; we

could have got the same effect by not specifying any accessibility level, as the private accessibility is the default one for classes). However, public accessibility is then restored for members ***x*** and ***y*** (lines 2-4). Note that we have not restored the original accessibility of ***z*** — in class **B** it will become **private**. Note also that on lines 3-4 we specified only qualified *names*, not full declarations with type specification.

Similarly, on lines 6-7 we restored **protected** accessibility of **fff** (just by specifying its qualified name in **protected** section of class definition). The methods **ggg** and **hhh** will be **private** in class **B**.

In addition to members inherited from the base class(es), a derived class can define its own members, not present in any of its base classes. That means that objects of a derived class cannot be smaller that objects of base classes — they always contain all members of base classes (even private ones, although these are not visible) and possibly their own, additional members.

Derived class can redefine members with the same name as members inherited from base classes. We then say about **overriding** inherited members. It is best *not* to override fields — this, alhough possible, usually leads to uncontrolled chaos. However, overriding *methods* is very common and is actually the essence of object programming.

Methods and fields of a base class (say, **A**) reside in this class' namespace and are accessible for other members of the same class. The scope of a derived class (say, **B**) is contained in this namespace: its methods have access to non-private members of the base class, provided they were not overriden ("shadowed") by names declared in the derived class. Even if this is the case, non-private base class members can still be accessed, but require qualification with this class' name ('`A::`').

Let us consider an example:

P177: inher.cpp Access to inherited members

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class A {
5.  public:
6.      int fun(int x) { return x*x; }
7.  };
8.
9.  class B : private A {
10.     int fun(int x, int y) {
11.         return A::fun(x) + y*y;
12.     }
13. public:
14.     int pub(int x, int y) { return fun(x,y); }
15. };
16.
17. int main() {
18.     A a;
19.     B b;
20.     cout << "a.fun(3)    = " << a.fun(3)    << endl;
21.     cout << "b.pub(3,4) = " << b.pub(3,4) << endl;
22.  // cout << "b.fun(3,4) = " << b.fun(3,4) << endl;
23. }
```

The function **fun** is public in class **A**. Class **B** is derived form **A**, but privately, so it becomes private in **B**. The function **fun** is overriden in class **B**; also an additional, (public) function, **pub**, is defined. The function **pub** uses the name **fun**. What does this name correspond to?

Function **pub** is a member of **B**, so in its scope **fun** refers to the private version defined in this class (lines 10-12). In the definition, function **fun** from the base class is called, so we have to use its qualified name (line 11); otherwise, the name **fun** would refer again to the overriding method from **B** what would lead to an infinite recursion. Qualifying resolves this ambiguity and we get correctly

```
a.fun(3)    = 9
b.pub(3,4)  = 25
```

Note that we *can* call **fun** for an object of class **A** (line 20), as it is public in this class. However, we cannot invoke this function for an object of class **B** (line 22, commented out), because in class **B** function **fun** is private. We can, however, invoke **pub**, and this function, being a member of **B**, has access to private member **fun**.

As in other object-oriented languages, an object of a derived class can often be treated as an object of its base class. For example, a pointer (or a reference) to an object of a derived class can be used in context where a pointer to an object of the base class is expected. In such situation, dereferencing the pointer yields a *subobject* of the base class contained in the object of derived class. Such conversion of pointers and references is known as **upcasting** and belong to standard conversions (which do not have to be stated explicitly to be performed).

Standard upcasting of pointers and references will be performed only when derived class inherits *publicly* from its base class; in other cases upcasting must be explicit.

This mechanism is of fundamental importance for object-oriented programming and we will encounter many examples of its use.

- we can pass a reference to an object of a derived class from its copy constructors to constructors of base class;
- we can return pointers or references to objects of a derived class in functions declared as returning pointers (references) of the base class;
- binary operators defined by methods with parameter declared as a reference to object of base class will be invoked when their right argument is of a derived type. This does not apply to their left argument! Suppose, e.g., that we overloaded, as a method, the addition operator in class **A** and class **B** inherits from **A**. Then, if ***a*** is of type **A** and ***b*** is of type **B**, expression '`b+a`' will *not* work, because it is equivalent to '`b.operator+(a)`', so ***b*** is *not* referred to by a pointer or reference; however '`a+b`' will be correct — argument ***b*** will be upcast to type **A&**.

Conversions in the opposite direction — from pointer/reference to an object of the base class to pointer/reference of a derived type will *not* be performed automatically: if this is what we want, we have to make casting explicit (e.g., with the help of dynamic cast operator **dynamic_cast**), as discussed in section on conversions . This type of casting is called **downcasting**.

Note that we are talking about pointer and references, not about conversions of *objects*. For example, if a parameter of a function is declared as being of type **A** (i.e., the argument is passed by value, not by reference or by pointer), then we should *not* use arguments of a derived type **B**. Passing argument by value means that its copy has to be pushed on the stack and be of the correct size (that of objects of type **A**). This to be possible, the object would have to be “sliced”: in fact only a part of it (namely its subobject of type **A**) would be

passed, what could lead to surprising results.

Each object of a derived class contains, as a subobject, an object of its base class. Quite often we need to define a pointer or a reference to this subobject. Suppose **A** is the base class and **B** is derived from **A**. Then:

- if ***pb*** is a pointer of type **B*** pointing to an object ***b*** of type **B**, then `(A*)pb` or `static_cast<A*>(pb)` is the pointer to the subobject of type **A** contained in ***b***. Instead of ***pb*** one can use, equivalently, ***&b***;
- similarly, `(A&)b` or `static_cast<A&>(b)` is a reference to this subobject;
- if ***pa*** is a pointer of type **A*** pointing to an object ***b*** of type **B**, then `(B*)pa` or `dynamic_cast<B*>(pa)` is a pointer to object of type **B**. Dynamic cast can be used only if **A** is polymorphic, as we will discuss later on;
- similarly, `(B&)(*pa)` or `dynamic_cast<B&>(*pa)` is a reference to this object.

The conversions described above are useful when we deal with polymorphic classes, especially when defining constructors and operators in derived classes.

Consider an example:

P178: cast.cpp Convesions between object and its subobject

```
1. #include <iostream>
2. using namespace std;
3.
4. struct A {
5.     int x;
6.     int y;
7.     A() : x(1), y(2) {}
8. };
9.
10. struct B : public A {
11.     int x;  // ????
12. };
13.
14. int main() {
15.     B b, *pb = &b;
16.     b.x = 11;
17.     b.y = 12;
18.
19.     cout << "b.x=" << b.x        << " b.y="
20.          << b.y    << " b.A::x=" << b.A::x
21.          << " b.A::y=" << b.A::y << endl;
22.
23.     cout << "\n      pb->x=" <<    pb ->x   << endl;
24.     cout <<   "((A*)pb)->x=" << ((A*)pb)->x << endl;
25.     cout <<   "        b.x=" <<       b.x   << endl;
26.     cout <<   "  ((A&)b).x=" << ((A&)b).x   << endl;
27.     cout <<   "((A*)&b)->x=" << ((A*)&b)->x << endl;
28.
29.     A* pa = new B;
30.     ((B&)*pa).x = 11;
31.
32.     cout << "\n    (*pa).x=" << (*pa).x     << endl;
33.     cout <<   "((B&)*pa).x=" << ((B&)*pa).x << endl;
34.     cout <<   "      pa->x=" <<       pa->x << endl;
35.     cout <<   "((B*)pa)->x=" << ((B*)pa)->x << endl;
36.
37.     cout << "\nsizeof(b) = " << sizeof b << endl;
38.     int* t = (int*) &b;
39.     cout << t[0] << " " << t[1] << " " << t[2] << endl;
40. }
```

Class **A** has members ***x*** and ***y***. Derived class **B** defines a field ***x*** (linia 11), which overrides (shadows) inherited member of the same name (what, as far as fields — not methods — are concerned is *not* a recommended practice!). In the main program, we create an object of class **B** and we initialize its members on lines 16-17. Note that the object contains *three* members: ***x***, ***y*** and ***x*** inherited from the base class **A**. In the scope of class **B**, the name ***x*** refers to the member defined in this class (with value 11), while qualified name ***A::x*** refers to the inherited member (of value 1). Therefore, ***b.x*** is 11, but ***b.A::x*** is 1. We can see it in the printout

```
b: x=11 y=12 b.A::x=1 b.A::y=12

      pb->x=11
((A*)pb)->x=1
        b.x=11
  ((A&)b).x=1
((A*)&b)->x=1

    (*pa).x=1
((B&)*pa).x=11
      pa->x=1
((B*)pa)->x=11

sizeof(b) = 12
1 12 11
```

As we can see, after casting ***pb*** to type **A***, the value of `((A*)pb)->x` is 1, as now we refer to ***x*** from the scope of **A**, i.e., to ***x*** from the subobject of type **A** contained in the object of type **B**. Similarly, `(A&)b` is a reference to this subobject.

As ***y*** has *not* been overriden, in the scope of class **B** names ***y*** and ***A::y*** refer to exactly the same variable.

An object of class **B** is created on line 29 — note that the pointer ***pa*** to this object is of type **A***, not **B***. Now we can see an effect of downcasting. Variable ***pa*** is of type **A*** but points to an object of type **B**, therefore `pa->x` refers to member ***x*** in subobject of type **A** contained in the object ***b***. After downcasting the value of ***pa*** to type **B***, it is ***x*** from the scope of **B** which is pointed to (line 35).

The size of object of type **B** is printed on line 37. It is 12, what corresponds to three integers — ***x*** and ***y*** from the subobject of type **A** and ***x*** added in class **B** (note, however, that this fact *can* depend on architecture and compiler used). Treating the adress of the object as the address of a three-element array of integers (line 38), we can convince ourselves that indeed these three integers are 1, 12 (***x*** and ***y*** from the subobject) and 11 (***x*** added in class **B**).

It is very important to remember that

constructors and destructor are *not* inherited.

On the other hand, constructors and destructors play a very important rôle, in particular for classes with members of pointer types, where logical content of objects is not contained in them physically. Therefore, problems related to constructing objects of derived classes will be now discussed separately.

T.R. Werner, September 27, 2023; 22:32

21.2 Constructors and destructors of derived classes

Constructors are not inherited. However, when an object of a derived class is being created, a subobject of its base class has to be constructed first, so the compiler has to know what constuctor should be used. If no constructor has been defined in the derived class, then default constructor will be used for constructing the subobject of the base class (hence, it must exist!). One has to remember that

Subobjects of base class are created *before* the body of any constructor defined in the derived class is executed.

A question arises, what to do if a subobject of a base class should be created by a non-default constructor. We cannot inform the compiler about that in the body of a constructor of the derived class — by then the subobject has already to exist! The only possibility is to invoke an appropriate constructor of the base class *before* entering a constructor of the derived class — this can *only* be done from initialization list (see section on initialization lists). An explicit call to a constructor of the base class should then be put in the initialization list of every constructor of the derived class. If **A** is the base class, then in the initialization list of a derived class we add `A(...)`, where, of course, arguments should be given in place of the dots. Only a constructor of the *direct* base class can be placed in the initialization list (of the "father", but not the "grandfather" — of course, father's constructor may contain initialization list with an invocation of a grandfather's constructor).

In the following example, the base class **Point** has no default constructor:

P179: pix.cpp Invocation of base class' constructor

```
1.  struct Point {
2.      int x;
3.      int y;
4.      Point(int x, int y)
5.          : x(x), y(y)
6.      { }
7.  };
8.
9.  struct Pixel: public Point {
10.     int color;
11.     Pixel(int x, int y, int color)
12.         : Point(x,y), color(color)
13.     { }
14. };
```

Class **Pixel** inherits from **Point**. As **Point** does not have a default constructor (i.e., a constructor which can be called without arguments), class **Pixel** *has* to have at least one constructor which calls, from its initialization list, the only constructor of the base class (line 12).

Note that it is not permitted to initialize from the initialization list individual members

inherited from the base class. In the example above, something like `x(x)` would be illegal, because ***x*** is inherited from the base class. However, `color(color)` is valid, because the field ***color*** has been added in the derived class (is not inherited). If we want (or have to) initialize inherited members from initialization list of a constructor of a derived class, we have to do it by calling an appropriate constructor of the base class, as in the example above.

A constructor of a derived class can play the rôle of the copy constructor. Then, in its initialization list, we may want to call the copy constructor of the base class; otherwise the *default* constructor for the subobject of the base class will be used, what may or may not be what we want (additionally, this default constructor has to exist!). If we do want to call explicitly the *copy* constructor of the base class, what should we use as the argument? The answer is quite simple: the type of the parameter of the copy constructor of the base class **A** is **A&** (or, more often, **const A&**), so it is enough to pass a reference to the object of the derived class **B** which is the argument of the copy constructor in the derived class **B** — as we know, if a parameter of a function is of pointer or reference type, one can pass as an argument a reference or a pointer to an object of a derived class.

In the example below, there are no pointer fields but still we define default and copy constructors:

P180: cop.cpp Copy constructors

```
1. #include <iostream>
2. using namespace std;
3.
4. class A {
5.     int a;
6. public:
7.     A(const A& aa) {
8.         a = aa.a;
9.         cout << "Copy-ctor A, a = " << a << endl;
10.     }
11.
12.     A(int aa = 0) {
13.         a = aa;
14.         cout << "Def-ctor  A, a = " << a << endl;
15.     }
16.
17.     void showA() { cout << "a = " << a; }
18. };
19.
20. class B: public A {
21.     int b;
22. public:
23.     B(const B& bb)
24.         : A(bb)
25.     {
26.         b = bb.b;
27.         cout << "Copy-ctor B, b = " << b << endl;
28.     }
29.
30.     B(int bb = 1)
31.         : A(1)
32.     {
33.         b = bb;
34.         cout << "Def-ctor  B, b = " << b << endl;
35.     }
36.
37.     void showB() {
38.         showA();
39.         cout << ", b = " << b << endl;
40.     }
41. };
42.
```

```
43. int main() {
44.     B b1(2);
45.     b1.showB();
46.
47.     B b2(b1);
48.     b2.showB();
49. }
```

The copy constructor of the derived class **B** is defined on lines 23-28. In its initialization list, we call explicitly the copy constructor from the base class **A**, using, as an argument, the reference to an object of class **B**. As **B** is derived form **A**, such invocation is valid. The result is

```
Def-ctor  A, a = 1
Def-ctor  B, b = 2
a = 1, b = 2
Copy-ctor A, a = 1
Copy-ctor B, b = 2
a = 1, b = 2
```

We can remove the call to copy constructor of **A** by commenting out line 24. Then the default constructor of **A** will be used for building the subobject of type **A** contained in the object of class **B** which is just being created. We can see it from the printout of the program after commenting out line 24:

```
Def-ctor  A, a = 1
Def-ctor  B, b = 2
a = 1, b = 2
Def-ctor  A, a = 0
Copy-ctor B, b = 2
a = 0, b = 2
```

Looking at the printouts, one can see that

Constructors are called from "top" to "bottom": first from the base class, then from the derived class. If a class inherits from several classes, their constructors are invoked in the order they are mentioned in the inheritance list in the definition of the derived class.

Therefore, if **C** inherits from **B**, which in turn inherits from **A**, then, during creation of an object of class **C**, constructor of **A** will be invoked first, followed by constructor of **B**, finally followed by constructor of **C**. If there are members of object types, they are created, in the order as they appear in the definition of the class, *after* the subobject of the base classes have been completely created but *before* the body of a constructor is executed.

As we have said, destructors are not inherited either. If they are defined,

destructors are invoked in the order opposite to the order in which constructors are called.

Hence, during the destruction of an object, its "own" destructor is called first, then non-inherited member objects are deleted, followed by the same procedure for subobjects of base classes. This is illustrated by the following program:

P181: condes.cpp Order of invocation of constructors and destructors

```
1. #include <iostream>
2. using namespace std;
3.
4. struct K {
5.     char k;
6.     K(char kk = 'k') {
7.         k = kk;
8.         cout << "Ctor K\n";
9.     }
10.
11.     ~K() {
12.         cout << "Dtor K\n";
13.     }
14. };
15.
16. struct A {
17.     char a;
18.     A(char aa = 'a') {
19.         a = aa;
20.         cout << "Ctor A\n";
21.     }
22.
23.     ~A() {
24.         cout << "Dtor A\n";
25.     }
26. };
27.
28. struct B: public A {
29.     char b;
30.     K    k;
31.     B(char bb = 'b') : A(bb) {
32.         b = bb;
33.         cout << "Ctor B\n";
34.     }
35.
36.     ~B() {
37.         cout << "Dtor B\n";
38.     }
39. };
40.
41. struct C: public B {
42.     char c;
43.     C(char cc = 'c') : B(cc) {
44.         c = cc;
45.         cout << "Ctor C\n";
46.     }
47.
48.     ~C() {
49.         cout << "Dtor C\n";
50.     }
51. };
52.
53. int main() {
54.     C c;
55. }
```

We have a hierarchy of classes here: **A** ← **B** ← **C** (arrows, as usually, point from derived classes to their base classes). Additionally, class **B** has an object member of type **K**. In **main** we just create an object of type **C**, which is destroyed when **main** exits. The printout is:

```
Ctor A
Ctor K
Ctor B
Ctor C
Dtor C
Dtor B
Dtor K
Dtor A
```

One can see that a suboject of class **A** is created first, and then a subobject of class **B**. This subobjects contains, as a member, an object of class **K** — as the printout shows, it is created *before* invoking constructor of **B**. At the very end, constructor of **C** is called.

Destructors are called in the reverse order. In particular, when the subobject of **B** is deleted, destructor is called first and only then its member of class **K** is destroyed.

Prev: 21.1 Fundamentals of inheritance **Up:** 21. Inheritance and polymorphism **Next:** 21.3 Assignment operator in derived

21.3 Assignment operator in derived classes

It is not always trivial to define (overload) properly the assignment operator in a derived class. It may happen that there is a non-private assignment operator defined in base class. If a derived class does not overload the assignment operator, then the version from the base class will be used to perform the assignment of the inherited subobject; however, default assignment (field-by-field) will be used to assign non-inherited members (what can be not what we want, if these are pointer members). For example, the program below

P182: inhas.cpp Inheriting assignment operator

```
1. #include <iostream>
2. using namespace std;
3.
4. struct A {
5.     char a;
6.     A(char aa = 'a') {
7.         a = aa;
8.     }
9.
10.    A& operator=(const A& aa) {
11.        a = aa.a;
12.        cout << "A::operator=()\n";
13.        return *this;
14.    }
15. };
16.
17. struct B: public A {
18.    char b;
19.    B(char bb = 'b') : A(bb) {
20.        b = bb;
21.    }
22. };
23.
24. int main() {
25.    B b1(1),b2(2);
26.    b1 = b2;
27. }
```

prints

```
A::operator=()
```

what shows that indeed an asignments of objects of class **B** invokes **operator=()** from the base class **A** (and this operator, of course, does not “know” about members added in class **B**).

If we redefine assignment operator in a subclass, we have to do all the work there: invoke assignment from the base class *explicitly* (to assign the inherited part of the object) and take care about members added in the subclass. Note that in this case **operator=()** from the base class will *not* be called “by itself”.

How to invoke **operator=()** from the base class in a method of a derived class? We can do it explicitly “by name” (qualified with the name of the base class). Or, we can get the same

effect by simply assigning to ***this***, if the compiler is informed that ***this*** should be treated as an object of the base class: in this way we can force the compiler to use **operator=()** from the base class.

It may sound complicated, but is quite simple. Suppose that **B** inherits from **A** and the assignment operator is overloaded in **A**. The overloading method **operator=()** has parameter of type **A&**, but, as we know, in such cases one can pass a reference to an object of a subclass, which will be the object of a subclass which is just being assigned from (the one appearing on the right hand side of the assignment). Note that this mechanism would not work for arguments passed by value (if the parameter of the **operator=()** method in the base class were declared, legally, as **A** or **const A** instead of **A&** or **const A&**). Therefore, redefining **operator=()** in a derived class, one can write:

```
B& B::operator=(const B& b) {
    this->A::operator=(b);
    // ...
    return *this;
}
```

or, with the same effect,

```
B& B::operator=(const B& b) {
    (A&)(*this) = b;
    // ...
    return *this;
}
```

In this case, the reference to the object ***this*** of class **B** has been upcasted to type **A&**, so **operator=()** from the base class **A** will be used, as in the previous case. Of course, instead of old-style casting `(A&)` we could have used the C++ style

```
static_cast<A&>(*this) = b;
```

Finally, one can cast the pointer **this** and then dereference it to obtain something of type **A** to enforce the use of **operator=()** from the base class:

```
B& B::operator=(const B& b) {
    *((A*)this) = b;
    // ...
    return *this;
}
```

although this form looks less comprehensible.

In the program below, we overload assignment operators and define constructors, including copy-constructors, and destructors for two classes: class **Person** and derived from it class **Employee**. Both classes have pointer fields, so correct copy-constructors, destructors and assignment operators are necessary.

P183: inhe.cpp Classes with pointer fields: inheritance

```
1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4.
5. class Person {
6.     char* name;
7. public:
8.     Person()
9.         : name(strcpy(new char[14], "NameUnknown"))
10.     {
11.         cout << "Default Ctor Person: "
```

```
12.                 << name << endl;
13.     }
14.
15.     Person(const char* n)
16.         : name(strcpy(new char[strlen(n)+1], n))
17.     {
18.         cout << "Ctor char* Person: " << name << endl;
19.     }
20.
21.     Person(const Person& os)
22.         : name(strcpy(new char[strlen(os.name)+1],
23.                                         os.name))
24.     {
25.         cout << "Copy-Ctor Person: "
26.              << name << endl;
27.     }
28.
29.     Person& operator=(const Person& os) {
30.         if ( this != &os ) {
31.             delete [] name;
32.             name = strcpy(new char[strlen(os.name)+1],
33.                                             os.name);
34.             cout << "Assignment Person: "
35.                  << name << endl;
36.         }
37.         return *this;
38.     }
39.
40.     ~Person() {
41.         cout << "Deleting Person: " << name << endl;
42.         delete [] name;
43.     }
44.
45.     const char* getName() const { return name; }
46. };
47.
48. class Eployee : public Person {
49.     char* position;
50. public:
51.     Eployee()
52.         : position(strcpy(new char[14], "PosUnknown"))
53.     {
54.         cout << "Default Ctor Eployee: "
55.              << position << endl;
56.     }
57.
58.     Eployee(const char* s, const char* n)
59.       : Person(n),position(strcpy(new char[strlen(s)+1],s))
60.     {
61.         cout << "Ctor char* char* Eployee: "
62.              << position << endl;
63.     }
64.
65.     Eployee(const Eployee& empl)
66.         : Person(empl), position(strcpy(new
67.               char[strlen(empl.position)+1],empl.position))
68.     {
69.         cout << "Copy-Ctor Eployee: "
70.              << position << endl;
71.     }
72.
73.     Eployee& operator=(const Eployee& empl) {
74.         if ( this != &empl ) {
75.             (Person&)(*this) = empl;
76.             delete [] position;
77.             position = strcpy(new
78.                 char[strlen(empl.position)+1],
79.                 empl.position);
80.             cout << "Assignment Eployee: "
81.                  << position << endl;
82.         }
83.         return *this;
84.     }
85.
86.     ~Eployee() {
87.         cout << "Deleting Eployee: " << position << endl;
```

```
88.         delete [] position;
89.     }
90.
91.     const char* getPosition() const { return position; }
92. };
93.
94. int main() {
95.     cout << "\nMain: Creating object nobody" << endl;
96.     Eployee nobody;
97.     cout << "Main: object nobody created: "
98.          << nobody.getPosition() << " "
99.          << nobody.getName()   << endl;
100.
101.     cout << "\nMain: Creating object brown" << endl;
102.     Eployee brown("Boss", "Brown");
103.     cout << "Main: object brown created: "
104.          << brown.getPosition() << " "
105.          << brown.getName()   << endl;
106.
107.     cout << "\nMain: Copying brown -> copy" << endl;
108.     Eployee copy(brown);
109.     cout << "Main: object copy created: "
110.          << copy.getPosition() << " "
111.          << copy.getName()   << endl;
112.
113.     cout << "\nMain: Assignment nobody = copy" << endl;
114.     nobody = copy;
115.     cout << "Main: nobody = copy assigned: "
116.          << nobody.getPosition() << " "
117.          << nobody.getName() << endl << endl;
118. }
```

Note that on lines 59 and 66 we explicitly invoke base class constructors from the initialization lists of the constructors in the derived class **Employee**. Similarly, on line 75, inside the method overloading the assignment operator in the derived class, we explicitly call the analogous method from the base class. The output of the program:

```
Main: Creating object nobody
Default Ctor Person: NameUnknown
Default Ctor Eployee: PosUnknown
Main: object nobody created: PosUnknown NameUnknown

Main: Creating object brown
Ctor char* Person: Brown
Ctor char* char* Eployee: Boss
Main: object brown created: Boss Brown

Main: Copying brown -> copy
Copy-Ctor Person: Brown
Copy-Ctor Eployee: Boss
Main: object copy created: Boss Brown

Main: Assignment nobody = copy
Assignment Person: Brown
Assignment Eployee: Boss
Main: nobody = copy assigned: Boss Brown

Deleting Eployee: Boss
Deleting Person: Brown
Deleting Eployee: Boss
Deleting Person: Brown
Deleting Eployee: Boss
Deleting Person: Brown
```

demonstrates the order in which constructors, destructors and methods overloading the assignment operator are called. When the program exits, all three objects are deleted; they all contain the same data but they are three independent objects, what is confirmed by the fact that all destructors succeed. One can see that constructors of the base class **Person** are called first, followed by invocations of constructors of the derived class **Employee**. During destructions the order is reversed: destructors of the derived class are called first.

21.4 Virtual methods and polymorphism

It is possible to define in the derived classes methods which have their signature and return type (see chapter on functions) exactly the same as methods defined in the base class. This is *not* overloading, but **overriding** (overloaded functions have the same name but *different* signatures).

Imagine the following situation:

```
class A {
    // ...
    void fun() { ... }
    // ...
};

class B : public A {
    // ...
    void fun() { ... }
    // ...
};
```

Let us define:

```
A a, *pa  = new A, *pab = new B,
     &raa = a,     &rab = *pab;
```

Now

- ***a*** in an object of type **A**;
- ***wa*** is a pointer of type **A*** to an object of class **A**. We say that the *static* type of the object pointed to by ***wa*** is **A**, and its *dynamic* type is also **A**;
- ***pab*** is a pointer of type **A*** to an object of type **B**. Static type of the object pointed to by ***pab*** is **A**, but its true, dynamic type is **B**;
- ***raa*** is a reference of type **A&** to an object of type **A**. Static type of the object referenced by ***raa*** is **A**, its dynamic type is also **A**;
- ***rab*** is a reference of type **A&** to an object of type **B**. Static type of the object referenced by ***rab*** is **A**, but its true, dynamic type is **A**.

Recall that

> **Static type of an object pointed to by a pointer or referenced by a reference is determined by the declaration of this pointer or reference. By its *dynamic* type we mean its real type. Dynamic type usually can only be determined during execution of the program, while its static type is known during compilation.**

Suppose now that we invoke function **fun** using variables ***a***, ***pa***, ***pab***, ***raa*** and ***rab*** to see what method will be called: the one from class **A** or the one from class **B**. The answer is: in all cases, i.e.,

```
a.fun(); pa->fun(); pab->fun(); raa.fun(); rab.fun();
```

it will be the method from class **A**, although in the third and fifth calls, the dynamic type of the object is **B**. As one can see, it is the *static* type which decides here (even if the method *has* been overriden in the derived class, as it is the case in our example).

For those who know Java or other object-oriented languages, this can be a surprise. In those languages, what decides is the *dynamic* type of the object pointed to (referenced) by a pointer (reference). A pointer can be of the base class type, but if we invoke a method for an object pointed to by this pointer and this object happens to be an object of a derived class, the method will be searched for in the scope of the derived class. We say that such a method is **virtual**. Therefore, in Java all non-private, non-final methods *are* virtual. Classes with virtual methods are called **polymorphic** (because the effect of refering an object through a pointer of this class' type depends on the true type of the object pointed to, so it can have “many shapes”).

However, there is a price one has to pay for polymorphism. For non-polymorphic classes, which methods should be called is determined on the basis of its static type: it is known during compilation, so the invocations can be “hardwired” into the executable. We call it **early binding**. For polymorphic calls, a particular method that should be invoked can only be looked up at run time, after checking the true type of the object (**late binding**). This, of course, takes time and deteriorates efficiency of the program. Moreover, for this to be possible, each object of a polymorphic type has to hold information (address) of a special table containing addresses of different versions of the method. Consequently, objects of polymorphic types are bigger than those of non-polymorphic types. As one can see, polymorphism leads to both execution-time and memory-consumption overheads.

For that reason, classes in C++ are by default *non-polymorphic*, and methods defined in the classes are *non-virtual*. However, the programmer can “switch on” polymorphism for any class he/she defines — this has to be done individually for every class that he/she wants to behave in the polymorphic way.

For a class to be polymorphic it is enough to declare at least one of its method as virtual (it can be the class' destructor, but *not* a constructor — those can never be virtual).

If a method has been declared virtual in a base class, all methods which override it in derived classes are virtual as well, even without any special declaration of this fact in definitions of the derived classes (not only “sons”, but also “grandsons”, grand-grandsons,...). However, declaring them again as virtual in derived classes is not an error, and in fact is recommended as it makes the code more comprehensible for the reader.

A method can be made virtual by adding the modifier **virtual** in its declaration (as we have already said, in derived classes this can, but does not have to be repeated); for example:

```
class A {
    // ...
    virtual double fun(int,int);
    // ...
};

class B : public A {
    // ...
    double fun(int,int);
    // ...
};
```

Suppose we have defined, as previously,

```
A a, *pa  = new A, *pab = new B,
     &raa = a,     &rab = *pab;
```

Now classes are polymorphic, method **fun** is virtual, so the mechanism of polymorphism (late binding) will work. Invocations

```
a.fun(), pa->fun(), pab->fun(), raa.fun(), rab.fun()
```

refer to the method **fun** from class **B** in the third and fifth case, because:

- invocation is through a pointer or a reference of the base class type (**A*** or **A&**);
- the true type of the object pointed to (referred) is the derived type **B**;
- method **fun** is virtual;
- method **fun** has been overriden in the drived class **B**.

Nothing wrong would happen if the method **fun** were not overriden in class **B**. The version of **fun** visible in **B** would have been invoked anyway, but without overriding it would be just the version from **A** inherited by **B**.

Even if a virtual method *has* been overriden in a derived class, we still can access the version from the base class using a pointer (reference) of the derived class' type. It is then necessary to use full, qualified name of the method — qualifying the name with class' name switches off polymorphism and results in early binding; invocations

```
pab->A::fun();   rab.A::fun();
```

call the method **fun** from the base class **A** even if this method is virtual and has been overriden in **B**. If so, the following is also possible:

```
b.A::fun()
```

We call the version of **fun** from class **A** for the object of type **B** directly, without using pointers or references. This will tell the compiler not to use overriding version of **fun** from **B**, but rather the inherited ("original") version from **A**. The opposite, of course, is *not* possible: one cannot, even using qualified names, call a method from **B** using the name of an *object* of base type **A** (and not a pointer or reference):

```
a.B::fun() // WRONG !!!
```

would be illegal.

Overriding a method in a subclass, we can declare it with narrower accessibility than the version from the base class (this is opposite to what we know from Java, where one can make accessibility of overriding methods wider, but never narrower).

Suppose we call an overriden method from **B**, but through a pointer of type **A*** (pointing to an object of class **B**). What will be its accessibility, if it has been made narrower in class **B**? The answer is: accessibility will be that declared in class **A**. The accessibility is determined statically, at compile time, so the static type of the object will be used. If a method is **public** in the base class, then it will be publicly assessible through pointers of the base class' type even if it is **private** in the derived class! Consider an example:

P184: figure.cpp Accessibility of virtual methods

```
1. #include <iostream>
```

```
 2. using namespace std;
 3.
 4. class Figure {
 5. protected:
 6.     int  height;
 7. public:
 8.     Figure(int height = 0) : height(height)
 9.     { }
10.
11.     virtual void what() const {
12.         cout << "Figure: h=" << height <<endl;
13.     }
14. };
15.
16. class Rectangle : public Figure {
17. private:
18.     int  base;
19.     void what() const {
20.         cout << "Rectangle: (h,b)=(" << height
21.              << "," << base << ")\n";
22.     }
23. public:
24.     Rectangle(int height = 0, int base = 0)
25.         : Figure(height), base(base)
26.     { }
27. };
28.
29. int main() {
30.     Figure    *f = new Rectangle(4,5)  , &rf = *f;
31.     Rectangle *p = new Rectangle(40,50);
32.
33.       // what private in Rectangle, but not in Figure!
34.     f->what();                          // Rectangle
35.     rf.what();                          // Rectangle
36.
37.       // p->what(); wrong, since what private in Rectangle
38.       // But those two lines are OK!
39.     ((Figure*)p)->what();               // Rectangle
40.     ((Figure&)*p).what();               // Rectangle
41.
42.       // OK: public version from the base class Figure
43.     p->Figure::what();                  // Figure
44. }
```

Virtual method **what** is public in class **Figure** (line 11). In class **Rectangle**, this method is overriden as **private** (line 19). Variables ***f*** and ***rf*** are of type **Figure*** and **Figure&** respectively, but the referenced object is of class **Rectangle** (line 30). Therefore, its *dynamic* type is **Rectangle**, but the static type is **Figure**. Note the invocations from lines 34 and 35. We call the method **what** from class **Rectangle**, because the method is virtual and it is the dynamic type which decides which method will be called. In this class **what** is private, but accessibility is determined by the static type of the object, which is **Figure**; the method was public there, so the calls succeed:

```
Rectangle: (h,b)=(4,5)
Rectangle: (h,b)=(4,5)
Rectangle: (h,b)=(40,50)
Rectangle: (h,b)=(40,50)
Figure: h=40
```

Invocation from the commented out line 37 would fail. Here the static type of the object referenced by ***p*** is **Rectangle**, and **what** is private in this class.

Note that both calls from lines 39 and 40 are valid. The type of the variable ***p*** is a pointer to an object of the derived class (and indeed, ***p*** points to an object of this type), but its value is upcast to the type **Figure***, so the static type of the object becomes **Figure**, and in this class **what** is public — an analogous mechanism was used for the reference ***rf***.

The last line of the printout comes from the statement on line 43 of the program. Both the

static and dynamic types of the object pointed to by ***p*** is **Rectangle**, but we have qualified the name of the method **what** by the base class' name **Figure**. The call is not polymorphic: we are explicitly calling the method from the base class, where it is public.

To convince ourselves that polymorphism is not “for free” and should be avoided if it is not needed, let us consider the following example:

P185: polysiz.cpp Polymorphism and size of objects

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class A {
5.      int i;
6.  public:
7.      A() : i(0)
8.      { }
9.  };
10.
11. class B {
12.     int i;
13. public:
14.     B() : i(0)
15.     { }
16.     ~B()
17.     { }
18. };
19.
20. class C {
21.     int i;
22. public:
23.     C() : i(0)
24.     { }
25.     virtual ~C()
26.     { }
27. };
28.
29. int main() {
30.    cout << "sizeof(A): " << sizeof(A) << endl;
31.    cout << "sizeof(B): " << sizeof(B) << endl;
32.    cout << "sizeof(C): " << sizeof(C) << endl;
33. }
```

Class **B** differs from **A** only in that id defines a destructor (unnecessary in this class). The destructor is not virtual, so the class is not polymorphic. Class **C** is like **B**, but its destructor *is* virtual, so the class **C** becomes polymorphic. We print sizes of objects of the three classes on lines 30-32:

```
sizeof(A): 4
sizeof(B): 4
sizeof(C): 8
```

As one can see, adding a method (destructor in this case) did not make the objects larger; adding polymorphism, however, leads to larger objects (by four bytes). For this simple class it means that objects are by 100%100% larger! Many classes contain mainly pointer-type fields, so adding even 4 bytes to every object can lead to quite substantial overhead.

21.5 Abstract classes

In C++, as in many other object-oriented languages (like Java), one can define **abstract classes**, i.e., classes in which methods are declared, but not defined. Such methods should be, of course, virtual — they are then implemented (defined) in derived classes (which then become **concrete** classes). One cannot create objects of abstract classes: they usually serve as a definition of an interface, that is a set of methods, or “messages”, that we want to be implemented in various ways by many concrete (nonabstract) derived classes.

One can, of course, instantiate (create objects of) derived classes, in which virtual methods declared but not defined in the base class are overriden providing a concrete implementation. What is very important, one can refer to these object through pointers and references of the static type corresponding to the abstract base class.

A method is declared as **pure virtual** (without any implementation) by replacing its body by '`=0`', e.g.:

```
virtual void fun(int i) = 0;
```

In this way we inform the compiler that the method can remain undefined: the whole class becomes then abstract.

Actually, pure virtual method *can* be defined. Nevertheless, even if a definition has been provided, the class remains abstract and cannot be instantiated. Derived classes, to become concrete, have to override such a method anyway. The version from the base class is still assessible by refering it through objects of derived classes and using qualified name ('`TheBaseClass::fun()`').

Let us consider an example:

P186: virt.cpp Pure virtual methods

```
1.  #include <iostream>
2.  #include <cmath>     // atan
3.  using std::ostream; using std::cout; using std::endl;
4.
5.  class Figure {
6.  protected:
7.      static const double PI;
8.  public:
9.      virtual double getArea() const      = 0;
10.     virtual double getPerimeter() const = 0;
11.     virtual void   info(ostream&) const = 0;
12.     static  double totalArea(Figure* arr[], int size) {
13.         double sum = 0;
14.         for (int i = 0; i < size; ++i)
15.             sum += arr[i]->getArea();
16.         return sum;
17.     }
18.     static  Figure* maxPerim(Figure* arr[], int size) {
19.         int ind = 0;
20.         for (int i = 0; i < size; ++i)
```

```
21.             if (arr[i]->getPerimeter() >
22.                     arr[ind]->getPerimeter())
23.                 ind = i;
24.         return arr[ind];
25.     }
26. };
27. const double Figure::PI = 4*atan(1.);
28. void Figure::info(ostream& str) const {
29.     str << "Figure: ";
30. }
31.
32. class Circle : public Figure {
33.     double radius;
34. public:
35.     Circle(double r) : radius(r){ }
36.     double getArea() const      { return PI*radius*radius; }
37.     double getPerimeter() const { return 2*PI*radius; }
38.     void   info(ostream& str) const {
39.         Figure::info(str);
40.         str << "circle with radius " << radius;
41.     }
42. };
43.
44. class Square : public Figure {
45.     double side;
46. public:
47.     Square(double s) : side(s){ }
48.     double getArea() const      { return side*side; }
49.     double getPerimeter() const { return 4*side; }
50.     void   info(ostream& str) const {
51.         Figure::info(str);
52.         str << "square with side   " << side;
53.     }
54. };
55.
56. int main() {
57.     Figure* arr[] = { new Circle(1.), new Square(1.),
58.                       new Circle(2.), new Square(3.)
59.                     };
60.     int size = sizeof(arr)/sizeof(arr[0]);
61.     for (int i = 0; i < size; ++i) {
62.         arr[i]->info(cout);
63.         cout << endl;
64.     }
65.     Figure* maxper = Figure::maxPerim(arr,size);
66.     cout << "Total area: " << Figure::totalArea(arr,size)
67.          << "\nFigure with maximum perimeter: ";
68.     maxper->info(cout);
69.     cout << "\n has perimeter "
70.          << maxper->getPerimeter() << endl;
71.     for (size_t i=0; i < std::size(tab); ++i) delete tab[i];
72. }
```

Class **Figure** is an abstract class. This is quite natural, because it would be impossible to implement in a sensible way methods which calculate areas or perimeters of figures “in general”. We need a concrete figure, like a square or a circle, to be able to calculate such quantities. Therefore, implementation of these methods is deferred to concrete derived classes. Such hierarchy of classes has many advantages. Notice that two static functions has been defined in the class **Figure**. Their parameter is of type *an array of pointers to figures*. What figures? Not objects of class **Figure**, because this class is abstract and it is impossible to create object of such a class. However, these will be objects of concrete classes derived form **Figure**, so *for sure* all virtual methods *will* be implemented. Therefore, invoking them through pointers or references will succeed, no matter what the type of these object will be. Note also that the objects pointed to by pointers from the array are not necessarily of the same type — in our example they are **Circle** s and **Square** s. We can even add new classes, e.g., **Triangle** s, and the base class, in particular functions **totalArea** and **maxPerim**, will not need any modifications! So, abstract classes can define an interface for a whole family of classes, some of them, perhaps, not yet written.

The method **info** is declared as pure virtual class, but actually it *has been* implemented (lines 28-30). Still, it remains pure virtual and every concrete derived class *has* to override it. The implementation from the base class, however, can be accessed by its full qualified name (lines 39 and 51).

Let us consider another example of pure abstract class: it defines an interface for creating and operating on stacks (of integers).

P187: stacks.cpp Stack interface with a factory method

```
1. #include <iostream>
2. using namespace std;
3.
4. class STACK
5. {
6. public:
7.     virtual void push(int) = 0;
8.     virtual int pop()      = 0;
9.     virtual bool empty() const = 0;
10.    static STACK* getInstance(int);
11.    virtual ~STACK() { }
12. };
13.
14. class ListStack: public STACK {
15.
16.    struct Node {
17.        int   data;
18.        Node* next;
19.        Node(int data, Node* next)
20.            : data(data), next(next)
21.        { }
22.    };
23.
24.    Node* head;
25.
26.    ListStack() {
27.        head = nullptr;
28.        cerr << "Creating ListStack" << endl;
29.    }
30.
31.    ListStack(const ListStack&) { }
32.    void operator=(ListStack&) { }
33.
34. public:
35.    friend STACK* STACK::getInstance(int);
36.
37.    int pop() {
38.        int   data = head->data;
39.        Node* temp = head->next;
40.        delete head;
41.        head = temp;
42.        return data;
43.    }
44.
45.    void push(int data) {
46.        head = new Node(data, head);
47.    }
48.
49.    bool empty() const {
50.        return head == nullptr;
51.    }
52.
53.    ~ListStack() {
54.        cerr << "Deleting ListStack" << endl;
55.        while (head) {
56.            Node* node = head;
57.            head = head->next;
58.            cerr << " deleting node " << node->data <<endl;
```

```
59.             delete node;
60.         }
61.     }
62. };
63.
64. class ArrayStack : public STACK {
65.
66.     int  top;
67.     int* arr;
68.     enum {MAX_SIZE = 100};
69.
70.     ArrayStack() {
71.         top = 0;
72.         arr = new int[MAX_SIZE];
73.         cerr << "Creating ArrayStack" << endl;
74.     }
75.
76.     ArrayStack(const ArrayStack&) { }
77.     void operator=(ArrayStack&) { }
78.
79. public:
80.     friend STACK* STACK::getInstance(int);
81.
82.     void push(int data) {
83.         arr[top++] = data;
84.     }
85.
86.     int pop() {
87.         return arr[--top];
88.     }
89.
90.     bool empty() const {
91.         return top == 0;
92.     }
93.
94.     ~ArrayStack() {
95.         cerr << "Deleting ArrayStack with " << top
96.              << " elements remaining" << endl;
97.         delete [] arr;
98.     }
99. };
100.
101. STACK* STACK::getInstance(int size) {
102.     if (size > 100)
103.         return new ListStack();
104.     else
105.         return new ArrayStack();
106. }
107.
108. int main() {
109.
110.     STACK* stack;
111.
112.     stack = STACK::getInstance(120);
113.     stack->push(1);
114.     stack->push(2);
115.     stack->push(3);
116.     stack->push(4);
117.     cerr << "pop " << stack->pop() << endl;
118.     cerr << "pop " << stack->pop() << endl;
119.     delete stack;
120.
121.     stack = STACK::getInstance(50);
122.     stack->push(1);
123.     stack->push(2);
124.     stack->push(3);
125.     stack->push(4);
126.     cerr << "pop " << stack->pop() << endl;
127.     cerr << "pop " << stack->pop() << endl;
128.     delete stack;
129. }
```

Class **STACK** declares a set of typical methods for handling stacks and defines one static function **getInstance**, which returns the pointer to a stack. Non-static methods have no

implementation: that is provided in two derived classes: **ListStack** and **ArrayStack**. The former implements stack as a singly linked list, the latter as an array. Constructors of both classes are private: the only way to instatiate these classes is by using the static factory function **getInstance** from the base class — it can do it since it is declared as a friend of the derived classes (lines 35 and 80). The function creates an object either of class **ListStack** or of class **ArrayStack**, depending on the requested size of the stack: the array implementation is chosen for small stacks and the list implementation for larger stacks (lines 101-106). Since the function creates new objects of classes **ListStack** and **ArrayStack**, it must be defined *after* the definition of the derived classes, when the compiler already knows that neither of them remained abstract.

The **main** function is a client of class **STACK**. Two objects are created, both of static type **STACK**, but of different dynamic types, as one can see from the printout

```
Creating ListStack
pop 4
pop 3
Deleting ListStack
 deleting node 2
 deleting node 1
Creating ArrayStack
pop 4
pop 3
Deleting ArrayStack with 2 elements remaining
```

Notice that both stacks are used in the same way: essentially, the client does not have to know of what the true the object are. Actually, he does not even know that these two object are or of different type.

Copy-constructor and assignment operator in both concrete classes have been defined as private (lines 31-32 and 76-77). This makes impossible copying and assigning polymorphic objects representing stacks, what would not make much sense, as objects of classes **ListStack** and **ArrayStack** are completely different (they have different fields, and even their sizes are different).

Prev: 21.4 Virtual methods and polymorphism **Up:** 21. Inheritance and polymorphism **Next:** 21.6 Virtual destructors

21.6 Virtual destructors

Generally, destructor of a class which is likely to be publicly subclassed should be made virtual. The reason is the following. Suppose we have a pointer of the base class type, but pointing to an object of a derived class. What will happen if we call the destructor (using **delete**) through this pointer? If the base-class destructor is *not* virtual, we will have a problem: this situation will corespond to early binding, so the destructor from the base class will be invoked. Of course, it knows nothing about, for example, fields or resources added in a subclass. If, however, the destructor *is* virtual, we will have late binding and the invocation will be polymorphic: the destructor from the true class of the object will be called. After that, the destructor from the base class will be invoked anyway, according to the usual rules of destroying objects.

In the following example, an object of the derived class **Full** is pointed to by pointer ***person*** of the static type corresponding to the base class **Name***.

P188: virdes.cpp Virtual destructor

```
1. #include <iostream>
2. using namespace std;
3.
4. class Name {
5.     char* name;
6. public:
7.     Name(const char* n)
8.         : name(strcpy(new char[strlen(n)+1], n))
9.     {
10.         cout << "Ctor Name: " << name << endl;
11.     }
12.
13.     virtual
14.     ~Name() {
15.         cout << "Dtor Name: " << name << endl;
16.         delete [] name;
17.     }
18. };
19.
20. class Full : public Name {
21.     char* first;
22. public:
23.     Full(const char* i, const char* n)
24.         : Name(n),
25.           first(strcpy(new char[strlen(i)+1], i))
26.     {
27.         cout << "Ctor Full, first: " << first << endl;
28.     }
29.
30.     ~Full() {
31.         cout << "Dtor Full, first: " << first << endl;
32.         delete [] first;
33.     }
34. };
35.
36. int main() {
37.     Name* person = new Full("John", "McGuire");
38.     delete person;
39. }
```

However, due to polymorphism, when the object is deleted (line 38), the destructor from the true class of the object (i.e., **Full**) will be invoked. It will release memory allocated for the first name. Then the destructor from **Name** will be called on the subobject of the base class contained in the object, deallocating memory occupied by the last name:

```
Ctor Name: McGuire
Ctor Full, first: John
Dtor Full, first: John
Dtor Name: McGuire
```

Commenting out line 13, we would make the destructor non-virtual. Then **delete** from line 38 would result in calling destructor from the base class only, without releasing the memory occupied by the first name: **Nazwisko**:

```
Ctor Name: McGuire
Ctor Full, first: John
Dtor Name: McGuire
```

Note that there is some kind of inconsistency here: destructor **~Full** in the derived class overrides **~Name** from the base class, although its name is different — destructors are exceptional in this respect: in all other cases the name of an overriding method must be the same as the name of the method being overriden.

Prev: 21.5 Abstract classes **Up:** 21. Inheritance and polymorphism **Next:** 21.7 Multiple inheritance

21.7 Multiple inheritance

Classes in C++ can inherit from (be a subclass of) several base classes. It makes very complicated hierarchies of inheritance possible; one has to remember, however, that such hierarchies can quickly become so complicated that even their author cannot handle them any more. Therefore, **multiple inheritance** (or **multibase inheritance**) should normally be avoided, if there is no important reason to introduce it. Below we will introduce multibase inheritance without going into details, which can be found in more advanced text books.

All the base classes of a class have to be listed in the inheritance list, separated by commas. For each of them, one should declare accessibility level (**private**, **protected** or **public**). If omitted, it is assumed to be **private** if keyword **class** has been used at the beginning of the class' declaration, and **public** for classes declared with keyword **struct**. All classes which appear in the inheritance list has to be already known to the compiler — a forward declaration of them is not enough.

Therefore,

```
class C : public A, B {
    // ...
};
```

declares class **C** which inherits from **A** (publicly) and from **B** (privately). In principle, it is allowed for the two base classes to contain members of the same name. In the derived class, we can access these names by qualifying them with the name of their classes (and a double colon). However, as we remember, qualifying names blocks polymorphism, so it can be dangerous.

Objects of derived classes contain subobjects of all their base classes. When an object of a derived type is created, subobjects of its base classes will have to be created (in the order of their appearance in the inheritance list) before entering derived class' constructor — their constructors can be therefore called explicitly from initialization lists of the derived class' constructors (default constructors will be used otherwise).

In the following example, we define an abstract class **Printable** which describes a simple functionality of “being printable”. The class has one field determining an output stream and one pure virtual method **print**. We also define a standard class **Person**, inheriting from **Printable** and, consequently, defining the method **print**. Note that on the initialization list of its constructor we explicitly call the constructor of **Printable** (line 21) passing as an argument an appropriate stream (***cout*** by default). Next, we define abstract class **Figure** and its two subclasses **Circle** and **Square**. They both implement the method **print**, but only **Circle** inherits also from **Printable**, so *two* base classes appear on the initialization list of its constructor (line 43).

P189: multbase.cpp Multibase inheritance

```
1. #include <iostream>
2. #include <string>
3. #include <cmath>
4. using namespace std;
5.
6. class Printable {
7. protected:
8.     ostream& str;
9. public:
10.     Printable(ostream& str)
11.         : str(str)
12.     { }
13.     virtual void print() const = 0;
14. };
15.
16. class Person : public Printable {
17.     string name;
18.     int   birth;
19. public:
20.     Person(string n, int b, ostream& str = cout)
21.         : Printable(str), name(n), birth(b)
22.     { }
23.
24.     void print() const {
25.         str << name + " (" << birth << ")" << endl;
26.     }
27. };
28.
29. class Figure {
30. protected:
31.     static const double PI;
32.     string name;
33. public:
34.     virtual double getArea() const = 0;
35.     Figure(string name) : name(name) { }
36. };
37. const double Figure::PI = 4*atan(1.);
38.
39. class Circle : public Figure, public Printable {
40.     double radius;
41. public:
42.     Circle(string n, double r, ostream& str = cout)
43.         : Figure(n), Printable(str), radius(r)
44.     { }
45.     double getArea() const { return PI*radius*radius; }
46.     void   print() const {
47.         str << "circle " << name << " with radius "
48.             << radius << " and area " << getArea() << endl;
49.     }
50. };
51.
52. class Square : public Figure {
53.     double side;
54. public:
55.     Square(string n, double s)
56.         : Figure(n), side(s)
57.     { }
58.     double getArea() const { return side*side; }
59.     void print() const {
60.         cout << "square " << name << " with side "
61.              << side << " and area " << getArea() << endl;
62.     }
63. };
64.
65. void printArray(Printable* arr[], int size) {
66.     for (int i = 0; i < size; ++i)
67.         arr[i]->print();
68. }
69.
70. int main() {
71.     Circle ci1("first",2,cout),
72.          *pci2 = new Circle("second",3);
73.     Square sq1("first",4),
```

```
74.          *psq2 = new Square("second",5);
75.     Person ps1("Jim",1972),
76.          *pps2 = new Person("Tom",1978,cout);
77.
78.     Printable* arr[] = {&ci1, &ps1, pci2, pps2};
79.
80.     cout << "** Printing array of Printables" << endl;
81.     printArray(arr,4);
82.
83.     cout << "** Printing Squares" << endl;
84.     sq1.print();
85.     psq2->print();
86.
87.     delete pci2;
88.     delete psq2;
89.     delete pps2;
90. }
```

A free (global) function which prints information about objects of type **Printable** is defined on lines 65-68. Note that passing a table of pointers to this function (line 81), we could pass adresses of objects of type **Person** and **Circle**, but not **Square**, although this class also implements the method **print**. However, it does not inherit from **Printable**, so a pointer to its object cannot have static type **Printable***. The program prints

```
** Printing array of Printables
circle first with radius 2 and area 12.5664
Jim (1972)
circle second with radius 3 and area 28.2743
Tom (1978)
** Printing Squares
square first with side 4 and area 16
square second with side 5 and area 25
```

Abstract class **Printable** was characteristic in this example. It declares one pure virtual function which determines a well defined functionality, which is applicable to many, often completely unrelated, classes. Such simple (usually abstract) classes adding a single functionality to inheriting classes are called **mix-in classes** (they roughly correspond to interfaces known from Java). While complicated multibase inheritance is dangerous and difficult to handle, mix-in classes, as the one described above, are relatively safe and useful.

---

22. Exceptions

Like Java, Python and many other programming languages (but not C), C++ can handle **exceptions** , i.e., situations when an an error occures at run time and the program cannot continue (this can be calling a method for a **nullptr** pointer, trying to read from a nonexisting file etc.). Exception handling allows the programmer to prepare the program for such situations: the program can try to “repair” something and resume its work, or stop execution, but in a “civilized” way. The programmer can even decide that certain types of exceptions should be simply ignored. In more sophisticated applications, exception handling can constitute quite a large portion of the whole program — no bank would want its customers to see a screen with a message like *“segmentation fault, core dumped”* or *“memory cannot be read”*.

Exception handling in C++ is additionally complicated by the fact that the language was designed to be compatible with C, where such a mechanism does not exist.

Subsections

- 22.1 Throwing exceptions
- 22.2 Catching exceptions
- 22.3 Hierarchies of exceptions
- 22.4 Exceptions in constructors and destructors
- 22.5 Exception specifications
- 22.6 Rethrowing exceptions
- 22.7 Standard exceptions

22.1 Throwing exceptions

An exception can be **thrown** (or **raised**) in every place of the program. Sometimes we do not even know that a library function we are using can throw an exception. However, it is better to know it. The author of such function could not know what to do in a given situation, so he/she decided to throw an exception in the hope that someone using the function will know it and will react appropriately. Therefore, the mechanism of exceptions provides a way of communication between different parts of the program, possibly located in different modules and written by different programmers.

In a function that we write ourselves, we can specify what exceptions the function can throw at run time. Exceptions are described by objects of any type. We often design special classes, objects of which describe possible exceptions, but it is not obligatory — equally well these can be objects of a built-in type, like an integer or a string.

An exception can be **thrown** (or **raised**) by using the keyword **throw**:

```
throw excpt;
```

where ***wyjatek*** is na object of any type. This statement interrupts the execution of the function it appears in and the system looks for a procedure which declares (we will see how) that it can handle exceptions of this type. If such procedure is found, the exception is assumed to be handled and the body of the procedure is executed. After that the program, if not stopped by the procedure, will continue from the place *after* the procedure. There is no automatic return to the location where the exception was thrown.

If no such procedure has been found in the function where the exception was thrown, the flow of control leaves the function and the stack frame of its invocation is removed from the stack (the stack is “rewound”). This means, among other things, that all local variables created by the function are also removed. What is, however, very important, is the fact that for variables of object types destructors *will* be called. The program then continues to look for an exception handling procedure of an appropriate type in the calling function. If not found, the execution of this function is interrupted as well, its stack frame is rewound, and so on. Finally, there are two possibilities:

- an exception handling procedure is found. The exception is then considered handled, the flow of control enters the procedure and, if the program is not stopped by this procedure and no other exception is thrown it it, the program continues with the code immediately following the procedure (there is *no* return to the point where the exception was thrown);
- an exception handling procedure is not found and the program reached the **main** function. Then the program is terminated by calling the function **terminate**, which in turn calls **abort**. However, the programmer can set another function which will be called in such situations instead of **terminate**. In order to do this, he/she must call **set_terminate**, passing the pointer to a function (of type **void** and parameterless) which should be substituted for **terminate**. This function cannot return — the program has to be terminated anyway, but, perhaps, in a more civilized way (closing open

connections, removing temporary files etc.).

In the program below, we set another function (**termin**), which will substitute **terminate** (line 18):

P190: term.cpp Unhandled exceptions

```
1. #include <iostream>
2. #include <cmath>      // sqrt
3. #include <cstdlib>    // exit
4. #include <exception>
5. using namespace std;
6.
7. void termin() {
8.     cout << "termin: exit(7)" << endl;
9.     exit(7);
10. }
11.
12. double Sqrt(double x) {
13.     if (x < 0) throw "x < 0";
14.     return sqrt(x);
15. }
16.
17. int main() {
18.     set_terminate(&termin);
19.
20.     double z, x;
21.
22.     x =  16;
23.     z = Sqrt(x);
24.     cout << "Sqrt(" << x << ")=" << z << endl;
25.
26.     x = -16;
27.     z = Sqrt(x);
28.     cout << "Sqrt(" << x << ")=" << z << endl;
29. }
```

Function **Sqrt** throws an exception when the value of argument passed to it is negative. This exception is not handled anywhere in the program, so **termin** will be invoked. This function must not return; it terminates the program by calling **exit** (from the header `cstdlib`) with the return code equal to 7. This return code can be catched by the operating system and used in an useful purpose. In Linux, the shell (whether it is **bash** or **tcsh**) keeps the return code of the last program (command) in a predefined shell variable named '$?' (which is often used in shell scripts). We can check this by executing the following commands in a shell:

```
cpp> g++ -pedantic-errors -Wall -o term term.cpp
cpp> ./term
Sqrt(16)=4
termin: exit(7)
cpp> echo $?
7
```

Exceptions in the program above terminated the program. We can, however, handle them and decide ourselves whether the program should be stopped or, perhaps after appropriate actions, continued.

22.2 Catching exceptions

Exception handling procedure can be set by defining **catch** clauses. They have a form similar to that of a definition of a function with one parameter; the type of the parameter determine the type of exceptions that the procedure will handle. The procedure can handle only exceptions of this type (or its subtypes) and only those which were thrown in a compound statement preceded by the keyword **try** and located immediately before the **catch** phrase:

```
1.      try {
2.          // statements
3.      }
4.      catch(Type t) {
5.          // exception handlig
6.      }
7.      // ...
```

In this schematic example, an exception of type **Type** can be thrown by a statement inside **try** phrase. This, in particular, *can* be a function invocation — if an exception was thrown inside this function and has not been handled there, it will be treated as if it were thrown by the function invocation itself.

Suppose an exception of type **Type** (or its subtype) was indeed thrown inside the compound statement constituting the **try** clause. Then the flow of control passes immediately to the **catch** phrase, which itself is another compound statement. Inside the **catch** phrase, one has access to object of type **Type** which is the one used in the **throw** statement that has generated the exception. Passing this argument is similar to passing arguments to functions: if we receive it by value (as above), we actually get a copy of the original object. It is usually better to receive this object by reference, because in this way we can avoid making unnecessary copies of the objects and benefit from the mechanism of polymorphism.

It often happens that the information on the type of an exception is all what we need; the *value* of the object thrown can be ignored. In such cases, we do not even have to specify a name for this object — just declaring its type is sufficient.

When the flow of control enters a **catch** clause, the exception is considered to be handled. In particular, a new exception can be thrown here. If nothing like this will happen and there is no **return** statement, the program continues with the statement immediately following the **catch** clause (line 7 in our example).

Of course, if no exception has been thrown inside the **try** clause, the corresponding **catch** clause will be ignored.

Consider the following example:

P191: except.cpp Exceptions as objects of polymorphic types

```
1. #include <iostream>
2. #include <string>
3. #include <sstream>   // ostringstream
4. #include <cmath>     // sqrt, log, atan
5. using namespace std;
6.
7. struct Error {
8.     virtual string info() = 0;
9. };
10.
11. class Negative : public Error {
12.     double x;
13. public:
14.     Negative(double x) : x(x) { }
15.     string info() {
16.         ostringstream strum;
17.         strum << "Negative argument: x = " << x;
18.         return strum.str();
19.     }
20. };
21.
22. class OutOfRange : public Error {
23.     double x, min, max;
24. public:
25.     OutOfRange(double x, double mi, double ma)
26.         : x(x), min(mi), max(ma)
27.     { }
28.     string info() {
29.         ostringstream strum;
30.         strum << "Argument x = " << x << "\n    "
31.               << "    out of range [" << min << ","
32.               << max << "]";
33.         return strum.str();
34.     }
35. };
36.
37. double fun(double x) {
38.     if (x < 0 || x > 2) throw OutOfRange(x,0,3);
39.     return sqrt(x*(3-x));
40. }
41.
42. double logPI(double x) {
43.     static double LPI = log(4*atan(1.));
44.     if (x <= 0) throw Negative(x);
45.     return log(x)/LPI;
46. }
47.
48. int main() {
49.     double x = 4*atan(1.);
50.
51.     cout << "x = " << x << endl;
52.     try {
53.         double z1 = logPI(x);
54.         cout << "  z1 = " << z1 << endl;
55.         double z2 = fun(x);
56.         cout << "  z2 = " << z2 << endl;
57.     }
58.     catch(Error& blad) {
59.         cerr << "  Error: " << blad.info() << endl;
60.     }
61. }
```

We define an abstract class **Error** and two derived classes: **Negative** and **OutOfRange**. They both implement virtual method **info** inherited from the base class.

On lines 37-46, two global functions are defined; they both can throw an exception: function **fun** exception of type **OutOfRange** if the value of the argument is out range $[0, 3]$, and function **logPI** exception of type **Negative** if the argument is negative (the function calculates the logarithm of the argument to the base π). We then call both function in the **main** inside a **try** clause. A possible exception is caught by the **catch** clause defined on

lines 58-60. Note that it catches exceptions of type **Error** *by reference*, so it will be able to catch errors of all types derived from this abstract class and use polymorphism. This is illustrated by our example: classes are polymorphic and the method **info** is virtual. Therefore, invocation of **info** from line 59 will execute the method from the true (dynamic) class of the object passed as the argument,

```
x = 3.14159
  z1 = 1
  Error: Argument x = 3.14159
        out of range [0,3]
```

which in our case will be class **OutOfRange**, since the exception was thrown in function **fun**.

Prev: 22.1 Throwing exceptions **Up:** 22. Exceptions **Next:** 22.3 Hierarchies of exceptions

22.3 Hierarchies of exceptions

There can be more than one **catch** phrases following any **try** block. If this is the case, and if an exception has been thrown in the **try** clause, then the correct **catch** will be looked for sequentially, one by one, until the type of exception matches the type declared in the header of a **catch** block. When found, this **catch** block will be executed and the process of searching for handling procedure will stop — no other **catch** block will be checked, even if the type of exception they declare is a better match for the type of exeception to be handled. The rules of conversions, however, are slightly different than those which are used for function invocation. Upcasting (conversions to "wider" types) will be performed *only* for exceptions of object types, not for primitive types like **int** or **double**. For example, the following program

P192: hier.cpp Conversions of exceptions

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5.     try {
6.         throw 7;
7.     }
8.     catch(double) { cout << "double" << endl; }
9.     catch(int   ) { cout << "int   " << endl; }
10. }
```

will print ' `int`', although the first **catch** block (which declares **double** as the type of exception) would match the **int** exception after standard **int** → **double** conversion. After substituting '7.' (with a decimal dot) for '7' in line 6, the first **catch** would be selected, as it would then match exactly the type of the exception.

However, for object-type exceptions, the upcasting *will* take place:

P193: hierob.cpp Object-type exceptions

```
1. #include <iostream>
2. using namespace std;
3.
4. class A { };
5. class B : public A { };
6.
7. int main() {
8.     try {
9.         throw B();
10.     }
11.     catch(A) { cout << "A" << endl; }
12.     catch(B) { cout << "B" << endl; }
13. }
```

as we can see from the output

```
cpp> g++ -pedantic-errors -Wall -o hierob hierob.cpp
hierob.cpp: In function `int main()':
hierob.cpp:12: warning: exception of type `B' will be
                    caught by earlier handler for `A'
cpp> ./hierob
A
```

Th ecompiler issued a warning, because the second **catch** phrase in this example is not available at all. The same will hold for pointer-type exceptions; the program

P194: hier1.cpp Object-type exceptions

```
1. #include <iostream>
2. using namespace std;
3.
4. struct A {
5.     const char* info() { return "A*"; }
6. };
7.
8. struct B : A {
9.     const char* info() { return "B*"; }
10. };
11.
12. int main()
13. {
14.     try {
15.         throw new B;
16.     }
17.     catch(A* a) { cout << a->info() << endl; }
18.     catch(B* b) { cout << b->info() << endl; }
19. }
```

prints

```
cpp> g++ -pedantic-errors -Wall -o hier1 hier1.cpp
hier1.cpp: In function `int main()':
hier1.cpp:18: warning: exception of type `B*' will be
                    caught by earlier handler for `A*'
cpp> ./hier1
A*
```

Generally, if class **B** inherits from **A**, exception is of type **B** (or **B***), and **catch** clause declares type **A** (or **A***, or **A&**), then the exception will be caught by this clause if:

- **A** and **B** are the same type;
- **B** inherits *publicly* from **A** and *only* from **A** (**B** can inherit from other classes, but not publicly);
- both types are pointers, and the types of objects pointed to by them meet one of the requirements mentioned above;
- **catch** declares **A&**, and type **B** of the exception meets one of these requirements.

Od course, types of exceptions handled by consecutive **catch** are not required to have anything in common: in all cases the first which matches the type of exception will be selected.

One can also use (most often after other, more specific, phrases) a **catch** block with ellipsis (three dots) in place of type declaration:

```
catch(...) {
    // ...
}
```

what means *"catch every exception, no matter what its type is"*.

22.4 Exceptions in constructors and destructors

Exceptions can also be thrown in constructors and destructors. These are cases particularly hard to handle.

If an exception is thrown inside a constructor, the object *in statu nascendi* will not be completed and its destructor will not be invoked (what is logical, because formally the object has not even been created). However, it may happen that the object has object-type members which have already been fully constructed. These object will be destroyed and their destructors *will* be executed. However, the object can contain pointer-type members pointing to already allocated objects: their destructors will not be called. The same holds for other resources, like open files or data base connections — they are usually released and closed in the destructor, which will not be called. All this can lead to serious memory leakages and/or problems with open files, data base connections and the like. We can often avoid such problems by wrapping members referring to resources acquired so far by a constructor in member objects equipped with appropriate destructors: when the constructor fails, these destructors, being destructors of already complete member objects, *will* be executed. Let us look at an example

P195: resour.cpp Releasing resources after exceptions

```
#include <iostream>
#include <cstring>
#include <cstdio>  // FILE, fopen, fclose
using namespace std;

class A {
    struct nam {
        char* n;
        nam(const char* n)
            : n(strcpy(new char[strlen(n)+1],n))
        { }
        ~nam() {
            cerr << "dtor nam: " << n << endl;
            delete [] n;
        }
    };

    nam Name;
    FILE*    file;
public:
    A(const char* n, const char* p)
        : Name(n)
    {
        file = fopen(p,"r");
        // ...
//      throw 1;
        // ...
    }

    // other fields and methods

    ~A() {
        cerr << "dtor A" << endl;
```

```
34.         if (file) fclose(file);
35.     }
36. };
37.
38. int main() {
39.     try {
40.         A a("Carrington","afile.cpp");
41.     } catch(...) {
42.         cerr << "object instantiation failed\n";
43.     }
44. }
```

Class **A** has a member corresponding to a name represented as a C-string. The string is dynamically allocated , but the pointer to it is not a member of class directly: instead, there is an auxiliary structure **nam**, and it is an *object* of this class which is a member of **A**. The pointer to the C-string with the name is a member of this object. The structure **nam** has a destructor, which therefore will be called if constructor of **A** fails when the member object ***Name*** has already been cretaed. The destructor takes care of the memory allocated for the name.

Class **A** has additional pointer as the field: this is a pointer to an object of structure **FILE** (this is a standard type describing files in C).

What will happen if the line 26 (`throw 1`) is commented out? The object is created properly; no exception is thrown. When the flow of control leaves the **try** block, the object ***a*** of class **A**, being a local object in the block, is removed and its destructor is invoked which, in turn, closes the file. Then member objects are removed and their destructors called; in our case it is the member object ***Name*** which is destroyed — its destructor will deallocate memory occupied by the C-string with the name. The printout

```
dtor A
dtor nam: Carrington
```

shows that object ***a*** has been removed correctly.

Let us now activate the line 26, which causes an exception to be thrown during the process of constructing the object ***a***. At that time, member ***Name*** has already been created, so its destructor is called and the memory for the C-string is released. However, as the printout shows

```
dtor nam: Carrington
object instantiation failed
```

the file was not closed, because it was referred to by a “bare” pointer and the destructor of **A** had never been called.

In the example above, exception thrown in the constructor was not handled *in situ*, before leaving the constructor; it escapes from the constructor and is handled by a **catch** block in the calling function (**main**, in our case). This could be dangerous in the case of a *destructor*. The problem here is that destructors are called automatically in the process of rewinding the stack while handling an exception. An additional exception in a destructor would then lead to a situation when there are *two* exceptions being handled simultaneously. This is not possible in C++. If this happens, the program is unavoidably terminated by calling the function **terminate**. Therefore, if an exception can occur in a destructor, it should be handled by an appropriate **catch** block *inside* the destructor, thus preventing it from

escaping to the caller.

Prev: 22.3 Hierarchies of exceptions **Up:** 22. Exceptions **Next:** 22.5 Exception specifications

22.5 Exception specifications

Function declarations can specify whether and what types of unhandled exceptions can a given function throw during execution.

It is now recommended to use only two forms of such specification, as shown below:

```
int f(int i) noexcept;
int g(int i);
```

Here we declare that function **f** will never throw any exception; if it does, the program will immediately terminate by calling the **terminate** function. However, function **g** may throw any kind of exception. Exception specification belongs to the signature of a function, so it has to be repeated at every declaration and at the definition.

The **noexcept** specifier can take an argument of type (convertible to) **bool**; the above two declarations are equivalent to

```
int f(int i) noexcept(true);
int g(int i) noexcept(false);
```

The **noexcept** specifier can also be used as an operator For example

```
void m(int) noexcept(noexcept(f(i)));
void n(int) noexcept(noexcept(g(i)));
```

means that **m** has the same exception specification as **f** and **n** the same as **g** (of course, here `f(i)` and `g(i)` are *not* function invocations).

For virtual methods, all overriding versions in derived classes may only be less restrictive, but never more restrictive.

22.6 Rethrowing exceptions

Very often we catch an exception, undertake some appropriate actions (like releasing resources) but still we are not able to handle it completely, or we do not know if the program should continue or not under circumstances. It is then possible to **rethrow** *the same* exception, so it can be handled again by another **catch** clause. Such behavior can be achieved by using just the keyword **throw** with no arguments. Consider the following scheme:

```
void fun() {
    // opening sockets
    try {
        // using sockets
    }
    catch(...)  // we catch all exceptions
    {
        // closing sockets
        throw; // rethrowing the same exception
    }
    // closing sockets
}
```

The function opens an internet connection and then uses it; it may happen, of course, that something will go wrong with it (bad address, problems with the network etc.). Therefore, we catch all possible exceptions and close the connection inside the **catch** clause. But then the functin does not know what to do next, so it rethrows the same exception allowing it to escape to the calling function which then can catch it and decide whether the program should be continued or not. If no exception occurs, the program will execute statements after the **catch** clause and the connection will be closed anyway. Note that allowing the exception to escape from the function without this partial handling could lead to resource leakage, because the calling function would not be able to close the connection properly: it has been opened locally inside the function and is not accessible to the caller.

Prev: 22.5 Exception specifications **Up:** 22. Exceptions **Next:** 22.7 Standard exceptions

22.7 Standard exceptions

The Standard Library of C++ defines many types of exceptions which can be thrown by library functions (functions from C libraries do never throw any exceptions). All this types are derived from class **exception**. It declares method **what** which is supposed to return, as a C-string, a description of the problem which caused the exception. Derived classes should provide a sensible implementation of this method.

Let us mention the most important standard types of exceptions:

bad_alloc (from header `new`) — Thrown by **new** operator when memory allocation fails.

bad_cast (from header `typeinfo`) — Thrown by **dynamic_cast** operator when a requested conversions fails.

bad_typeid (from header `typeinfo`) — Thrown by **typeid** operator when its operand is an empty pointer.

bad_exception (from header `exception`) — It is thrown when an unhandled exception occurs which is not specified in the exception specification of a function, but only if itself *is* specified there. This mechanism can handle unexpected or unknown types of exceptions without terminating the program.

ios::failure (from header `ios`) — It is thrown when a stream has changed its state into an “unwanted” one. What “unwanted” means must be first specified by calling function **exceptions** for this stream object.

23. Modules and namespaces

All modern programming languages provide tools which can be used to modularize applications, i.e., to assemble programs from smaller, to some extend independent units. These units can often be developed and tested independently, perhaps by different programmers. Usually, there are also mechanism making it easier to avoid name clashes, occurring especially when an applications is developed by many programmers.

Subsections

- 23.1 Modules of a program
  - 23.1.1 Headers and implementation files
- 23.2 Namespaces

Subsections

- 23.1.1 Headers and implementation files

23.1 Modules of a program

One program is normally written physically in many files. As we know from section on preprocessor directives , a source file can include (also recursively) another source files by using the `#include` directive. Then, what the compiler will "see" is one combined file constituting one **module** of the program. Such a module is called a **translation unit**. The whole applications in turn can consist of many such translation units combined together into one executable by the linker.

One translation unit can (and should) correspond to a logical unit: a set of classes and functions operating on a well defined part of the whole problem.

Translation units can be compiled independently, perhaps on different machines. However, the C++ compiler checks types of all variables and correctness of all function invocations. Therefore, if a function is used in a translation unit, at least its *declaration* has to be available in this unit; the compiler does not have to know its *definition* to check if it is invoked correctly. The definition itself should be unique, located in exactly one translation unit (it does not apply to inlined functions, whose *definition* must be accessible in every translation unit where they are used). Of course, all declarations and the definition should be compatible.

After linking all the modules into one executable, global functions from all translation units will "see" each other without any particular action on our part. We can say that global functions live in one common scope of the program (we say that global functions are **exported**). Some C programmers use the keyword **extern** before declarations of functions whose definition is given in another translation unit: this practice is acceptable but not required in C++.

As an exception, global function declared with **static** specifier are *not* exported — they will be visible only in the module where they were defined.

Global *variables* are treated differently. They are *not* exported by default: a global variable declared in one translation unit is not visible in other modules. If we insist on making a global variable visible in more units, we have to define it exactly once in one of the modules and declare it as external in other modules (by adding specifier **extern** — see section on external variables. If the opposite is required, i.e., we want to define a global variable and be sure that it will *not* be visible in other units even if they declare the same name as the name of an external variable, then we can declare it as global **static** variable (see section on static variables).

23.1.1 Headers and implementation files

When writing a program, we should try to keep it as simple and easy to develop as possible. On the other hand, it should offer a clear interface for the users, so they can use it without being forced to delve into details. One of the mechanisms which can make it easier to achieve these two goals consists of dividing the code into different types of source files.

It is not unusual that the same set of functions, classes, templates, enumerations etc. is used in several modules. Their declarations have to be exactly the same and they have to be repeated in all these modules — this is, however, impractical, because it would be very hard to keep all these declarations synchronized in the process of developing and/or modifying the program. Instead, one can gather them in one file and include it (using preprocessor directive `#include`) into all translation units where they are needed (section on preprocessor directives). We do *not* include definitions in such files, only declarations (but we do include *definitions* of inlined functions). The files with declarations define the interface: one can inspect them to see what functions are available, how to invoke them etc. It is recommended to put there precise comments and “user's instructions”. Such files are called **header** files, or just headers. Traditionally, their names have extension `.h`. Usually they are not very large, as they contain nothing but declarations.

Definitions of entities declared in a header are placed in an **implementation** file; there is no generally accepted extension for names of such files, but often it is `.cxx` or `.C`, or simply `.cpp`. Implementation file includes (with directive `#include`) the corresponding header with declarations: in this way the compiler can check if declarations agree with definitions.

When the structure of a program gets more and more complicated and there are many `#include` directives (which, as we remember, can be nested), it can easily happen that the same header is included more than once — this can be harmless, but sometimes can cause problems. In order to avoid such situation, we can use the trick with the directive `#ifndef` described in section on preprocessor directives .

When headers and implementation source files pertaining to a functionality that we want to use are ready, we can include them into many different applications where a given functionality can be useful. It is then enough

- to `#include` appropriate headers into source files of a program;
- make sure that the compiled binaries of implementation files are accessible during linking the executable of the program (source files of implementation is then not needed).

Let us consider a simple example. There are many occasions when a procedure for sorting arrays of integers is required. Also, we decide that printing such arrays would be a useful thing to have. We then write a module consisting of two files. Header file `sortinte.h` declares functions for sorting and printing arrays of integers; it also contains comments which may be useful for users of the module:

P196: sortinte.h Header file

```
1. #ifndef _SORTINTE_H
2. #define _SORTINTE_H
3.
4. // comments...
```

```
5.
6. void sort(int[],int);
7. void writarr(const int[],int);
8. #endif
```

In a separate file, **sortinteImpl.cpp**, we implement both functions:

P197: sortinteImpl.cpp Implementation file

```
1. #include <iostream>
2. #include "sortinte.h"   // including header
3.                          // with declarations
4. using namespace std;
5.
6. // implementation of function sort
7. void sort(int a[], int size) {
8.     int i, indmin = 0;
9.     for (i = 1; i < size; ++i)
10.         if (a[i] < a[indmin]) indmin = i;
11.     if (indmin != 0) {
12.         int p = a[0];
13.         a[0] = a[indmin];
14.         a[indmin] = p;
15.     }
16.
17.     for (i = 2; i < size; ++i) {
18.         int j = i, v = a[i];
19.         while (v < a[j-1]) {
20.             a[j] = a[j-1];
21.             j--;
22.         }
23.         if (i != j ) a[j] = v;
24.     }
25. }
26.
27. // implementation of function writarr
28. void writarr(const int t[], int size) {
29.     cout << "[ ";
30.     for (int i = 0; i < size; ++i)
31.         cout << t[i] << " ";
32.     cout << "]" << endl;
33. }
```

At the beginning of **sortinteImpl.cpp** we include the header. Strictly speaking, it is not needed here, because definitions are also declarations, but we do it so the compiler can check if definitions agree with the interface determined by declarations known to the user, who only has access to the header file. Now we can compile the implementation file with option '`-c`' to produce an object file containing the compiled source but not yet linked to an executable. Such files usually have extension **.o**.

```
cpp> ls                                                  (1)
sortinteImpl.cpp  sortinte.h                             (2)
cpp> g++ -pedantic-errors -Wall -c sortinteImpl.cpp      (3)
cpp> ls                                                  (4)
sortinteImpl.cpp  sortinte.h  sortinteImpl.o             (5)
```

We list files whose names start with '`sortinte`' on line 1. Then on line 3 we compile implementation file **sortinteImpl.cpp** with option'`-c`'. As we can see on line 5, file **sortinteImpl.o** has been produced. Of course, for the compilation to succeed, header **sortinte.h** had to exist, as it will be included to **sortinteImpl.cpp** by preprocessor to form one translation unit.

Suppose now that we are writing an aplication **sortinteApp.cpp** which uses the sorting module. We include *only* the header. It contains all declarations needed by the compiler to check if invocations of functions **sort** and **writarr** are correct.

P198: sortinteApp.cpp Application

```
1. #include "sortinte.h"  // only header!
2. #include <iostream>
3. using namespace std;
4.
5. int main() {
6.     int arr[] = {9,7,2,6,4,5,6,2,7,9,2,9,5,2},
7.         size  = sizeof(arr)/sizeof(arr[0]);
8.
9.     cout << "Original array: ";
10.    writarr(arr, size);
11.
12.    sort(arr, size);
13.
14.    cout << "  Sorted array: ";
15.    writarr(arr, size);
16. }
```

Now we compile our application `sortinteApp.cpp`, perhaps after a year since creation of the object file `sortinteImpl.o`. The linker will need the object `sortinteImpl.o`, but not the source file `sortinteImpl.cpp`. We get executable `sortinteApp` which we can run to obtain the results of our application:

```
cpp> ls
sortinteApp.cpp  sortinte.h  sortinteImpl.o
cpp> g++ -o sortinteApp sortinteApp.cpp sortinteImpl.o
cpp> ./sortinteApp
Original array: [ 9 7 2 6 4 5 6 2 7 9 2 9 5 2 ]
  Sorted array: [ 2 2 2 2 4 5 5 6 6 7 7 9 9 9 ]
```

Suppose that after another year we realized that the insertion sort algorithm used in out sorting function is ineffective and should be replaced by the heap-sort algorithm. What should we do? We have to change implementation in `sortinteImpl.cpp`, compile it and send new object file `sortinteImpl.o` to the user. The user can link the new version with his/her application without modifying a line in the code of the application, because the interface described by the header has not been changed.

Compiled implementation code is usually distributed in the form of shared libraries; in Linux such libraries have names with extension `.so` (shared object) while in Windows it is `.dll` (dynamic-link library). In this case even the stage of relinking is not necessary — just substituting new libraries for their old version will suffice.

This is the way in which the C++ environment is built. The programmer includes header files to his/her applications. These headers are usually located in directory `include` of the C++ instalation and are just normal text files which can (and should) be inspected. Implementation is provided in the form of binary libraries, usually in catalog `lib`.

Subsections

- 23.1.1 Headers and implementation files

23.1 Modules of a program

One program is normally written physically in many files. As we know from section on preprocessor directives , a source file can include (also recursively) another source files by using the `#include` directive. Then, what the compiler will "see" is one combined file constituting one **module** of the program. Such a module is called a **translation unit**. The whole applications in turn can consist of many such translation units combined together into one executable by the linker.

One translation unit can (and should) correspond to a logical unit: a set of classes and functions operating on a well defined part of the whole problem.

Translation units can be compiled independently, perhaps on different machines. However, the C++ compiler checks types of all variables and correctness of all function invocations. Therefore, if a function is used in a translation unit, at least its *declaration* has to be available in this unit; the compiler does not have to know its *definition* to check if it is invoked correctly. The definition itself should be unique, located in exactly one translation unit (it does not apply to inlined functions, whose *definition* must be accessible in every translation unit where they are used). Of course, all declarations and the definition should be compatible.

After linking all the modules into one executable, global functions from all translation units will "see" each other without any particular action on our part. We can say that global functions live in one common scope of the program (we say that global functions are **exported**). Some C programmers use the keyword **extern** before declarations of functions whose definition is given in another translation unit: this practice is acceptable but not required in C++.

As an exception, global function declared with **static** specifier are *not* exported — they will be visible only in the module where they were defined.

Global *variables* are treated differently. They are *not* exported by default: a global variable declared in one translation unit is not visible in other modules. If we insist on making a global variable visible in more units, we have to define it exactly once in one of the modules and declare it as external in other modules (by adding specifier **extern** — see section on external variables. If the opposite is required, i.e., we want to define a global variable and be sure that it will *not* be visible in other units even if they declare the same name as the name of an external variable, then we can declare it as global **static** variable (see section on static variables).

23.1.1 Headers and implementation files

When writing a program, we should try to keep it as simple and easy to develop as possible. On the other hand, it should offer a clear interface for the users, so they can use it without being forced to delve into details. One of the mechanisms which can make it easier to achieve these two goals consists of dividing the code into different types of source files.

It is not unusual that the same set of functions, classes, templates, enumerations etc. is used in several modules. Their declarations have to be exactly the same and they have to be repeated in all these modules — this is, however, impractical, because it would be very hard to keep all these declarations synchronized in the process of developing and/or modifying the program. Instead, one can gather them in one file and include it (using preprocessor directive `#include`) into all translation units where they are needed (section on preprocessor directives). We do *not* include definitions in such files, only declarations (but we do include *definitions* of inlined functions). The files with declarations define the interface: one can inspect them to see what functions are available, how to invoke them etc. It is recommended to put there precise comments and “user's instructions”. Such files are called **header** files, or just headers. Traditionally, their names have extension `.h`. Usually they are not very large, as they contain nothing but declarations.

Definitions of entities declared in a header are placed in an **implementation** file; there is no generally accepted extension for names of such files, but often it is `.cxx` or `.C`, or simply `.cpp`. Implementation file includes (with directive `#include`) the corresponding header with declarations: in this way the compiler can check if declarations agree with definitions.

When the structure of a program gets more and more complicated and there are many `#include` directives (which, as we remember, can be nested), it can easily happen that the same header is included more than once — this can be harmless, but sometimes can cause problems. In order to avoid such situation, we can use the trick with the directive `#ifndef` described in section on preprocessor directives .

When headers and implementation source files pertaining to a functionality that we want to use are ready, we can include them into many different applications where a given functionality can be useful. It is then enough

- to `#include` appropriate headers into source files of a program;
- make sure that the compiled binaries of implementation files are accessible during linking the executable of the program (source files of implementation is then not needed).

Let us consider a simple example. There are many occasions when a procedure for sorting arrays of integers is required. Also, we decide that printing such arrays would be a useful thing to have. We then write a module consisting of two files. Header file `sortinte.h` declares functions for sorting and printing arrays of integers; it also contains comments which may be useful for users of the module:

P196: sortinte.h Header file

```
1. #ifndef _SORTINTE_H
2. #define _SORTINTE_H
3.
4. // comments...
```

```
5.
6. void sort(int[],int);
7. void writarr(const int[],int);
8. #endif
```

In a separate file, **sortinteImpl.cpp**, we implement both functions:

P197: sortinteImpl.cpp Implementation file

```
1. #include <iostream>
2. #include "sortinte.h"   // including header
3.                          // with declarations
4. using namespace std;
5.
6. // implementation of function sort
7. void sort(int a[], int size) {
8.     int i, indmin = 0;
9.     for (i = 1; i < size; ++i)
10.         if (a[i] < a[indmin]) indmin = i;
11.     if (indmin != 0) {
12.         int p = a[0];
13.         a[0] = a[indmin];
14.         a[indmin] = p;
15.     }
16.
17.     for (i = 2; i < size; ++i) {
18.         int j = i, v = a[i];
19.         while (v < a[j-1]) {
20.             a[j] = a[j-1];
21.             j--;
22.         }
23.         if (i != j ) a[j] = v;
24.     }
25. }
26.
27. // implementation of function writarr
28. void writarr(const int t[], int size) {
29.     cout << "[ ";
30.     for (int i = 0; i < size; ++i)
31.         cout << t[i] << " ";
32.     cout << "]" << endl;
33. }
```

At the beginning of **sortinteImpl.cpp** we include the header. Strictly speaking, it is not needed here, because definitions are also declarations, but we do it so the compiler can check if definitions agree with the interface determined by declarations known to the user, who only has access to the header file. Now we can compile the implementation file with option '`-c`' to produce an object file containing the compiled source but not yet linked to an executable. Such files usually have extension **.o**.

```
cpp> ls                                                  (1)
sortinteImpl.cpp  sortinte.h                             (2)
cpp> g++ -pedantic-errors -Wall -c sortinteImpl.cpp      (3)
cpp> ls                                                  (4)
sortinteImpl.cpp  sortinte.h  sortinteImpl.o             (5)
```

We list files whose names start with '`sortinte`' on line 1. Then on line 3 we compile implementation file **sortinteImpl.cpp** with option'`-c`'. As we can see on line 5, file **sortinteImpl.o** has been produced. Of course, for the compilation to succeed, header **sortinte.h** had to exist, as it will be included to **sortinteImpl.cpp** by preprocessor to form one translation unit.

Suppose now that we are writing an aplication **sortinteApp.cpp** which uses the sorting module. We include *only* the header. It contains all declarations needed by the compiler to check if invocations of functions **sort** and **writarr** are correct.

P198: sortinteApp.cpp Application

```
1. #include "sortinte.h"  // only header!
2. #include <iostream>
3. using namespace std;
4.
5. int main() {
6.     int arr[] = {9,7,2,6,4,5,6,2,7,9,2,9,5,2},
7.         size  = sizeof(arr)/sizeof(arr[0]);
8.
9.     cout << "Original array: ";
10.    writarr(arr, size);
11.
12.    sort(arr, size);
13.
14.    cout << "  Sorted array: ";
15.    writarr(arr, size);
16. }
```

Now we compile our application `sortinteApp.cpp`, perhaps after a year since creation of the object file `sortinteImpl.o`. The linker will need the object `sortinteImpl.o`, but not the source file `sortinteImpl.cpp`. We get executable `sortinteApp` which we can run to obtain the results of our application:

```
cpp> ls
sortinteApp.cpp  sortinte.h  sortinteImpl.o
cpp> g++ -o sortinteApp sortinteApp.cpp sortinteImpl.o
cpp> ./sortinteApp
Original array: [ 9 7 2 6 4 5 6 2 7 9 2 9 5 2 ]
  Sorted array: [ 2 2 2 2 4 5 5 6 6 7 7 9 9 9 ]
```

Suppose that after another year we realized that the insertion sort algorithm used in out sorting function is ineffective and should be replaced by the heap-sort algorithm. What should we do? We have to change implementation in `sortinteImpl.cpp`, compile it and send new object file `sortinteImpl.o` to the user. The user can link the new version with his/her application without modifying a line in the code of the application, because the interface described by the header has not been changed.

Compiled implementation code is usually distributed in the form of shared libraries; in Linux such libraries have names with extension `.so` (shared object) while in Windows it is `.dll` (dynamic-link library). In this case even the stage of relinking is not necessary — just substituting new libraries for their old version will suffice.

This is the way in which the C++ environment is built. The programmer includes header files to his/her applications. These headers are usually located in directory `include` of the C++ instalation and are just normal text files which can (and should) be inspected. Implementation is provided in the form of binary libraries, usually in catalog `lib`.

23.2 Namespaces

Namespaces are a relatively new mechanism added to C++ standard. It provides a way to gather logically connected set of names (of classes, functions, enumerations, etc.) into a group of names separated from other names appearing in a program.

Suppose that a programmer has written a set of classes and functions responsible for graphical user's interface (GUI) of an application. Someone else created another set of classes, responsible for the so called “business logic”. If the application is large, it is not unlikely that both programmer used the same names for some of their functions or classes. This can lead to name clashes which are sometimes hard to detect and correct. However, the problem can easily be avoided by using the mechanism of **namespaces**. The two sets of names can be declared in separate namespaces. This can be achieved like this

```
namespace GUI {
    class Menu { /* ... */ };
    void show(Menu&);
    double calculate(double) {
        // ...
    }
    // ...
}

namespace Business {
    class Tax { /* ... */ };
    double calculate(double&);
    // ...
}
```

Note that name **calculate** appears in both namespaces. In one of them a function of this name has been defined, in the other only declared. This does not lead to any conflict: the functions belong to different namespaces and they do not have to have anything in common, whether their signatures are the same or not. When a name ***name*** is declared in a namespace ***namspac***, then its full name becomes ***namspac::name***, i.e., it is qualified with the name of the namespace it belongs to. For example, in order to define function **calculate** from namespace **Business**, we have to use its full name (unless the definition itself is put into the **namespace** block):

```
double Business::calculate(double& z) {
    // ...
}
```

and to define, the destructor of class **Menu** from namespace **GUI**:

```
GUI::Menu::~Menu() { /* ... */ }
```

One can add names to an already existing namespace. For example, the same destructor could have been defined like this:

```
namespace GUI {
    Menu::~Menu() { /* ... */ }
    const double PI = 3.14;
}
```

Note that the name **Menu** is not qualified with the name of the namespace, because it is defined inside the namespace **GUI**. As we can see, it is also possible to add new names to a namespace — here we have added a name of a floating-point constant ***PI***.

Functions **calculate** from both namespaces can now be used safely without threat of any conflict, as their full names are different:

```
double x, y, v, w;
// ...
x = GUI::calculate(v);
y = Business::calculate(w);
```

In this way names from different namespaces can be completely separated. If, for example, different parts of an applications or library are developed by different programmers, they can put their classes or functions in different namespaces. These classes or functions can be then used in the application, but must be intentionally selected by qualifying their names with the name of a namespace they are declared in.

On the other hand, there are situations when we are sure that there is no name conflicts between names from a namespace and names visible in the current scope. Qualifying all names from the namespace by its name can be quite cumbersome. One can avoid this by the so called **using declaration**. It is expressed by keyword **using**, which includes given name into the current scope so it can be referred to without qualification with the name of the namespace it belongs to. For example, after

```
using GUI::calculate;
```

the name **calculate** will refer to the function of this name from namespace **GUI**, so it will be an alias of **GUI::calculate**. Of course, function **calculate** from namespace **Business** is still accessible, but its name will have to be explicitly qualified by the name of namespace it comes from: **Business::calculate(x)**.

It is also possible, although not recommended, to "open" a namespace completely, importing *all* names it contains to the current scope. One can do it by *using directive* expressed by keywords **using namespace**. For example, after

```
using namespace GUI;
```

all names from namespace **GUI** can be used in the current scope without qualification. However, opening in this way large namespaces, especially if they are not known to us in details, can lead to name conflicts that the mechanism of namespaces was supposed to eliminate.

Let us consider a simple example:

P199: nmspc.cpp Namespaces

```
1.  #include <iostream>
2.
3.  namespace A {
4.      const int two = 2;
5.      const int six = 6;
6.      void write() { std::cout << "ns-A" << " "; }
7.  }
8.
9.  namespace B {
10.     void write() { std::cout << "ns-B" << " "; }
11. }
12.
```

```
13. namespace C {
14.     const int two = 22;
15.     const int six = 66;
16. }
17.
18. int main() {
19.     using A::write;
20.     using namespace C;
21.
22.     write();
23.     B::write();
24.     std::cout << six << std::endl;
25. }
```

We define three namespaces here. Line 19 includes the name **write** from namespace **A** to the scope of function **main**, so unqualified name **write** will refer to **A::write** (as on line 22). On line 20 we open the whole namespace **C**. Therefore, all names declared in this namespace can be used in **main** without qualifications. In particular, name ***six*** used on line 24 refers to a constant of this name defined in namespace **C**, and not to identically named constant from namespace **A**. Of course, function **write** from namespace **B** is accessible, but must be referred to by its qualified name (line 23). The program prints '`ns-A ns-B 66`'.

Note that names ***cout*** and ***endl*** have to be qualified in this program (lines 6, 10 and 24). Header **`iostream`** declares them in namespace **std**. In our previous examples, we always used

```
using namespace std;
using namespace std;
```

at the very beginning, so we could refer to names from **std** without qualifications — here qualification is needed. Generally, the standard C++ library declares all names in namespace **std** (except for **operator new**, **operator delete** and preprocessor macros).

Namespaces are relatively new in C++. In C header files are also used, but all names declared in them are added to the current scope and do not need any qualifications. To make the C++ language compatible with C, where there are no namespaces at all, the following mechanism is used. We can include traditional C header files by writing their names as they are used in this language. All names declared there are then included in the current scope as if they were imported by using directive. However, the same headers can be included by name with added letter 'c' at the beginning and without extension **`.h`**. Then the names declared in this header are added to namespace **std** and should therefore be qualified in the program (or imported by using declaration or using directive). For example, preprocessor directive

```
#include <string.h>
```

is equivalent to

```
#include <cstring>
```

but in the first case all names declared in the header are accessible without qualification, while the second form puts all these names into namespace **std**. The same holds for the following 18 standard header files of the C programming language:

C	C ++	C	C ++	C	C ++
assert.h	cassert	ctype.h	cctype	errno.h	cerrno
float.h	cfloat	iso646.h	ciso646	limits.h	climits
locale.h	clocale	math.h	cmath	setjmp.h	csetjmp
signal.h	csignal	stdarg.h	cstdarg	stddef.h	cstddef

stdio.h	cstdio	stdlib.h	cstdlib	string.h	cstring
time.h	ctime	wchar.h	cwchar	cwtype.h	cwctype

Table 23.1: Header files form C in C++

Note that there is no header `iostream.h` — it is defined by many implementations, but does not belong to the standard and should not be used (always use `iostream` instead).

The remaining 32 standard headers do not have their counterparts in C and are characteristic for C++ only:

algorithm	iomanip	list	ostream	streambuf
bitset	ios	locale	queue	string
complex	iosfwd	map	set	typeinfo
deque	iostream	memory	sstream	utility
exception	istream	new	stack	valarray
fstream	iterator	numeric	stdexcept	vector
functional	limits			

Table 23.2: Header files of C++ only

Let us conclude this chapter with another version of example `stacks.cpp`. Now we put names from the module describing stacks into namespace **mySTACKS**, to avoid clashes between names from our module and names from other namespaces. File `mySTACK.h` is a header declaring abstract class **STACK**:

P200: mySTACK.h Header of abstract class

```
1.  #ifndef mySTACK_H
2.  #define mySTACK_H
3.
4.  namespace mySTACKS {
5.      class STACK
6.      {
7.      public:
8.          virtual void push(int) = 0;
9.          virtual int pop()      = 0;
10.         virtual bool empty()   = 0;
11.         static STACK* getInstance(int);
12.         virtual ~STACK() { }
13.     };
14. }
15. #endif
```

File `mySTACKS.h` contains the header for implementation — it includes `mySTACK.h` and adds to namespace **mySTACKS** concrete classes which will implement the abstract class **STACK**:

P201: mySTACKS.h Header for implementing classes

```
1.  #ifndef mySTACKS_H
2.  #define mySTACKS_H
3.
4.  #include "mySTACK.h"
5.
6.  namespace mySTACKS {
7.
8.      class ListStack: public STACK {
9.          struct Node {
10.             int   data;
11.             Node* next;
12.             Node(int data, Node* next);
13.         };
```

```
        Node* head;
        ListStack();
    public:
        friend STACK* STACK::getInstance(int);
        int pop();
        void push(int data);
        bool empty();
        ~ListStack();
    };

    class ArrayStack : public STACK {
        int  top;
        int* arr;
        ArrayStack();
    public:
        friend STACK* STACK::getInstance(int);
        void push(int data);
        int pop();
        bool empty();
        ~ArrayStack();
    };
}
#endif
```

File with implementation, `mySTACKSImpl.cpp`, includes the header `mySTACKS.h` (which in turn includes `mySTACK.h`) and provides implementation for concrete classes

P202: mySTACKSImpl.cpp Implementation

```
#include <iostream>
#include "mySTACKS.h"
using namespace mySTACKS;

// ListStack

ListStack::Node::Node(int data, Node* next)
    : data(data), next(next)
{ }

ListStack::ListStack() {
    head = nullptr;
    std::cerr << "Creating ListStack" << std::endl;
}

int ListStack::pop() {
    int   data = head->data;
    Node* temp = head->next;
    delete head;
    head = temp;
    return data;
}

void ListStack::push(int data) {
    head = new Node(data, head);
}

bool ListStack::empty() {
    return head == nullptr;
}

ListStack::~ListStack() {
    std::cerr << "Deleting ListStack" << std::endl;
    while (head) {
        Node* node = head;
        head = head->next;
        std::cerr << " node " << node->data << std::endl;
        delete node;
    }
}

// ArrayStack

ArrayStack::ArrayStack() {
    top = 0;
```

```
    arr = new int[100];
    std::cerr << "Creating ArrayStack" << std::endl;
}

void ArrayStack::push(int data) {
    arr[top++] = data;
}

int ArrayStack::pop() {
    return arr[--top];
}

bool ArrayStack::empty() {
    return top == 0;
}

ArrayStack::~ArrayStack() {
    std::cerr << "Deleting ArrayStack with " << top
              << " elements remaining" << std::endl;
    delete [] arr;
}

// STACK

STACK* STACK::getInstance(int size) {
    if (size > 100)
        return new ListStack();
    else
        return new ArrayStack();
}
```

Now we can compile this file to get binary file **mySTACKSImpl.o**

```
cpp> g++ -pedantic-errors -Wall -c mySTACKSImpl.cpp
```

Only file **mySTACKSImpl.o** and the header **mySTACK.h** is needed to compile an application which uses our medule:

P203: stacksApp.cpp Application

```
#include <iostream>
#include "mySTACK.h"

int main() {

    mySTACKS::STACK* stack;

    stack = mySTACKS::STACK::getInstance(120);
    stack->push(1);
    stack->push(2);
    stack->push(3);
    stack->push(4);
    std::cout << stack->pop() << " ";
    std::cout << stack->pop() << std::endl;
    delete stack;

    stack = mySTACKS::STACK::getInstance(50);
    stack->push(1);
    stack->push(2);
    stack->push(3);
    stack->push(4);
    std::cout << stack->pop() << " ";
    std::cout << stack->pop() << std::endl;
    delete stack;
}
```

Compilation:

```
cpp> g++ -pedantic-errors -Wall -o stacksApp \
               stacksApp.cpp mySTACKSImpl.o
```

creates executable **stacksApp**, which can now be run to produce the results:

```
cpp> ./stacksApp
Creating ListStack
4 3
Deleting ListStack
 node 2
 node 1
Creating ArrayStack
4 3
Deleting ArrayStack with 2 elements remaining
```

Prev: 23.1 Modules of a program **Up:** 23. Modules and namespaces **Next:** 24. The Standard Library (STL)

24. The Standard Library (STL)

The C++ platform provides, besides the language itself, a library of many ready-to-use functions, classes, templates etc. The library is diveded in many parts that we can get access to by `#include`ing headers describing a functionality we need (see section on header files). Implementation of functions and classes declared in header files (in the binary, compiled form), is distributed by providers of C++ compilers; it can also be obtained from independent providers, both commercially or as open-source projects.

The Standard Library is not so rich as libraries provided with other languages, because it does not cover such topics like graphics, networking, data base connectivity etc. It focuses mainly on collections (containers) and algorithms operating on them. For historical reasons, this part of the library is often called STL — Standard Template Library.

The standard library provides mainly *templates* of functions and classes; the user can concretize them using build-in as well as object types (in particular, user defined). Such types, to be used with templates from the library, very often have to conform to some requirements: depending on how we want to use them it may be necessary to provide default and copy/move constructors, define (overload) '<' and '=' operators etc. The reasons for that will soon become clear.

Of course, the library does not contain templates *only*; we already know its parts which are not implemented as templates, like facilities provided in **`cstdlib`**, **`cstring`**, **`cmath`** etc. (these particular modules are inherited from C).

This chapter can be treated as a brief introduction only; more details can be found in books cited in bibliography .

Subsections

- 24.1 Collections and iterators
  - 24.1.1 Vectors
  - 24.1.2 Iterators
  - 24.1.3 Operations on collections
- 24.2 Algorithms and function objects
  - 24.2.1 Algorithms
  - 24.2.2 Function objects
- 24.3 Examples
- 24.4 List of algorithm

T.R. Werner, September 27, 2023; 22:32

Subsections

- 24.1.1 Vectors
- 24.1.2 Iterators
- 24.1.3 Operations on collections

24.1 Collections and iterators

The main part of the library is related to **collections**. These are data structures consisting of elements of the same type and equipped with a set of functions operating on this data (adding and removing elements from a collection, searching, sorting, modifying, copying etc.). Structures like lists, vectors, deques, heaps, stacks, maps are directly implemented; by employing them one can, relatively easy, implement other structures, like various kinds of trees or graphs.

24.1.1 Vectors

The simplest collection is the **vector**, accessible after including the header `vector`. A vector can be thought of as an array with its size unspecified in advance — it grows automatically as we add new elements to it. Elements are located in a well defined order and can be accessed randomly by specifying their index. One can add and remove elements on any position, but adding and removing at the last position is particularly effective (in constant time, not depending on the vector's size). The programmer does not have to worry about memory management — storage is allocated `all by itself' when a vector grows. As **vector** is actually a template, not a class, in order to create a vector we have to concretize this template for a given type — this can also be a built-in type, like **int** or **double**. The default constructor creates an empty vector (there are other constructors). We can add elements using the function **push_back**, which adds a *copy* of an object at the end of the vector (temporary objects may be moved instead, what can be much more effective). A copy, so one has to ensure that objects that are to be added can be correctly copied (or moved). There is also the **emplace_back** method which takes arguments for a constructor and creates the object directly inside the vector.

Elements can be accessed in various ways. The simplest way is to use indices — exactly as for arrays (indexing starts, of course, form zero). This is an efficient way, but sometimes dangerous, as it is *not* checked if the index is valid (non-negative and not equal or larger than the current size). If such a check is important, we can use method **at** supplying an index. Then, if the index is not valid, exception **out_of_range** (from header `stdexcept`), will be thrown and we can handle it appropriately:

P204: ate.cpp Vectors

```
1.  #include <vector>
2.  #include <stdexcept>
3.  #include <iostream>
4.  #include <string>
5.  using namespace std;
6.
7.  int main() {
8.      vector<string> vs;                                  ❶
9.
10.     vs.push_back("Mary");
11.     vs.push_back("Lucy");
12.     vs.push_back("Ella");
13.     vs.push_back("Jill");
14.
15.     try {
16.         for ( int i = 0; i < 5 /* ERROR */; i++ )      ❷
17.             cout << vs.at(i) << " " ;
18.     } catch(out_of_range) {
19.         cout << "\n*** Bad index! *** "
20.              << " vector has only " << vs.size()
21.              << " elements!" << endl;
22.     }
23.     cout << endl;
24.
25.     cout << "First element: " << vs.front() << endl;  ❸
26.     cout << "Last  element: " << vs.back()  << endl;  ❹
27.
28.     vs.pop_back();                                      ❺
29.
30.     cout << "After pop_back: ";
31.     int size = (int)vs.size();                          ❻
32.     for ( int i = 0; i < size; i++)
33.         cout << vs[i] << " " ;
34.     cout << endl;
35. }
```

In the following program, there are five iterations of the loop (❷) although there are only four elements in the vector. Therefore, the last iteration throws an exception (but handled here):

```
Mary Lucy Ella Jill
*** Bad index! ***  vector has only 4 elements!

First element: Mary
Last  element: Jill
After pop_back: Mary Lucy Ella
```

As one can see (❶), object ***vs*** is an object of class **vector<string>** which is a concretization of template **vector** for type **string**.

The collection consists of elements of type **string**. Method **size**, used on line ❻, returns, as one can guess, the size of a collection (not only for vectors). The returned type is an unsigned integer type whose alias is the name **vector::size_type** — that is why we used a cast. We could have used a plain **int**, but that would expose us to an unfriendly compiler's warning. Method **pop_back** (❺) removes the last element but does not return it. Methods **front** and **back** (❸ i ❹) return, by reference, the first and the last elements of the vector without removing them.

24.1.2 Iterators

In order to use collections one has to employ **iterators**. They provide means to access elements of collections and to iterate over them in a type independent way. Semantics of iterators resembles that of pointers, although they do not have to be implemented as

pointers. Their true type is normally of no interest to us — its name (alias) is accessible under the name **coll<Type>::iterator** where ***coll*** is the name of collection's template and **Type** is a type which was used to concretize this template (for example **vector<int> ::iterator** will be the alias of type of iterator related to vectors of **int** s). A collection can be, or we want treat is as being, a collection of constant (immutable) elements. In that case, one has to use separate type of iterator, **const_iterator** (for example, **vector<int> ::const_iterator**).

Iterators have semantics of pointers to elements of an array, i.e., if ***it*** is an iterator pointing to an element of a collection, then `*it` is a reference to this element (consequently, an l-value), and `it->memb` is the name of member ***memb*** of this object (if such a member exists). Similarly, after '`++it`', the iterator ***it*** will point to the next element of the collection.

There are two particularly important methods in all collection classes; they both return iterators connected to the collection that the methods were are called for. Method **begin** returns an iterator pointing to the first element of the collection. Method **end** returns an iterator pointing to a *nonexistent* element which is (or rather would be) the `first after the last' element. [There are also global functions, defined in the `iterator` header, **begin** and **end** which take a collection and return the same iterators.] If we iterate over a collection using an iterator, then it reaches the value returned by **end** when the whole collection has been exhausted and there are no other elements in it. That means that instead of checking whether the index of a loop is still smaller that the size of a collection, as we normally do in loops over arrays, we have to check if the iterator is still *different* than the value returned by **end**:

P205: iterat.cpp Iterators

```
1.  #include <vector>
2.  #include <iostream>
3.  #include <string>
4.  using namespace std;
5.
6.  int main() {
7.      vector<string> vs{"Mary", "Lucy", "Ella","Jill"};
8.
9.      for ( vector<string>::iterator ite = vs.begin();
10.                                     ite!= vs.end(); ++ite)
11.         cout << *ite << " ";
12.     cout << endl;
13.
14.     // or
15.
16.     vector<string>::iterator it, kon = vs.end();
17.
18.     for ( it = vs.begin(); it != kon; ++it )
19.         cout << *it << " ";
20.     cout << endl;
21.
22.     // or
23.
24.     using SIT=vector<string>::iterator;        ❶
25.
26.     SIT iter, fin = vs.end();
27.
28.     for ( iter = vs.begin(); iter != fin; ++iter )
29.         cout << *iter << " ";
30.     cout << endl;
31.
32.     // or the best way to do it
33.
34.     for ( auto i = vs.begin(); i != vs.end(); ++i )
35.         cout << *i << " ";
```

```
36.     cout << endl;
37. }
```

The program above illustrates iteration over a collection. If there are several iterator variables of the same type in a program, one can make life easier by defining, with **typedef** or **using**, a short alias for this type, as we did in line ❶. As we have already said, `++it` advances an iterator by one position, while `*it` is a reference to the element currently pointed to by this iterator. Of course, the most convenient form of such a loop is the last one, with **auto**.

If elements of a collection are unmodifiable (or we want to treat them as such), we have to use a special type of iterator, **const_iterator**. This is often used when we pass a collection to a function which, by assumption, should not change its elements:

P206: constit.cpp Iterators to unmodifiable elements

```
1. #include <vector>
2. #include <iostream>
3. using namespace std;
4.
5. struct Person {
6.     char name[20];
7.     int  year;
8.     void print() const {
9.         cout << name << "-" << year << endl;
10.    }
11. } john = {"John",25}, mary = {"Mary",18}, sue = {"Sue",9};
12.
13.
14. void printPerson(const vector<const Person*>& list) {
15.     for (auto it = list.cbegin(); it != list.cend(); it++)
16.         (*it)->print();
17. }
18.
19. int main() {
20.     vector<const Person*> list;
21.
22.     list.push_back(&john);
23.     list.push_back(&mary);
24.     list.push_back(&sue);
25.
26.     printPerson(list);
27. }
```

Note that the keyword **const** appears several time here. When defining the type of the collection

```
vector<const Person*>
```

it means that this collection will be a vector of pointers to unmodifiable objects of type **Person**, i.e., that the function cannot change any `person' which is pointed to by an element of the collection (elements themselves are pointers here). The same keyword **const** in definition of a parameter of function **printPerson** means that this is a collection of unmodifiable elements, i.e., that the pointers which are elements of the collection cannot be changed. Therefore, inside the function neither elements of the collection (pointers) nor the objects they point to ("persons") can be modified. As the function `promised' not to change objects pointed to by pointers from the collection, but it calls the **print** method on them, the method *had* to be declared as **const**. But **printPerson** promised also not to change

elements of the collection (i.e., pointers). Therefore, the iterator used *had* to be of type **const_iterator**. Note that we used here **cbegin** and **cend** methods to get iterators — these function are similar to **begin** and **end**, but return **const_iterator** s.

Parameter of **printPerson** is of reference type. This is how the collections are usually passed to functions — passing them by value would lead to copying the whole collection, with all its elements. This was not a problem for arrays, because in that case what was passed was the address of the first element only.

There are also **reverse iterators**, of type **reverse_iterator**, which iterate over collections in reverse order; for example the program

P207: revite.cpp Reverse iterators

```
1. #include <vector>
2. #include <iostream>
3. #include <string>
4. using namespace std;
5.
6. int main() {
7.     vector<string> vs{"Mary", "Lucy", "Ella", "Jill"};
8.
9.     for (auto r = vs.rbegin(); r != vs.rend(); ++r)
10.         cout << *r << " ";
11.     cout << endl;
12. }
```

prints

```
Jill Ella Lucy Mary
```

Here, we used **rbegin** and **rend** methods: they return reverse iterators.

Finally, there is also a type **const_reverse_iterator**.

All iterators fall into a few different categories which determine their functionality. Iterators connected with various types of collections can have different types of iterators.

Collection	Iterator
vector	random access
list	bidirectional
forward_listlist	forward
deque	random access
map	bidirectional
multimap	bidirectional
set	bidirectional
multiset	bidirectional
string	random access
array	random access
valarray	random access

Table 24.1: Iterators related to collections

Iterators of **vector** s are **random access iterators**; they allow for incrementing and decrementing by any integer value, exactly as we can add integers to pointers. For example, after

```
vector<string> vs;
// ...
auto it = vs.begin() + vs.size()/2;
```

iterator ***it*** points to the middle element of the collection. Random access iterators can also be compared by using relational operators (like ' >') or subtracted (the result is the number of elements between locations pointed by these iterators). Elements pointed to by such iterators can be both read and assigned to. Random access iterators are characteristic for vectors **vector**, double-ended queues (**deque**), strings (**string**), i.e., collections of characters.

Somewhat more restricted are **bidirectional iterators**. **bidirectional iterator** They allow for incrementing and decrementing, but only “by one”, i.e., by using '++' and '- -' operators. They do not support pointer arithmetic (adding or subtracting whole numbers). One can test such iterators for equality (using ' ==' and ' ! =' operators), but not compare them (by ' <' or ' >'). They are characteristic for lists.

Even less functionality is provided by **forward iterators**. They behave like bidirectional iterators, but allow for moving in one direction only: forwards.

Finally, the most restricted sorts of iterators are **input iterators** and **output iterators**. They only allow `single pass' iterations in one direction (forward) and, additionally, it is possible to dereference an input iterator to obtain the value it points to, but not to assign a new value through this iterator; similarly, for output iterators it is possible to assign a value through them, but not to read the value which is pointed to. As their names suggest, input and output iterators are related to input and output streams (treated as collections; examples will be given below).

	Out	In	Frwrd	Bidir.	Rand.Acc.
Read	No	Yes	Yes	Yes	Yes
Write	Yes	No	Yes	Yes	Yes
Iterat.	++	++	++	++, - -	++, - -, +, -, +=, -=
Compar.		= =, ! = = =, ! = = =, ! = = =, ! =, <, >, < =, > =			

Table 24.2: Operations on iterators

Iterators play a fundamental rôle in the standard library providing a glue which connects algorithms (functions) with collections that they operate on.

24.1.3 Operations on collections

The standard library defines many useful operations which can be applied to collections. One has to remember, however, that the set of permissible operations and efficiency of those which are permitted depends on the collection at hand. For example: subtracting two iterators connected with a vector gives the number of elements between locations that they point to; this is fast and efficient operation (with constant-time complexity), because vectors are implemented in such a way, that their elements occupy a contiguous region of memory. The same operation can be highly inefficient for large linked lists (**list**) — therefore, there is a special function, **distance**, which serves the same purpose (in

linear time) for lists:

P208: dista.cpp Operations on collections

```
1. #include <iostream>
2. #include <vector>
3. #include <string>
4. #include <list>
5. using namespace std;
6.
7. int main() {
8.     vector<string> vec;
9.
10. #if   defined(__WIN32)
11.     cout << "Enter words (^Z ends):\n";
12. #elif defined(__linux)
13.     cout << "Enter words (^D ends):\n";
14. #else
15.     #error Unknown system
16. #endif
17.
18.     string s;
19.     while ( cin >> s ) vec.push_back(s);
20.     cin.clear();
21.
22.     list<string> lis(vec.begin(),vec.end());          ❶
23.
24.     cout << "Word to find: ";
25.     cin  >> s;
26.
27.     auto sit = vec.cbegin();
28.     auto lit = lis.cbegin();
29.
30.       // vector
31.     for ( ; sit != vec.cend(); ++sit)
32.         if ( *sit == s ) break;
33.     if ( sit != vec.cend() )
34.         cout << "(vec) Word " << s << " on position "
35.              << sit - vec.cbegin() << endl;           ❷
36.     else
37.         cout << "Word " << s << " did not appear" << endl;
38.
39.       // list
40.     for ( ; lit != lis.cend(); ++lit)
41.         if ( *lit == s ) break;
42.     if ( lit != lis.end() )
43.         cout << "(lis) Word " << s << " on position "
44.              << distance(lis.cbegin(),lit) << endl;   ❸
45.     else
46.         cout << "Word " << s << " did not appear" << endl;
47. }
```

Note the constructor of list in line ❶(similar constructors exist for other collections as well) — it takes two iterators to another collection and creates a new collection, possibly of another type, initialized with *copies* of elements from an existing collection (a vector in our case). As always, when we specify a range of elements by two iterators, we mean the range from the element pointed to by the first iterator *inclusive* to the element pointed to by the second iterator *exclusive*. As iterators connected with vectors are random access, we can use subtraction of iterators to find position of an element (❷). For a list, we have to use the function **distance** (❸), as iterators associated with lists are bidirectional only (not random access), so they do not support pointer arithmetic. An example of running this program:

```
List of words (^D ends):
Jill
```

```
Kate
Jane
^D
Word to find: Jane
(vec) Word Jane on position 2
(lis) Word Jane on position 2
```

Entering words can be terminated by pressing Ctrl-D (Ctrl-Z under Windows).

Plain arrays can be, to some extend, treated as collections. Pointers to their elements play then the rôle of iterators. For example, after

```
int arr[] = {1,4,6,8,9};
vector<int> v(arr+1,arr+4);
```

vector ***v*** will contain numbers 4, 6 and 8, as the pointer `arr+1` points to element 4, and pointer `arr+4` to element 9 (and will *not* be copied, since elements pointed to by the second of two iterators defining a range are not included in this range).

As we have already mentioned, also strings (objects of class **string**) can be treated as collections (of characters). A few examples illustrating this feature have been given in section on strings .

Operating on collections, we should be able to erase (remove) elements of the collection. This can be achieved by using method **erase**. It takes an iterator pointing to an element to be erased, or a pair of iterators defining the range of elements to be erased. For example, after

```
vector<Person> os;

os.push_back(Person("Jenny"));
os.push_back(Person("Jill"));
os.push_back(Person("Jane"));
os.push_back(Person("Janet"));

os.erase(os.begin()+1, os.end());
```

all elements of the vector ***os*** except the first one will be erased. Elements should *not* be erased in a loop, because erasing the first will change the collection, thus invalidating iterators pointing to its elements.

Similarly, we can add new elements to a collection using method **insert**. We have to specify the iterator pointing to element in the collection *before* which new elements are to be inserted. New elements which are to be inserted are specified by a pair of iterators to another collection:

```
double tab[] = {2, 4, 6, 7, 8, 1.5, 5, 7};
list<double> lis(5);
lis.insert(lis.begin(),10.5);
lis.insert(lis.begin(),tab+1,tab+3);
```

Note the second line here. We create a list of five elements with default values. For numbers, default value is zero (of an appropriate type); for object types, default constructor will be used (which must, of course, exist). Then we add 10.5 at the beginning of the collection; the remaining elements will be pushed to the right and the collection will now have six elements. In the fourth line, we add, at the new beginning of the list, two numbers

from array ***tab***: these will be elements from positions 1 and 2 of the array, i.e., numbers 4 and 6. The list will now have 8 elements:

```
4 6 10.5 0 0 0 0 0
```

For lists (also **deque** s, but not **vector** s), operations on the beginning of a collection is particularly efficient. Analogously to methods **push_back**, **pop_back** and **back**, which we have already met before and which operate on the end of collections, there are methods **push_front**, **pop_front** and **front**. Generally, operations on beginning and end of collections are much more efficient than operations which remove or erase elements located inside collections.

Prev: 24. The Standard Library (STL) **Up:** 24. The Standard Library (STL) **Next:** 24.2 Algorithms and function objects

Subsections

- 24.1.1 Vectors
- 24.1.2 Iterators
- 24.1.3 Operations on collections

24.1 Collections and iterators

The main part of the library is related to **collections**. These are data structures consisting of elements of the same type and equipped with a set of functions operating on this data (adding and removing elements from a collection, searching, sorting, modifying, copying etc.). Structures like lists, vectors, deques, heaps, stacks, maps are directly implemented; by employing them one can, relatively easy, implement other structures, like various kinds of trees or graphs.

24.1.1 Vectors

The simplest collection is the **vector**, accessible after including the header `vector`. A vector can be thought of as an array with its size unspecified in advance — it grows automatically as we add new elements to it. Elements are located in a well defined order and can be accessed randomly by specifying their index. One can add and remove elements on any position, but adding and removing at the last position is particularly effective (in constant time, not depending on the vector's size). The programmer does not have to worry about memory management — storage is allocated `all by itself' when a vector grows. As **vector** is actually a template, not a class, in order to create a vector we have to concretize this template for a given type — this can also be a built-in type, like **int** or **double**. The default constructor creates an empty vector (there are other constructors). We can add elements using the function **push_back**, which adds a *copy* of an object at the end of the vector (temporary objects may be moved instead, what can be much more effective). A copy, so one has to ensure that objects that are to be added can be correctly copied (or moved). There is also the **emplace_back** method which takes arguments for a constructor and creates the object directly inside the vector.

Elements can be accessed in various ways. The simplest way is to use indices — exactly as for arrays (indexing starts, of course, form zero). This is an efficient way, but sometimes dangerous, as it is *not* checked if the index is valid (non-negative and not equal or larger than the current size). If such a check is important, we can use method **at** supplying an index. Then, if the index is not valid, exception **out_of_range** (from header `stdexcept`), will be thrown and we can handle it appropriately:

P204: ate.cpp Vectors

```
1.  #include <vector>
2.  #include <stdexcept>
3.  #include <iostream>
4.  #include <string>
5.  using namespace std;
6.
7.  int main() {
8.      vector<string> vs;                                  ❶
9.
10.     vs.push_back("Mary");
11.     vs.push_back("Lucy");
12.     vs.push_back("Ella");
13.     vs.push_back("Jill");
14.
15.     try {
16.         for ( int i = 0; i < 5 /* ERROR */; i++ )      ❷
17.             cout << vs.at(i) << " " ;
18.     } catch(out_of_range) {
19.         cout << "\n*** Bad index! *** "
20.              << " vector has only " << vs.size()
21.              << " elements!" << endl;
22.     }
23.     cout << endl;
24.
25.     cout << "First element: " << vs.front() << endl;  ❸
26.     cout << "Last  element: " << vs.back()  << endl;  ❹
27.
28.     vs.pop_back();                                      ❺
29.
30.     cout << "After pop_back: ";
31.     int size = (int)vs.size();                          ❻
32.     for ( int i = 0; i < size; i++)
33.         cout << vs[i] << " " ;
34.     cout << endl;
35. }
```

In the following program, there are five iterations of the loop (❷) although there are only four elements in the vector. Therefore, the last iteration throws an exception (but handled here):

```
Mary Lucy Ella Jill
*** Bad index! ***  vector has only 4 elements!

First element: Mary
Last  element: Jill
After pop_back: Mary Lucy Ella
```

As one can see (❶), object ***vs*** is an object of class **vector<string>** which is a concretization of template **vector** for type **string**.

The collection consists of elements of type **string**. Method **size**, used on line ❻, returns, as one can guess, the size of a collection (not only for vectors). The returned type is an unsigned integer type whose alias is the name **vector::size_type** — that is why we used a cast. We could have used a plain **int**, but that would expose us to an unfriendly compiler's warning. Method **pop_back** (❺) removes the last element but does not return it. Methods **front** and **back** (❸ i ❹) return, by reference, the first and the last elements of the vector without removing them.

24.1.2 Iterators

In order to use collections one has to employ **iterators**. They provide means to access elements of collections and to iterate over them in a type independent way. Semantics of iterators resembles that of pointers, although they do not have to be implemented as

pointers. Their true type is normally of no interest to us — its name (alias) is accessible under the name **coll<Type>::iterator** where ***coll*** is the name of collection's template and **Type** is a type which was used to concretize this template (for example **vector<int> ::iterator** will be the alias of type of iterator related to vectors of **int** s). A collection can be, or we want treat is as being, a collection of constant (immutable) elements. In that case, one has to use separate type of iterator, **const_iterator** (for example, **vector<int> ::const_iterator**).

Iterators have semantics of pointers to elements of an array, i.e., if ***it*** is an iterator pointing to an element of a collection, then `*it` is a reference to this element (consequently, an l-value), and `it->memb` is the name of member ***memb*** of this object (if such a member exists). Similarly, after '`++it`', the iterator ***it*** will point to the next element of the collection.

There are two particularly important methods in all collection classes; they both return iterators connected to the collection that the methods were are called for. Method **begin** returns an iterator pointing to the first element of the collection. Method **end** returns an iterator pointing to a *nonexistent* element which is (or rather would be) the `first after the last' element. [There are also global functions, defined in the `iterator` header, **begin** and **end** which take a collection and return the same iterators.] If we iterate over a collection using an iterator, then it reaches the value returned by **end** when the whole collection has been exhausted and there are no other elements in it. That means that instead of checking whether the index of a loop is still smaller that the size of a collection, as we normally do in loops over arrays, we have to check if the iterator is still *different* than the value returned by **end**:

P205: iterat.cpp Iterators

```
1.  #include <vector>
2.  #include <iostream>
3.  #include <string>
4.  using namespace std;
5.
6.  int main() {
7.      vector<string> vs{"Mary", "Lucy", "Ella","Jill"};
8.
9.      for ( vector<string>::iterator ite = vs.begin();
10.                                     ite!= vs.end(); ++ite)
11.         cout << *ite << " ";
12.     cout << endl;
13.
14.     // or
15.
16.     vector<string>::iterator it, kon = vs.end();
17.
18.     for ( it = vs.begin(); it != kon; ++it )
19.         cout << *it << " ";
20.     cout << endl;
21.
22.     // or
23.
24.     using SIT=vector<string>::iterator;        ❶
25.
26.     SIT iter, fin = vs.end();
27.
28.     for ( iter = vs.begin(); iter != fin; ++iter )
29.         cout << *iter << " ";
30.     cout << endl;
31.
32.     // or the best way to do it
33.
34.     for ( auto i = vs.begin(); i != vs.end(); ++i )
35.         cout << *i << " ";
```

```
36.     cout << endl;
37. }
```

The program above illustrates iteration over a collection. If there are several iterator variables of the same type in a program, one can make life easier by defining, with **typedef** or **using**, a short alias for this type, as we did in line ❶. As we have already said, `++it` advances an iterator by one position, while `*it` is a reference to the element currently pointed to by this iterator. Of course, the most convenient form of such a loop is the last one, with **auto**.

If elements of a collection are unmodifiable (or we want to treat them as such), we have to use a special type of iterator, **const_iterator**. This is often used when we pass a collection to a function which, by assumption, should not change its elements:

P206: constit.cpp Iterators to unmodifiable elements

```
1. #include <vector>
2. #include <iostream>
3. using namespace std;
4.
5. struct Person {
6.     char name[20];
7.     int  year;
8.     void print() const {
9.         cout << name << "-" << year << endl;
10.    }
11. } john = {"John",25}, mary = {"Mary",18}, sue = {"Sue",9};
12.
13.
14. void printPerson(const vector<const Person*>& list) {
15.     for (auto it = list.cbegin(); it != list.cend(); it++)
16.         (*it)->print();
17. }
18.
19. int main() {
20.     vector<const Person*> list;
21.
22.     list.push_back(&john);
23.     list.push_back(&mary);
24.     list.push_back(&sue);
25.
26.     printPerson(list);
27. }
```

Note that the keyword **const** appears several time here. When defining the type of the collection

```
vector<const Person*>
```

it means that this collection will be a vector of pointers to unmodifiable objects of type **Person**, i.e., that the function cannot change any `person' which is pointed to by an element of the collection (elements themselves are pointers here). The same keyword **const** in definition of a parameter of function **printPerson** means that this is a collection of unmodifiable elements, i.e., that the pointers which are elements of the collection cannot be changed. Therefore, inside the function neither elements of the collection (pointers) nor the objects they point to ("persons") can be modified. As the function `promised' not to change objects pointed to by pointers from the collection, but it calls the **print** method on them, the method *had* to be declared as **const**. But **printPerson** promised also not to change

elements of the collection (i.e., pointers). Therefore, the iterator used *had* to be of type **const_iterator**. Note that we used here **cbegin** and **cend** methods to get iterators — these function are similar to **begin** and **end**, but return **const_iterator** s.

Parameter of **printPerson** is of reference type. This is how the collections are usually passed to functions — passing them by value would lead to copying the whole collection, with all its elements. This was not a problem for arrays, because in that case what was passed was the address of the first element only.

There are also **reverse iterators**, of type **reverse_iterator**, which iterate over collections in reverse order; for example the program

P207: revite.cpp Reverse iterators

```
1. #include <vector>
2. #include <iostream>
3. #include <string>
4. using namespace std;
5.
6. int main() {
7.     vector<string> vs{"Mary", "Lucy", "Ella", "Jill"};
8.
9.     for (auto r = vs.rbegin(); r != vs.rend(); ++r)
10.         cout << *r << " ";
11.     cout << endl;
12. }
```

prints

```
Jill Ella Lucy Mary
```

Here, we used **rbegin** and **rend** methods: they return reverse iterators.

Finally, there is also a type **const_reverse_iterator**.

All iterators fall into a few different categories which determine their functionality. Iterators connected with various types of collections can have different types of iterators.

Collection	Iterator
vector	random access
list	bidirectional
forward_listlist	forward
deque	random access
map	bidirectional
multimap	bidirectional
set	bidirectional
multiset	bidirectional
string	random access
array	random access
valarray	random access

Table 24.1: Iterators related to collections

Iterators of **vector** s are **random access iterators**; they allow for incrementing and decrementing by any integer value, exactly as we can add integers to pointers. For example, after

```
vector<string> vs;
// ...
auto it = vs.begin() + vs.size()/2;
```

iterator ***it*** points to the middle element of the collection. Random access iterators can also be compared by using relational operators (like ' >') or subtracted (the result is the number of elements between locations pointed by these iterators). Elements pointed to by such iterators can be both read and assigned to. Random access iterators are characteristic for vectors **vector**, double-ended queues (**deque**), strings (**string**), i.e., collections of characters.

Somewhat more restricted are **bidirectional iterators**. **bidirectional iterator** They allow for incrementing and decrementing, but only “by one”, i.e., by using '++' and '- -' operators. They do not support pointer arithmetic (adding or subtracting whole numbers). One can test such iterators for equality (using ' ==' and ' ! =' operators), but not compare them (by ' <' or ' >'). They are characteristic for lists.

Even less functionality is provided by **forward iterators**. They behave like bidirectional iterators, but allow for moving in one direction only: forwards.

Finally, the most restricted sorts of iterators are **input iterators** and **output iterators**. They only allow `single pass' iterations in one direction (forward) and, additionally, it is possible to dereference an input iterator to obtain the value it points to, but not to assign a new value through this iterator; similarly, for output iterators it is possible to assign a value through them, but not to read the value which is pointed to. As their names suggest, input and output iterators are related to input and output streams (treated as collections; examples will be given below).

	Out	In	Frwrd	Bidir.	Rand.Acc.
Read	No	Yes	Yes	Yes	Yes
Write	Yes	No	Yes	Yes	Yes
Iterat.	++	++	++	++, - -	++, - -, +, -, +=, -=
Compar.		= =, ! = = =, ! = = =, ! = = =, ! =, <, >, < =, > =			

Table 24.2: Operations on iterators

Iterators play a fundamental rôle in the standard library providing a glue which connects algorithms (functions) with collections that they operate on.

24.1.3 Operations on collections

The standard library defines many useful operations which can be applied to collections. One has to remember, however, that the set of permissible operations and efficiency of those which are permitted depends on the collection at hand. For example: subtracting two iterators connected with a vector gives the number of elements between locations that they point to; this is fast and efficient operation (with constant-time complexity), because vectors are implemented in such a way, that their elements occupy a contiguous region of memory. The same operation can be highly inefficient for large linked lists (**list**) — therefore, there is a special function, **distance**, which serves the same purpose (in

linear time) for lists:

P208: dista.cpp Operations on collections

```
1. #include <iostream>
2. #include <vector>
3. #include <string>
4. #include <list>
5. using namespace std;
6.
7. int main() {
8.     vector<string> vec;
9.
10. #if   defined(__WIN32)
11.     cout << "Enter words (^Z ends):\n";
12. #elif defined(__linux)
13.     cout << "Enter words (^D ends):\n";
14. #else
15.     #error Unknown system
16. #endif
17.
18.     string s;
19.     while ( cin >> s ) vec.push_back(s);
20.     cin.clear();
21.
22.     list<string> lis(vec.begin(),vec.end());          ❶
23.
24.     cout << "Word to find: ";
25.     cin  >> s;
26.
27.     auto sit = vec.cbegin();
28.     auto lit = lis.cbegin();
29.
30.       // vector
31.     for ( ; sit != vec.cend(); ++sit)
32.         if ( *sit == s ) break;
33.     if ( sit != vec.cend() )
34.         cout << "(vec) Word " << s << " on position "
35.              << sit - vec.cbegin() << endl;           ❷
36.     else
37.         cout << "Word " << s << " did not appear" << endl;
38.
39.       // list
40.     for ( ; lit != lis.cend(); ++lit)
41.         if ( *lit == s ) break;
42.     if ( lit != lis.end() )
43.         cout << "(lis) Word " << s << " on position "
44.              << distance(lis.cbegin(),lit) << endl;   ❸
45.     else
46.         cout << "Word " << s << " did not appear" << endl;
47. }
```

Note the constructor of list in line ❶(similar constructors exist for other collections as well) — it takes two iterators to another collection and creates a new collection, possibly of another type, initialized with *copies* of elements from an existing collection (a vector in our case). As always, when we specify a range of elements by two iterators, we mean the range from the element pointed to by the first iterator *inclusive* to the element pointed to by the second iterator *exclusive*. As iterators connected with vectors are random access, we can use subtraction of iterators to find position of an element (❷). For a list, we have to use the function **distance** (❸), as iterators associated with lists are bidirectional only (not random access), so they do not support pointer arithmetic. An example of running this program:

```
List of words (^D ends):
Jill
```

```
Kate
Jane
^D
Word to find: Jane
(vec) Word Jane on position 2
(lis) Word Jane on position 2
```

Entering words can be terminated by pressing Ctrl-D (Ctrl-Z under Windows).

Plain arrays can be, to some extend, treated as collections. Pointers to their elements play then the rôle of iterators. For example, after

```
int arr[] = {1,4,6,8,9};
vector<int> v(arr+1,arr+4);
```

vector ***v*** will contain numbers 4, 6 and 8, as the pointer `arr+1` points to element 4, and pointer `arr+4` to element 9 (and will *not* be copied, since elements pointed to by the second of two iterators defining a range are not included in this range).

As we have already mentioned, also strings (objects of class **string**) can be treated as collections (of characters). A few examples illustrating this feature have been given in section on strings .

Operating on collections, we should be able to erase (remove) elements of the collection. This can be achieved by using method **erase**. It takes an iterator pointing to an element to be erased, or a pair of iterators defining the range of elements to be erased. For example, after

```
vector<Person> os;

os.push_back(Person("Jenny"));
os.push_back(Person("Jill"));
os.push_back(Person("Jane"));
os.push_back(Person("Janet"));

os.erase(os.begin()+1, os.end());
```

all elements of the vector ***os*** except the first one will be erased. Elements should *not* be erased in a loop, because erasing the first will change the collection, thus invalidating iterators pointing to its elements.

Similarly, we can add new elements to a collection using method **insert**. We have to specify the iterator pointing to element in the collection *before* which new elements are to be inserted. New elements which are to be inserted are specified by a pair of iterators to another collection:

```
double tab[] = {2, 4, 6, 7, 8, 1.5, 5, 7};
list<double> lis(5);
lis.insert(lis.begin(),10.5);
lis.insert(lis.begin(),tab+1,tab+3);
```

Note the second line here. We create a list of five elements with default values. For numbers, default value is zero (of an appropriate type); for object types, default constructor will be used (which must, of course, exist). Then we add 10.5 at the beginning of the collection; the remaining elements will be pushed to the right and the collection will now have six elements. In the fourth line, we add, at the new beginning of the list, two numbers

from array ***tab***: these will be elements from positions 1 and 2 of the array, i.e., numbers 4 and 6. The list will now have 8 elements:

```
4 6 10.5 0 0 0 0 0
```

For lists (also **deque** s, but not **vector** s), operations on the beginning of a collection is particularly efficient. Analogously to methods **push_back**, **pop_back** and **back**, which we have already met before and which operate on the end of collections, there are methods **push_front**, **pop_front** and **front**. Generally, operations on beginning and end of collections are much more efficient than operations which remove or erase elements located inside collections.

Prev: 24. The Standard Library (STL) **Up:** 24. The Standard Library (STL) **Next:** 24.2 Algorithms and function objects

Subsections

- 24.1.1 Vectors
- 24.1.2 Iterators
- 24.1.3 Operations on collections

24.1 Collections and iterators

The main part of the library is related to **collections**. These are data structures consisting of elements of the same type and equipped with a set of functions operating on this data (adding and removing elements from a collection, searching, sorting, modifying, copying etc.). Structures like lists, vectors, deques, heaps, stacks, maps are directly implemented; by employing them one can, relatively easy, implement other structures, like various kinds of trees or graphs.

24.1.1 Vectors

The simplest collection is the **vector**, accessible after including the header `vector`. A vector can be thought of as an array with its size unspecified in advance — it grows automatically as we add new elements to it. Elements are located in a well defined order and can be accessed randomly by specifying their index. One can add and remove elements on any position, but adding and removing at the last position is particularly effective (in constant time, not depending on the vector's size). The programmer does not have to worry about memory management — storage is allocated `all by itself' when a vector grows. As **vector** is actually a template, not a class, in order to create a vector we have to concretize this template for a given type — this can also be a built-in type, like **int** or **double**. The default constructor creates an empty vector (there are other constructors). We can add elements using the function **push_back**, which adds a *copy* of an object at the end of the vector (temporary objects may be moved instead, what can be much more effective). A copy, so one has to ensure that objects that are to be added can be correctly copied (or moved). There is also the **emplace_back** method which takes arguments for a constructor and creates the object directly inside the vector.

Elements can be accessed in various ways. The simplest way is to use indices — exactly as for arrays (indexing starts, of course, form zero). This is an efficient way, but sometimes dangerous, as it is *not* checked if the index is valid (non-negative and not equal or larger than the current size). If such a check is important, we can use method **at** supplying an index. Then, if the index is not valid, exception **out_of_range** (from header `stdexcept`), will be thrown and we can handle it appropriately:

P204: ate.cpp Vectors

```
1.  #include <vector>
2.  #include <stdexcept>
3.  #include <iostream>
4.  #include <string>
5.  using namespace std;
6.
7.  int main() {
8.      vector<string> vs;                                  ❶
9.
10.     vs.push_back("Mary");
11.     vs.push_back("Lucy");
12.     vs.push_back("Ella");
13.     vs.push_back("Jill");
14.
15.     try {
16.         for ( int i = 0; i < 5 /* ERROR */; i++ )      ❷
17.             cout << vs.at(i) << " " ;
18.     } catch(out_of_range) {
19.         cout << "\n*** Bad index! *** "
20.              << " vector has only " << vs.size()
21.              << " elements!" << endl;
22.     }
23.     cout << endl;
24.
25.     cout << "First element: " << vs.front() << endl;  ❸
26.     cout << "Last  element: " << vs.back()  << endl;  ❹
27.
28.     vs.pop_back();                                      ❺
29.
30.     cout << "After pop_back: ";
31.     int size = (int)vs.size();                          ❻
32.     for ( int i = 0; i < size; i++)
33.         cout << vs[i] << " " ;
34.     cout << endl;
35. }
```

In the following program, there are five iterations of the loop (❷) although there are only four elements in the vector. Therefore, the last iteration throws an exception (but handled here):

```
Mary Lucy Ella Jill
*** Bad index! ***  vector has only 4 elements!

First element: Mary
Last  element: Jill
After pop_back: Mary Lucy Ella
```

As one can see (❶), object ***vs*** is an object of class **vector<string>** which is a concretization of template **vector** for type **string**.

The collection consists of elements of type **string**. Method **size**, used on line ❻, returns, as one can guess, the size of a collection (not only for vectors). The returned type is an unsigned integer type whose alias is the name **vector::size_type** — that is why we used a cast. We could have used a plain **int**, but that would expose us to an unfriendly compiler's warning. Method **pop_back** (❺) removes the last element but does not return it. Methods **front** and **back** (❸ i ❹) return, by reference, the first and the last elements of the vector without removing them.

24.1.2 Iterators

In order to use collections one has to employ **iterators**. They provide means to access elements of collections and to iterate over them in a type independent way. Semantics of iterators resembles that of pointers, although they do not have to be implemented as

pointers. Their true type is normally of no interest to us — its name (alias) is accessible under the name **coll<Type>::iterator** where ***coll*** is the name of collection's template and **Type** is a type which was used to concretize this template (for example **vector<int> ::iterator** will be the alias of type of iterator related to vectors of **int** s). A collection can be, or we want treat is as being, a collection of constant (immutable) elements. In that case, one has to use separate type of iterator, **const_iterator** (for example, **vector<int> ::const_iterator**).

Iterators have semantics of pointers to elements of an array, i.e., if ***it*** is an iterator pointing to an element of a collection, then `*it` is a reference to this element (consequently, an l-value), and `it->memb` is the name of member ***memb*** of this object (if such a member exists). Similarly, after '`++it`', the iterator ***it*** will point to the next element of the collection.

There are two particularly important methods in all collection classes; they both return iterators connected to the collection that the methods were are called for. Method **begin** returns an iterator pointing to the first element of the collection. Method **end** returns an iterator pointing to a *nonexistent* element which is (or rather would be) the `first after the last' element. [There are also global functions, defined in the `iterator` header, **begin** and **end** which take a collection and return the same iterators.] If we iterate over a collection using an iterator, then it reaches the value returned by **end** when the whole collection has been exhausted and there are no other elements in it. That means that instead of checking whether the index of a loop is still smaller that the size of a collection, as we normally do in loops over arrays, we have to check if the iterator is still *different* than the value returned by **end**:

P205: iterat.cpp Iterators

```
1.  #include <vector>
2.  #include <iostream>
3.  #include <string>
4.  using namespace std;
5.
6.  int main() {
7.      vector<string> vs{"Mary", "Lucy", "Ella","Jill"};
8.
9.      for ( vector<string>::iterator ite = vs.begin();
10.                                     ite!= vs.end(); ++ite)
11.         cout << *ite << " ";
12.     cout << endl;
13.
14.     // or
15.
16.     vector<string>::iterator it, kon = vs.end();
17.
18.     for ( it = vs.begin(); it != kon; ++it )
19.         cout << *it << " ";
20.     cout << endl;
21.
22.     // or
23.
24.     using SIT=vector<string>::iterator;        ❶
25.
26.     SIT iter, fin = vs.end();
27.
28.     for ( iter = vs.begin(); iter != fin; ++iter )
29.         cout << *iter << " ";
30.     cout << endl;
31.
32.     // or the best way to do it
33.
34.     for ( auto i = vs.begin(); i != vs.end(); ++i )
35.         cout << *i << " ";
```

```
36.     cout << endl;
37. }
```

The program above illustrates iteration over a collection. If there are several iterator variables of the same type in a program, one can make life easier by defining, with **typedef** or **using**, a short alias for this type, as we did in line ❶. As we have already said, `++it` advances an iterator by one position, while `*it` is a reference to the element currently pointed to by this iterator. Of course, the most convenient form of such a loop is the last one, with **auto**.

If elements of a collection are unmodifiable (or we want to treat them as such), we have to use a special type of iterator, **const_iterator**. This is often used when we pass a collection to a function which, by assumption, should not change its elements:

P206: constit.cpp Iterators to unmodifiable elements

```
1. #include <vector>
2. #include <iostream>
3. using namespace std;
4.
5. struct Person {
6.     char name[20];
7.     int  year;
8.     void print() const {
9.         cout << name << "-" << year << endl;
10.    }
11. } john = {"John",25}, mary = {"Mary",18}, sue = {"Sue",9};
12.
13.
14. void printPerson(const vector<const Person*>& list) {
15.     for (auto it = list.cbegin(); it != list.cend(); it++)
16.         (*it)->print();
17. }
18.
19. int main() {
20.     vector<const Person*> list;
21.
22.     list.push_back(&john);
23.     list.push_back(&mary);
24.     list.push_back(&sue);
25.
26.     printPerson(list);
27. }
```

Note that the keyword **const** appears several time here. When defining the type of the collection

```
vector<const Person*>
```

it means that this collection will be a vector of pointers to unmodifiable objects of type **Person**, i.e., that the function cannot change any `person' which is pointed to by an element of the collection (elements themselves are pointers here). The same keyword **const** in definition of a parameter of function **printPerson** means that this is a collection of unmodifiable elements, i.e., that the pointers which are elements of the collection cannot be changed. Therefore, inside the function neither elements of the collection (pointers) nor the objects they point to ("persons") can be modified. As the function `promised' not to change objects pointed to by pointers from the collection, but it calls the **print** method on them, the method *had* to be declared as **const**. But **printPerson** promised also not to change

elements of the collection (i.e., pointers). Therefore, the iterator used *had* to be of type **const_iterator**. Note that we used here **cbegin** and **cend** methods to get iterators — these function are similar to **begin** and **end**, but return **const_iterator** s.

Parameter of **printPerson** is of reference type. This is how the collections are usually passed to functions — passing them by value would lead to copying the whole collection, with all its elements. This was not a problem for arrays, because in that case what was passed was the address of the first element only.

There are also **reverse iterators**, of type **reverse_iterator**, which iterate over collections in reverse order; for example the program

P207: revite.cpp Reverse iterators

```
1. #include <vector>
2. #include <iostream>
3. #include <string>
4. using namespace std;
5.
6. int main() {
7.     vector<string> vs{"Mary", "Lucy", "Ella", "Jill"};
8.
9.     for (auto r = vs.rbegin(); r != vs.rend(); ++r)
10.         cout << *r << " ";
11.     cout << endl;
12. }
```

prints

```
Jill Ella Lucy Mary
```

Here, we used **rbegin** and **rend** methods: they return reverse iterators.

Finally, there is also a type **const_reverse_iterator**.

All iterators fall into a few different categories which determine their functionality. Iterators connected with various types of collections can have different types of iterators.

Collection	Iterator
vector	random access
list	bidirectional
forward_listlist	forward
deque	random access
map	bidirectional
multimap	bidirectional
set	bidirectional
multiset	bidirectional
string	random access
array	random access
valarray	random access

Table 24.1: Iterators related to collections

Iterators of **vector** s are **random access iterators**; they allow for incrementing and decrementing by any integer value, exactly as we can add integers to pointers. For example, after

```
vector<string> vs;
// ...
auto it = vs.begin() + vs.size()/2;
```

iterator ***it*** points to the middle element of the collection. Random access iterators can also be compared by using relational operators (like ' >') or subtracted (the result is the number of elements between locations pointed by these iterators). Elements pointed to by such iterators can be both read and assigned to. Random access iterators are characteristic for vectors **vector**, double-ended queues (**deque**), strings (**string**), i.e., collections of characters.

Somewhat more restricted are **bidirectional iterators**. **bidirectional iterator** They allow for incrementing and decrementing, but only “by one”, i.e., by using '++' and '- -' operators. They do not support pointer arithmetic (adding or subtracting whole numbers). One can test such iterators for equality (using ' ==' and ' ! =' operators), but not compare them (by ' <' or ' >'). They are characteristic for lists.

Even less functionality is provided by **forward iterators**. They behave like bidirectional iterators, but allow for moving in one direction only: forwards.

Finally, the most restricted sorts of iterators are **input iterators** and **output iterators**. They only allow `single pass' iterations in one direction (forward) and, additionally, it is possible to dereference an input iterator to obtain the value it points to, but not to assign a new value through this iterator; similarly, for output iterators it is possible to assign a value through them, but not to read the value which is pointed to. As their names suggest, input and output iterators are related to input and output streams (treated as collections; examples will be given below).

	Out	In	Frwrd	Bidir.	Rand.Acc.
Read	No	Yes	Yes	Yes	Yes
Write	Yes	No	Yes	Yes	Yes
Iterat.	++	++	++	++, - -	++, - -, +, -, +=, -=
Compar.		= =, ! = = =, ! = = =, ! = = =, ! =, <, >, < =, > =			

Table 24.2: Operations on iterators

Iterators play a fundamental rôle in the standard library providing a glue which connects algorithms (functions) with collections that they operate on.

24.1.3 Operations on collections

The standard library defines many useful operations which can be applied to collections. One has to remember, however, that the set of permissible operations and efficiency of those which are permitted depends on the collection at hand. For example: subtracting two iterators connected with a vector gives the number of elements between locations that they point to; this is fast and efficient operation (with constant-time complexity), because vectors are implemented in such a way, that their elements occupy a contiguous region of memory. The same operation can be highly inefficient for large linked lists (**list**) — therefore, there is a special function, **distance**, which serves the same purpose (in

linear time) for lists:

P208: dista.cpp Operations on collections

```
1. #include <iostream>
2. #include <vector>
3. #include <string>
4. #include <list>
5. using namespace std;
6.
7. int main() {
8.     vector<string> vec;
9.
10. #if   defined(__WIN32)
11.     cout << "Enter words (^Z ends):\n";
12. #elif defined(__linux)
13.     cout << "Enter words (^D ends):\n";
14. #else
15.     #error Unknown system
16. #endif
17.
18.     string s;
19.     while ( cin >> s ) vec.push_back(s);
20.     cin.clear();
21.
22.     list<string> lis(vec.begin(),vec.end());          ❶
23.
24.     cout << "Word to find: ";
25.     cin  >> s;
26.
27.     auto sit = vec.cbegin();
28.     auto lit = lis.cbegin();
29.
30.       // vector
31.     for ( ; sit != vec.cend(); ++sit)
32.         if ( *sit == s ) break;
33.     if ( sit != vec.cend() )
34.         cout << "(vec) Word " << s << " on position "
35.              << sit - vec.cbegin() << endl;           ❷
36.     else
37.         cout << "Word " << s << " did not appear" << endl;
38.
39.       // list
40.     for ( ; lit != lis.cend(); ++lit)
41.         if ( *lit == s ) break;
42.     if ( lit != lis.end() )
43.         cout << "(lis) Word " << s << " on position "
44.              << distance(lis.cbegin(),lit) << endl;   ❸
45.     else
46.         cout << "Word " << s << " did not appear" << endl;
47. }
```

Note the constructor of list in line ❶(similar constructors exist for other collections as well) — it takes two iterators to another collection and creates a new collection, possibly of another type, initialized with *copies* of elements from an existing collection (a vector in our case). As always, when we specify a range of elements by two iterators, we mean the range from the element pointed to by the first iterator *inclusive* to the element pointed to by the second iterator *exclusive*. As iterators connected with vectors are random access, we can use subtraction of iterators to find position of an element (❷). For a list, we have to use the function **distance** (❸), as iterators associated with lists are bidirectional only (not random access), so they do not support pointer arithmetic. An example of running this program:

```
List of words (^D ends):
Jill
```

```
Kate
Jane
^D
Word to find: Jane
(vec) Word Jane on position 2
(lis) Word Jane on position 2
```

Entering words can be terminated by pressing Ctrl-D (Ctrl-Z under Windows).

Plain arrays can be, to some extend, treated as collections. Pointers to their elements play then the rôle of iterators. For example, after

```
int arr[] = {1,4,6,8,9};
vector<int> v(arr+1,arr+4);
```

vector ***v*** will contain numbers 4, 6 and 8, as the pointer `arr+1` points to element 4, and pointer `arr+4` to element 9 (and will *not* be copied, since elements pointed to by the second of two iterators defining a range are not included in this range).

As we have already mentioned, also strings (objects of class **string**) can be treated as collections (of characters). A few examples illustrating this feature have been given in section on strings .

Operating on collections, we should be able to erase (remove) elements of the collection. This can be achieved by using method **erase**. It takes an iterator pointing to an element to be erased, or a pair of iterators defining the range of elements to be erased. For example, after

```
vector<Person> os;

os.push_back(Person("Jenny"));
os.push_back(Person("Jill"));
os.push_back(Person("Jane"));
os.push_back(Person("Janet"));

os.erase(os.begin()+1, os.end());
```

all elements of the vector ***os*** except the first one will be erased. Elements should *not* be erased in a loop, because erasing the first will change the collection, thus invalidating iterators pointing to its elements.

Similarly, we can add new elements to a collection using method **insert**. We have to specify the iterator pointing to element in the collection *before* which new elements are to be inserted. New elements which are to be inserted are specified by a pair of iterators to another collection:

```
double tab[] = {2, 4, 6, 7, 8, 1.5, 5, 7};
list<double> lis(5);
lis.insert(lis.begin(),10.5);
lis.insert(lis.begin(),tab+1,tab+3);
```

Note the second line here. We create a list of five elements with default values. For numbers, default value is zero (of an appropriate type); for object types, default constructor will be used (which must, of course, exist). Then we add 10.5 at the beginning of the collection; the remaining elements will be pushed to the right and the collection will now have six elements. In the fourth line, we add, at the new beginning of the list, two numbers

from array ***tab***: these will be elements from positions 1 and 2 of the array, i.e., numbers 4 and 6. The list will now have 8 elements:

```
4 6 10.5 0 0 0 0 0
```

For lists (also **deque** s, but not **vector** s), operations on the beginning of a collection is particularly efficient. Analogously to methods **push_back**, **pop_back** and **back**, which we have already met before and which operate on the end of collections, there are methods **push_front**, **pop_front** and **front**. Generally, operations on beginning and end of collections are much more efficient than operations which remove or erase elements located inside collections.

Prev: 24. The Standard Library (STL) **Up:** 24. The Standard Library (STL) **Next:** 24.2 Algorithms and function objects

Subsections

- 24.1.1 Vectors
- 24.1.2 Iterators
- 24.1.3 Operations on collections

24.1 Collections and iterators

The main part of the library is related to **collections**. These are data structures consisting of elements of the same type and equipped with a set of functions operating on this data (adding and removing elements from a collection, searching, sorting, modifying, copying etc.). Structures like lists, vectors, deques, heaps, stacks, maps are directly implemented; by employing them one can, relatively easy, implement other structures, like various kinds of trees or graphs.

24.1.1 Vectors

The simplest collection is the **vector**, accessible after including the header `vector`. A vector can be thought of as an array with its size unspecified in advance — it grows automatically as we add new elements to it. Elements are located in a well defined order and can be accessed randomly by specifying their index. One can add and remove elements on any position, but adding and removing at the last position is particularly effective (in constant time, not depending on the vector's size). The programmer does not have to worry about memory management — storage is allocated `all by itself' when a vector grows. As **vector** is actually a template, not a class, in order to create a vector we have to concretize this template for a given type — this can also be a built-in type, like **int** or **double**. The default constructor creates an empty vector (there are other constructors). We can add elements using the function **push_back**, which adds a *copy* of an object at the end of the vector (temporary objects may be moved instead, what can be much more effective). A copy, so one has to ensure that objects that are to be added can be correctly copied (or moved). There is also the **emplace_back** method which takes arguments for a constructor and creates the object directly inside the vector.

Elements can be accessed in various ways. The simplest way is to use indices — exactly as for arrays (indexing starts, of course, form zero). This is an efficient way, but sometimes dangerous, as it is *not* checked if the index is valid (non-negative and not equal or larger than the current size). If such a check is important, we can use method **at** supplying an index. Then, if the index is not valid, exception **out_of_range** (from header `stdexcept`), will be thrown and we can handle it appropriately:

P204: ate.cpp Vectors

```
1.  #include <vector>
2.  #include <stdexcept>
3.  #include <iostream>
4.  #include <string>
5.  using namespace std;
6.
7.  int main() {
8.      vector<string> vs;                                  ❶
9.
10.     vs.push_back("Mary");
11.     vs.push_back("Lucy");
12.     vs.push_back("Ella");
13.     vs.push_back("Jill");
14.
15.     try {
16.         for ( int i = 0; i < 5 /* ERROR */; i++ )      ❷
17.             cout << vs.at(i) << " " ;
18.     } catch(out_of_range) {
19.         cout << "\n*** Bad index! *** "
20.              << " vector has only " << vs.size()
21.              << " elements!" << endl;
22.     }
23.     cout << endl;
24.
25.     cout << "First element: " << vs.front() << endl;  ❸
26.     cout << "Last  element: " << vs.back()  << endl;  ❹
27.
28.     vs.pop_back();                                      ❺
29.
30.     cout << "After pop_back: ";
31.     int size = (int)vs.size();                          ❻
32.     for ( int i = 0; i < size; i++)
33.         cout << vs[i] << " " ;
34.     cout << endl;
35. }
```

In the following program, there are five iterations of the loop (❷) although there are only four elements in the vector. Therefore, the last iteration throws an exception (but handled here):

```
Mary Lucy Ella Jill
*** Bad index! ***  vector has only 4 elements!

First element: Mary
Last  element: Jill
After pop_back: Mary Lucy Ella
```

As one can see (❶), object ***vs*** is an object of class **vector<string>** which is a concretization of template **vector** for type **string**.

The collection consists of elements of type **string**. Method **size**, used on line ❻, returns, as one can guess, the size of a collection (not only for vectors). The returned type is an unsigned integer type whose alias is the name **vector::size_type** — that is why we used a cast. We could have used a plain **int**, but that would expose us to an unfriendly compiler's warning. Method **pop_back** (❺) removes the last element but does not return it. Methods **front** and **back** (❸ i ❹) return, by reference, the first and the last elements of the vector without removing them.

24.1.2 Iterators

In order to use collections one has to employ **iterators**. They provide means to access elements of collections and to iterate over them in a type independent way. Semantics of iterators resembles that of pointers, although they do not have to be implemented as

pointers. Their true type is normally of no interest to us — its name (alias) is accessible under the name **coll<Type>::iterator** where ***coll*** is the name of collection's template and **Type** is a type which was used to concretize this template (for example **vector<int> ::iterator** will be the alias of type of iterator related to vectors of **int** s). A collection can be, or we want treat is as being, a collection of constant (immutable) elements. In that case, one has to use separate type of iterator, **const_iterator** (for example, **vector<int> ::const_iterator**).

Iterators have semantics of pointers to elements of an array, i.e., if ***it*** is an iterator pointing to an element of a collection, then `*it` is a reference to this element (consequently, an l-value), and `it->memb` is the name of member ***memb*** of this object (if such a member exists). Similarly, after '`++it`', the iterator ***it*** will point to the next element of the collection.

There are two particularly important methods in all collection classes; they both return iterators connected to the collection that the methods were are called for. Method **begin** returns an iterator pointing to the first element of the collection. Method **end** returns an iterator pointing to a *nonexistent* element which is (or rather would be) the `first after the last' element. [There are also global functions, defined in the `iterator` header, **begin** and **end** which take a collection and return the same iterators.] If we iterate over a collection using an iterator, then it reaches the value returned by **end** when the whole collection has been exhausted and there are no other elements in it. That means that instead of checking whether the index of a loop is still smaller that the size of a collection, as we normally do in loops over arrays, we have to check if the iterator is still *different* than the value returned by **end**:

P205: iterat.cpp Iterators

```
1.  #include <vector>
2.  #include <iostream>
3.  #include <string>
4.  using namespace std;
5.
6.  int main() {
7.      vector<string> vs{"Mary", "Lucy", "Ella","Jill"};
8.
9.      for ( vector<string>::iterator ite = vs.begin();
10.                                     ite!= vs.end(); ++ite)
11.         cout << *ite << " ";
12.     cout << endl;
13.
14.     // or
15.
16.     vector<string>::iterator it, kon = vs.end();
17.
18.     for ( it = vs.begin(); it != kon; ++it )
19.         cout << *it << " ";
20.     cout << endl;
21.
22.     // or
23.
24.     using SIT=vector<string>::iterator;        ❶
25.
26.     SIT iter, fin = vs.end();
27.
28.     for ( iter = vs.begin(); iter != fin; ++iter )
29.         cout << *iter << " ";
30.     cout << endl;
31.
32.     // or the best way to do it
33.
34.     for ( auto i = vs.begin(); i != vs.end(); ++i )
35.         cout << *i << " ";
```

```
36.     cout << endl;
37. }
```

The program above illustrates iteration over a collection. If there are several iterator variables of the same type in a program, one can make life easier by defining, with **typedef** or **using**, a short alias for this type, as we did in line ❶. As we have already said, `++it` advances an iterator by one position, while `*it` is a reference to the element currently pointed to by this iterator. Of course, the most convenient form of such a loop is the last one, with **auto**.

If elements of a collection are unmodifiable (or we want to treat them as such), we have to use a special type of iterator, **const_iterator**. This is often used when we pass a collection to a function which, by assumption, should not change its elements:

P206: constit.cpp Iterators to unmodifiable elements

```
1. #include <vector>
2. #include <iostream>
3. using namespace std;
4.
5. struct Person {
6.     char name[20];
7.     int  year;
8.     void print() const {
9.         cout << name << "-" << year << endl;
10.    }
11. } john = {"John",25}, mary = {"Mary",18}, sue = {"Sue",9};
12.
13.
14. void printPerson(const vector<const Person*>& list) {
15.     for (auto it = list.cbegin(); it != list.cend(); it++)
16.         (*it)->print();
17. }
18.
19. int main() {
20.     vector<const Person*> list;
21.
22.     list.push_back(&john);
23.     list.push_back(&mary);
24.     list.push_back(&sue);
25.
26.     printPerson(list);
27. }
```

Note that the keyword **const** appears several time here. When defining the type of the collection

```
vector<const Person*>
```

it means that this collection will be a vector of pointers to unmodifiable objects of type **Person**, i.e., that the function cannot change any `person' which is pointed to by an element of the collection (elements themselves are pointers here). The same keyword **const** in definition of a parameter of function **printPerson** means that this is a collection of unmodifiable elements, i.e., that the pointers which are elements of the collection cannot be changed. Therefore, inside the function neither elements of the collection (pointers) nor the objects they point to ("persons") can be modified. As the function `promised' not to change objects pointed to by pointers from the collection, but it calls the **print** method on them, the method *had* to be declared as **const**. But **printPerson** promised also not to change

elements of the collection (i.e., pointers). Therefore, the iterator used *had* to be of type **const_iterator**. Note that we used here **cbegin** and **cend** methods to get iterators — these function are similar to **begin** and **end**, but return **const_iterator** s.

Parameter of **printPerson** is of reference type. This is how the collections are usually passed to functions — passing them by value would lead to copying the whole collection, with all its elements. This was not a problem for arrays, because in that case what was passed was the address of the first element only.

There are also **reverse iterators**, of type **reverse_iterator**, which iterate over collections in reverse order; for example the program

P207: revite.cpp Reverse iterators

```
1. #include <vector>
2. #include <iostream>
3. #include <string>
4. using namespace std;
5.
6. int main() {
7.     vector<string> vs{"Mary", "Lucy", "Ella", "Jill"};
8.
9.     for (auto r = vs.rbegin(); r != vs.rend(); ++r)
10.         cout << *r << " ";
11.     cout << endl;
12. }
```

prints

```
Jill Ella Lucy Mary
```

Here, we used **rbegin** and **rend** methods: they return reverse iterators.

Finally, there is also a type **const_reverse_iterator**.

All iterators fall into a few different categories which determine their functionality. Iterators connected with various types of collections can have different types of iterators.

Collection	Iterator
vector	random access
list	bidirectional
forward_listlist	forward
deque	random access
map	bidirectional
multimap	bidirectional
set	bidirectional
multiset	bidirectional
string	random access
array	random access
valarray	random access

Table 24.1: Iterators related to collections

Iterators of **vector** s are **random access iterators**; they allow for incrementing and decrementing by any integer value, exactly as we can add integers to pointers. For example, after

```
vector<string> vs;
// ...
auto it = vs.begin() + vs.size()/2;
```

iterator ***it*** points to the middle element of the collection. Random access iterators can also be compared by using relational operators (like ' >') or subtracted (the result is the number of elements between locations pointed by these iterators). Elements pointed to by such iterators can be both read and assigned to. Random access iterators are characteristic for vectors **vector**, double-ended queues (**deque**), strings (**string**), i.e., collections of characters.

Somewhat more restricted are **bidirectional iterators**. **bidirectional iterator** They allow for incrementing and decrementing, but only “by one”, i.e., by using '++' and '- -' operators. They do not support pointer arithmetic (adding or subtracting whole numbers). One can test such iterators for equality (using ' ==' and ' ! =' operators), but not compare them (by ' <' or ' >'). They are characteristic for lists.

Even less functionality is provided by **forward iterators**. They behave like bidirectional iterators, but allow for moving in one direction only: forwards.

Finally, the most restricted sorts of iterators are **input iterators** and **output iterators**. They only allow `single pass' iterations in one direction (forward) and, additionally, it is possible to dereference an input iterator to obtain the value it points to, but not to assign a new value through this iterator; similarly, for output iterators it is possible to assign a value through them, but not to read the value which is pointed to. As their names suggest, input and output iterators are related to input and output streams (treated as collections; examples will be given below).

	Out	In	Frwrd	Bidir.	Rand.Acc.
Read	No	Yes	Yes	Yes	Yes
Write	Yes	No	Yes	Yes	Yes
Iterat.	++	++	++	++, - -	++, - -, +, -, +=, -=
Compar.		= =, ! = = =, ! = = =, ! = = =, ! =, <, >, < =, > =			

Table 24.2: Operations on iterators

Iterators play a fundamental rôle in the standard library providing a glue which connects algorithms (functions) with collections that they operate on.

24.1.3 Operations on collections

The standard library defines many useful operations which can be applied to collections. One has to remember, however, that the set of permissible operations and efficiency of those which are permitted depends on the collection at hand. For example: subtracting two iterators connected with a vector gives the number of elements between locations that they point to; this is fast and efficient operation (with constant-time complexity), because vectors are implemented in such a way, that their elements occupy a contiguous region of memory. The same operation can be highly inefficient for large linked lists (**list**) — therefore, there is a special function, **distance**, which serves the same purpose (in

linear time) for lists:

P208: dista.cpp Operations on collections

```
1. #include <iostream>
2. #include <vector>
3. #include <string>
4. #include <list>
5. using namespace std;
6.
7. int main() {
8.     vector<string> vec;
9.
10. #if   defined(__WIN32)
11.     cout << "Enter words (^Z ends):\n";
12. #elif defined(__linux)
13.     cout << "Enter words (^D ends):\n";
14. #else
15.     #error Unknown system
16. #endif
17.
18.     string s;
19.     while ( cin >> s ) vec.push_back(s);
20.     cin.clear();
21.
22.     list<string> lis(vec.begin(),vec.end());          ❶
23.
24.     cout << "Word to find: ";
25.     cin  >> s;
26.
27.     auto sit = vec.cbegin();
28.     auto lit = lis.cbegin();
29.
30.       // vector
31.     for ( ; sit != vec.cend(); ++sit)
32.         if ( *sit == s ) break;
33.     if ( sit != vec.cend() )
34.         cout << "(vec) Word " << s << " on position "
35.              << sit - vec.cbegin() << endl;           ❷
36.     else
37.         cout << "Word " << s << " did not appear" << endl;
38.
39.       // list
40.     for ( ; lit != lis.cend(); ++lit)
41.         if ( *lit == s ) break;
42.     if ( lit != lis.end() )
43.         cout << "(lis) Word " << s << " on position "
44.              << distance(lis.cbegin(),lit) << endl;   ❸
45.     else
46.         cout << "Word " << s << " did not appear" << endl;
47. }
```

Note the constructor of list in line ❶(similar constructors exist for other collections as well) — it takes two iterators to another collection and creates a new collection, possibly of another type, initialized with *copies* of elements from an existing collection (a vector in our case). As always, when we specify a range of elements by two iterators, we mean the range from the element pointed to by the first iterator *inclusive* to the element pointed to by the second iterator *exclusive*. As iterators connected with vectors are random access, we can use subtraction of iterators to find position of an element (❷). For a list, we have to use the function **distance** (❸), as iterators associated with lists are bidirectional only (not random access), so they do not support pointer arithmetic. An example of running this program:

```
List of words (^D ends):
Jill
```

```
Kate
Jane
^D
Word to find: Jane
(vec) Word Jane on position 2
(lis) Word Jane on position 2
```

Entering words can be terminated by pressing Ctrl-D (Ctrl-Z under Windows).

Plain arrays can be, to some extend, treated as collections. Pointers to their elements play then the rôle of iterators. For example, after

```
int arr[] = {1,4,6,8,9};
vector<int> v(arr+1,arr+4);
```

vector ***v*** will contain numbers 4, 6 and 8, as the pointer `arr+1` points to element 4, and pointer `arr+4` to element 9 (and will *not* be copied, since elements pointed to by the second of two iterators defining a range are not included in this range).

As we have already mentioned, also strings (objects of class **string**) can be treated as collections (of characters). A few examples illustrating this feature have been given in section on strings .

Operating on collections, we should be able to erase (remove) elements of the collection. This can be achieved by using method **erase**. It takes an iterator pointing to an element to be erased, or a pair of iterators defining the range of elements to be erased. For example, after

```
vector<Person> os;

os.push_back(Person("Jenny"));
os.push_back(Person("Jill"));
os.push_back(Person("Jane"));
os.push_back(Person("Janet"));

os.erase(os.begin()+1, os.end());
```

all elements of the vector ***os*** except the first one will be erased. Elements should *not* be erased in a loop, because erasing the first will change the collection, thus invalidating iterators pointing to its elements.

Similarly, we can add new elements to a collection using method **insert**. We have to specify the iterator pointing to element in the collection *before* which new elements are to be inserted. New elements which are to be inserted are specified by a pair of iterators to another collection:

```
double tab[] = {2, 4, 6, 7, 8, 1.5, 5, 7};
list<double> lis(5);
lis.insert(lis.begin(),10.5);
lis.insert(lis.begin(),tab+1,tab+3);
```

Note the second line here. We create a list of five elements with default values. For numbers, default value is zero (of an appropriate type); for object types, default constructor will be used (which must, of course, exist). Then we add 10.5 at the beginning of the collection; the remaining elements will be pushed to the right and the collection will now have six elements. In the fourth line, we add, at the new beginning of the list, two numbers

from array ***tab***: these will be elements from positions 1 and 2 of the array, i.e., numbers 4 and 6. The list will now have 8 elements:

```
4 6 10.5 0 0 0 0 0
```

For lists (also **deque** s, but not **vector** s), operations on the beginning of a collection is particularly efficient. Analogously to methods **push_back**, **pop_back** and **back**, which we have already met before and which operate on the end of collections, there are methods **push_front**, **pop_front** and **front**. Generally, operations on beginning and end of collections are much more efficient than operations which remove or erase elements located inside collections.

Prev: 24. The Standard Library (STL) **Up:** 24. The Standard Library (STL) **Next:** 24.2 Algorithms and function objects

Subsections

- 24.2.1 Algorithms
- 24.2.2 Function objects

24.2 Algorithms and function objects

The standard library provides more than 100 **algorithms**, , i.e., templates of functions operating on collections. Collections as such are never passed as arguments to the algorithms: instead, we always use iterators. Many algorithms take also functions (or function objects) as additional arguments, as we will see below.

24.2.1 Algorithms

We will briefly discuss a few of many algorithms provided by the standard library. They are accessible after including the header `algorithm`.

Sorting algorithms belong to the most useful. Their use is quite simple:

P209: sort.cpp Sorting

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    vector<string> vec{"Paris", "London", "Warsaw",
                       "Berlin", "Lisbon", "Oslo"};
    auto ini = vec.begin(), fin = vec.end();

    list<string> lis(vec.size()-2);                ❶
    copy(ini+1, fin-1, lis.begin());               ❷

    sort(ini,fin);                                 ❸
    lis.sort();                                    ❹

    copy(ini, fin,                                 ❺
         ostream_iterator<string>(cout, " "));
    cout << endl;

    copy(lis.begin(),lis.end(),
         ostream_iterator<string>(cout, " "));
    cout << endl;
}
```

We create a vector of strings. Then we create a list (❶) with initial size by two smaller than the size of the vector. Algorithm **copy** (❷) takes a range (as a pair of iterators) of elements from the source container and iterator of the target container indicating where elements are to be copied. Note that elements are *copied* onto already existing elements of the target container — the algorithm does not create new elements, only overwrites them. That's why we had to ensure that the initial size of the list is sufficient for all the elements from the vector that we want to copy.
We then sort the vector using the algorithm **sort** (❸) — as usual, we pass the range to be sorted as a pair of iterators. Note that **sort** algorithm expects random access iterators. That's why we cannot use it for a list — lists, however, have the *method* **sort** (see ❹). The same mechanism works for other algorithms which require random access iterators; collections providing only bidirectional iterators have corresponding methods defined (e.g., **merge**, **remove**, **reverse** etc.).

Function **copy** was also used in line ❺. This time the third argument is an iterator to a slightly bizarre target collection. This is an iterator created from template **ostream_iterator** (from header `iterator`) by its concretization for type **string**. The constructor of this concretization takes two arguments: an output stream, in our case ***cout***, and a string which will be used as the separator, a space in the example. In this way an output stream can be treated as a collection of elements to be printed (all elements must be of the same type). Iterators of such collections are of type `output iterator' — they are one-directional, single pass iterators which allow only for writing elements to the stream (there are also analogous input iterators). Our example program prints

```
Berlin Lisbon London Oslo Paris Warsaw
Berlin Lisbon London Warsaw
```

As we noted, the **copy** function copies elements from one collection to another overwriting existing elements. That is why, in the program above, we had to create the list (❶) with the appropriate size. This is sometimes not so simple, because very often we don't know how many elements we will copy. In such situations, we can use a special function **back_inserter** which returns an object which behaves like an output iterator, but adds new elements of the collection (by calling **push_back**). For example, in the following program, we copy elements form one vector to another (which is initially empty), but only those which meet some requirement specified by a predicate passed as the last argument to the **copy_if** function:

P210: backins.cpp *back_inserter function*

```
1.  #include <algorithm>    // copy
2.  #include <iostream>
3.  #include <iterator>     // inserters
4.  #include <vector>
5.
6.  int main() {
7.      std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
8.      std::vector<int> emp;
9.      std::copy_if(v.begin(), v.end(),
10.                  std::back_inserter(emp),
11.                  [](auto n){ return n%2 != 0; });
12.     std::copy(emp.begin(), emp.end(),
13.               std::ostream_iterator<int>(std::cout, "\n"));
14. }
```

(there are also similar functions **front_inserter** and **inserter**).

Passing functions to algorithms is rather common. For example, sorting algorithms can take additional argument which determines the way in which elements of a collection are to be compared (normally they are compared using the < operator). This should be a pointer to a function, a lambda or a function object (anything that behaves like a function) and which plays the rôle of a **comparator**, i.e., a function taking two arguments of type corresponding to the type of elements of the collection, and returning a **bool** value which answers the question *is the first argument strictly less than the second?*
For example, in the following program we use comparator **compar** (❶) which states that any odd number is less than any even number, and numbers of the same parity are ordered in the natural way (ascending order). This comparator is used in line ❹ as a function pointer.
We also have a simple structure **Evens** (❷) with overloaded function call operator: objects of these class can also be used as comparators (❺) — this particular comparator considers all even numbers as smaller than odd numbers, and for arguments of the same parity grater values are considered `smaller' (descending order).
Finally, we can also pass a lambda (❻) — it considers greater values as `smaller' so we will get the descending order.
The function (template) **printVec** (❸) just print a vector.

P211: sortev.cpp Comparators

```
1.  #include <iostream>
2.  #include <vector>
3.  #include <algorithm>
4.  #include <iterator>
5.
6.  bool compar(int a, int b) {                          ❶
7.      return (a+b)%2 != 0 ? a%2 != 0 : a < b;
8.  }
9.
10. struct Evens {                                       ❷
11.     bool operator()(int a, int b) {
12.         return (a+b)%2 != 0 ? a%2 == 0 : b < a;
13.     }
14. };
15.
16. template <typename T>
17. void printVec(const std::vector<T>& vec) {           ❸
18.     copy(vec.cbegin(), vec.cend(),
19.         std::ostream_iterator<T>(std::cout, " "));
20.     std::cout << '\n';
21. }
22.
23. int main() {
24.     std::vector<int> vec{2, 5, 2, 9, 1, 5, 7, 4};
25.     printVec(vec);
26.
27.     sort(vec.begin(), vec.end(), compar);            ❹
28.     printVec(vec);
29.
30.     sort(vec.begin(), vec.end(), Evens{});           ❺
31.     printVec(vec);
32.
33.     sort(vec.begin(), vec.end(),
34.          [](int a, int b) {return b < a;});          ❻
35.     printVec(vec);
36.
37. }
```

The program prints:

```
2 5 2 9 1 5 7 4
1 5 5 7 9 2 2 4
4 2 2 9 7 5 5 1
9 7 5 5 4 2 2 1
```

Generally, functions returning **bool** values are called **predicates**. They are extensively used by many algorithms, not only sorting ones. For example, function (strictly speaking, a template) **count_if** counts how many elements from a range of a collection defined by two iterators yield **true** when passed as the only argument to a predicate. In the example below, predicate **even** (line 21) returns **true** only for even numbers, so the algorithm will count how many elements in the vector are even:

P212: predic.cpp Predicates

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. #include <iterator>
5. using namespace std;
6.
7. bool even(int a) {
8.     return (a&1) == 0;
9. }
10.
11. int main() {
12.     vector<int> vec;
13.     int d;
14.     while ( cin >> d ) vec.push_back(d);
15.     cin.clear();
16.
17.     cout << "In the series ";
18.     copy(vec.begin(),vec.end(),
19.          ostream_iterator<int>(cout, " "));
20.     cout << "\nthere are "
21.          << count_if(vec.begin(), vec.end(), even)
22.          << " even numbers\n";
23. }
```

The program prints:

```
2 4 -2 7 5 4 31 4 5 -4
^D
In the series 2 4 -2 7 5 4 31 4 5 -4
there are 6 even numbers
```

The standard library contains also many searching algorithms, as, for example, **find** and **find_if**. Function **find** searches for the first occurence of element which is equal to an object passed as argument, while **find_if** searches for the first occurence of element for which a predicate passed as arguments returns **true**. As always, collection is defined by a pair of iterators. If the search succeeds, the iterator to the element found is returned; otherwise the functions returns the iterator **end()**. Examples of these (and others) algorithms can be found in the program below:

P213: search.cpp Searching

```
1. #include <vector>
```

```
2.  #include <iostream>
3.  #include <string>
4.  #include <cctype>    // tolower
5.  #include <algorithm>
6.  #include <iterator>
7.  using namespace std;
8.
9.  bool startsWithA(string& s) {
10.     return s[0] == 'A';
11. }
12.
13. void lowerc(string& name) {
14.     name[0] = tolower(name[0]);
15. }
16.
17. int main() {
18.     vector<string> vs;
19.
20.     vs.push_back("Maggie"); vs.push_back("Ann");
21.     vs.push_back("Monica"); vs.push_back("Agatha");
22.     vs.push_back("Alice");  vs.push_back("Ursula");
23.
24.     vector<string>::iterator k;
25.
26.     k = find(vs.begin(), vs.end(), "Ann");
27.     if ( k != vs.end() )
28.         cout << *k << " found\n";
29.     else
30.         cout << "Ann not found\n";
31.
32.     k = find(vs.begin(), vs.end(), "Kate");
33.     if ( k != vs.end() )
34.         cout << *k << " found\n";
35.     else
36.         cout << "Kate not found\n";
37.
38.
39.     cout << "\nAmong names\n";
40.     copy(vs.begin(),vs.end(),
41.          ostream_iterator<string>(cout," "));
42.     cout << "\nthe following start with \'A\':\n";
43.     k = vs.begin();
44.     while ( k < vs.end() ) {
45.         k = find_if(k, vs.end(), startsWithA);
46.         if ( k != vs.end() ) cout << *k++ << " ";
47.     }
48.     cout << endl;
49.
50.     for_each(vs.begin(),vs.end(),lowerc);
51.     cout << "\nAfter changing to lower case:\n";
52.     copy(vs.begin(),vs.end(),
53.          ostream_iterator<string>(cout," "));
54.     cout << endl;
55. }
```

We use the function **find** on lines 26 and 32. Then we check if the iterator returned is not equal to ***vs.end()***, what would mean that the search failed.

In the next part of the program we select from the vector of names those which start with the letter 'A' (loop on lines 44-47). To achieve that, we use **find_if** with predicate **startsWithA** (which is defined on lines 9-11).

Line 50 illustrates algorithm **for_each**. This is an example of **mutating algorithm**, i.e., an algorithm which modifies elements of a collection (searching algorithms did not). Function **for_each** scans all elements from a given range of a collection, and for each of them calls a user defined function (or function object). Elements of the collection are passed by reference, so the function can modify them. Return values, even if they exist, are ignored. In our example, the function (**lowerc**) changes the first letter of passed strings to lower

case. The printout is as follows:

```
Ann found
Kate not found

Among names
Maggie Ann Monica Agatha Alice Ursula
the following start with 'A':
Ann Agatha Alice

After changing to lower case:
maggie ann monica agatha alice ursula
```

24.2.2 Function objects

As we have seen, user defined functions, e.g., predicates, play very important rôle in various algorithms from the standard library. Actually, these are not necessarily functions; they can be “anything that can be called”. As we remember from the chapter on operator overloading , in any class one can overload the function-invocation operator — **operator()**. Then, if the name of an object is followed by round parentheses with appropriate arguments, this function will be called for the object and arguments will be passed as for any other method. Objects of classes which define method **operator()** are called **callable objects**. Callable objects play the rôle of **function objects**, also known as **functors** — in the standard library they can be used in contexts where a function is expected. As the matter of fact, function objects are more universal than normal functions. One of the reasons is that when we create them, we can pass additional information to their constructors; this information can then be used by the method **operator()**. Therefore, object functions are more flexible. Additionally, they are often more efficient, because they can be inlined, while if a normal function is passed to another function by pointer, the compiler cannot inline it.

Let us consider the following program:

P214: compar.cpp Function objects

```
1. #include <iostream>
2. #include <cmath>      // sqrt
3. #include <vector>
4. #include <algorithm>  // sort, copy
5. #include <iterator>
6. using namespace std;
7.
8. struct Comp {
9.    enum Methods {
10.       by_sum_of_digits,
11.       by_num_of_divisors,
12.       by_value_asc,
13.       by_value_desc
14.    };
15.
16.    Comp(Methods method): method(method) { }
17.
18.    bool operator()(int n1, int n2);
19.
20.    class NoComparator { };
21.
22. private:
23.    Methods method;
24.
25.    static int sum_of_digits(int n);
```

```
26.     static int numb_divs(int n);
27. };
28.
29. bool Comp::operator()(int n1, int n2) {
30.     switch (method) {
31.         case by_sum_of_digits:
32.             return sum_of_digits(n1) < sum_of_digits(n2);
33.         case by_num_of_divisors:
34.             return numb_divs(n1) < numb_divs(n2);
35.         case by_value_asc:
36.             return n1 < n2;
37.         case by_value_desc:
38.             return n2 < n1;
39.         default:
40.             throw NoComparator();
41.                         // should never happen
42.     }
43. }
44.
45. int Comp::sum_of_digits(int n) {
46.     // sum of digits of an integral number
47.     // (all taken with plus sign)
48.     n = n >= 0 ? n : -n;
49.     int s = 0;
50.     while (n) { s += n%10; n /= 10; }
51.     return s;
52. }
53.
54. int Comp::numb_divs(int n) {
55.     // number of positive divisors of an integral
56.     // number (including 1 and the number itself)
57.     n = n > 0 ? n : -n;
58.     if ( n < 3 ) return n;
59.     int sr = (int) sqrt(n+0.5);
60.     int cnt = sr*sr == n ? 1 : 2;
61.     for (int i = 2; i <= sr; ++i) if (n%i == 0) cnt += 2;
62.     return cnt;
63. }
64.
65. int main() {
66.     int tab[] = {7, 4, 8, 12, 13, 119, 16, 6};
67.     vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
68.     v.push_back(64);
69.
70.      // sorting with different methods and displaying results
71.
72.     cout <<   "Sorting by sum of digits\n";
73.     sort(v.begin(),v.end(),Comp(Comp::by_sum_of_digits));
74.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
75.
76.     cout << "\nSorting by number of divisors\n";
77.     sort(v.begin(),v.end(),Comp(Comp::by_num_of_divisors));
78.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
79.
80.
81.     cout << "\nSorting by value in asc. order\n";
82.     sort(v.begin(),v.end(),Comp(Comp::by_value_asc));
83.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
84.
85.     cout << "\nSorting by value in desc. order\n";
86.     sort(v.begin(),v.end(),Comp(Comp::by_value_desc));
87.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
88.
89.     cout << endl;
90. }
```

The program sorts a vector of integers using various sorting criteria. This time, the rôle of a comparator is played not by a function, but rather by an object of class **Comp**. The class defines method **operator()** (lines 18 and 29-43). Its only constructor requires an argument of enumeration type, which then specifies a method of sorting. The value of this enumeration type is held in objects's member ***method***.

In the **main** function we call **sort** passing as the third argument an anonymous object of class **Komp**. Therefore, the object created remembers the required criterion of sorting which has been passed as the argument. The sorting algorithm will invoke **operator()** with elements of the collection as arguments; as this is an object, however, it stores the information on the method of sorting requested in its member ***method*** whose value selects an appropriate branch of the **switch** statement (line 30). The printout is:

```
Sorting by sum of digits
12 4 13 6 7 16 8 64 119
Sorting by number of divisors
13 7 4 6 8 119 16 12 64
Sorting by value in asc. order
4 6 7 8 12 13 16 64 119
Sorting by value in desc. order
119 64 16 13 12 8 7 6 4
```

Function objects can also be used for creating our own manipulators. In section on manipulators we described manipulators with arguments. Several such manipulators are already defined in the standard library and available through the header `iomanip`, but how can we define our own manipulators? Let us look at an example:

P215: manip.cpp Manipulators as function objects

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. struct maniparg {
6.     string str;
7.     maniparg(int cnt, char c) : str(cnt,c) { }
8.     ostream& operator()(ostream& s) const {
9.         return s << str;
10.    }
11. };
12.
13. ostream& operator<<(ostream& s, const maniparg& manip) {
14.     return manip(s);
15. }
16.
17. int main() {
18.     cout << maniparg(7,'*') << "This is maniparg"
19.          << maniparg(3,'!') << maniparg(7,'*') << endl;
20.
21.     maniparg threeexcls(3,'!');
22.     maniparg sevenstars(7,'*');
23.
24.     cout << sevenstars <<  "This is maniparg"
25.          << threeexcls << sevenstars  << endl;
26. }
```

We create class **maniparg** and, in the usual way, we overload operator '<<' for this class (lines 13-15). This function will be invoked when an object of class **maniparg** appears on the right-hand side of operator '<<'. The stream and the object will be passed as arguments. The overloading function calls the object (line 14), i.e., method **operator()** from class **maniparg** (lines 8-10), and returns the obtained value. Method **operator()** gets an output stream through argument, prints the string (constructed from ***cnt*** repetitions of character ***c***) and returns the same string it got. In this way, the function overloading '<<' returns the same string as well, exactly as it should. Why is this construct useful?

Creating the object we can pass any information through a constructor. In our example we pass a character and number of repetitions and this information is held in the object as a string member ***str***. This member is then used by method **operator()** invoked from the function overloading operator '<<'. This overloading function in turn passes information about the stream, so the method has complete information: the string to be printed and the stream to print it into. The program prints

```
*******This is maniparg!!!*******
*******This is maniparg!!!*******
```

Note that equally well we can create in advance named objects of type **maniparg** and then use them as many times and as we wish, as we do with objects ***threeexcls*** and ***sevenstars*** (lines 24-25).

Prev: 24.1 Collections and iterators **Up:** 24. The Standard Library (STL) **Next:** 24.3 Examples

Subsections

- 24.2.1 Algorithms
- 24.2.2 Function objects

24.2 Algorithms and function objects

The standard library provides more than 100 **algorithms**, , i.e., templates of functions operating on collections. Collections as such are never passed as arguments to the algorithms: instead, we always use iterators. Many algorithms take also functions (or function objects) as additional arguments, as we will see below.

24.2.1 Algorithms

We will briefly discuss a few of many algorithms provided by the standard library. They are accessible after including the header `algorithm`.

Sorting algorithms belong to the most useful. Their use is quite simple:

P209: sort.cpp Sorting

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    vector<string> vec{"Paris", "London", "Warsaw",
                       "Berlin", "Lisbon", "Oslo"};
    auto ini = vec.begin(), fin = vec.end();

    list<string> lis(vec.size()-2);                ❶
    copy(ini+1, fin-1, lis.begin());               ❷

    sort(ini,fin);                                 ❸
    lis.sort();                                    ❹

    copy(ini, fin,                                 ❺
         ostream_iterator<string>(cout, " "));
    cout << endl;

    copy(lis.begin(),lis.end(),
         ostream_iterator<string>(cout, " "));
    cout << endl;
}
```

We create a vector of strings. Then we create a list (❶) with initial size by two smaller than the size of the vector. Algorithm **copy** (❷) takes a range (as a pair of iterators) of elements from the source container and iterator of the target container indicating where elements are to be copied. Note that elements are *copied* onto already existing elements of the target container — the algorithm does not create new elements, only overwrites them. That's why we had to ensure that the initial size of the list is sufficient for all the elements from the vector that we want to copy.
We then sort the vector using the algorithm **sort** (❸) — as usual, we pass the range to be sorted as a pair of iterators. Note that **sort** algorithm expects random access iterators. That's why we cannot use it for a list — lists, however, have the *method* **sort** (see ❹). The same mechanism works for other algorithms which require random access iterators; collections providing only bidirectional iterators have corresponding methods defined (e.g., **merge**, **remove**, **reverse** etc.).

Function **copy** was also used in line ❺. This time the third argument is an iterator to a slightly bizarre target collection. This is an iterator created from template **ostream_iterator** (from header `iterator`) by its concretization for type **string**. The constructor of this concretization takes two arguments: an output stream, in our case ***cout***, and a string which will be used as the separator, a space in the example. In this way an output stream can be treated as a collection of elements to be printed (all elements must be of the same type). Iterators of such collections are of type `output iterator' — they are one-directional, single pass iterators which allow only for writing elements to the stream (there are also analogous input iterators). Our example program prints

```
Berlin Lisbon London Oslo Paris Warsaw
Berlin Lisbon London Warsaw
```

As we noted, the **copy** function copies elements from one collection to another overwriting existing elements. That is why, in the program above, we had to create the list (❶) with the appropriate size. This is sometimes not so simple, because very often we don't know how many elements we will copy. In such situations, we can use a special function **back_inserter** which returns an object which behaves like an output iterator, but adds new elements of the collection (by calling **push_back**). For example, in the following program, we copy elements form one vector to another (which is initially empty), but only those which meet some requirement specified by a predicate passed as the last argument to the **copy_if** function:

P210: backins.cpp *back_inserter function*

```
1.  #include <algorithm>    // copy
2.  #include <iostream>
3.  #include <iterator>     // inserters
4.  #include <vector>
5.
6.  int main() {
7.      std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
8.      std::vector<int> emp;
9.      std::copy_if(v.begin(), v.end(),
10.                  std::back_inserter(emp),
11.                  [](auto n){ return n%2 != 0; });
12.     std::copy(emp.begin(), emp.end(),
13.               std::ostream_iterator<int>(std::cout, "\n"));
14. }
```

(there are also similar functions **front_inserter** and **inserter**).

Passing functions to algorithms is rather common. For example, sorting algorithms can take additional argument which determines the way in which elements of a collection are to be compared (normally they are compared using the < operator). This should be a pointer to a function, a lambda or a function object (anything that behaves like a function) and which plays the rôle of a **comparator**, i.e., a function taking two arguments of type corresponding to the type of elements of the collection, and returning a **bool** value which answers the question *is the first argument strictly less than the second?*
For example, in the following program we use comparator **compar** (❶) which states that any odd number is less than any even number, and numbers of the same parity are ordered in the natural way (ascending order). This comparator is used in line ❹ as a function pointer.
We also have a simple structure **Evens** (❷) with overloaded function call operator: objects of these class can also be used as comparators (❺) — this particular comparator considers all even numbers as smaller than odd numbers, and for arguments of the same parity grater values are considered `smaller' (descending order).
Finally, we can also pass a lambda (❻) — it considers greater values as `smaller' so we will get the descending order.
The function (template) **printVec** (❸) just print a vector.

P211: sortev.cpp Comparators

```
1.  #include <iostream>
2.  #include <vector>
3.  #include <algorithm>
4.  #include <iterator>
5.
6.  bool compar(int a, int b) {                          ❶
7.      return (a+b)%2 != 0 ? a%2 != 0 : a < b;
8.  }
9.
10. struct Evens {                                       ❷
11.     bool operator()(int a, int b) {
12.         return (a+b)%2 != 0 ? a%2 == 0 : b < a;
13.     }
14. };
15.
16. template <typename T>
17. void printVec(const std::vector<T>& vec) {           ❸
18.     copy(vec.cbegin(), vec.cend(),
19.         std::ostream_iterator<T>(std::cout, " "));
20.     std::cout << '\n';
21. }
22.
23. int main() {
24.     std::vector<int> vec{2, 5, 2, 9, 1, 5, 7, 4};
25.     printVec(vec);
26.
27.     sort(vec.begin(), vec.end(), compar);            ❹
28.     printVec(vec);
29.
30.     sort(vec.begin(), vec.end(), Evens{});           ❺
31.     printVec(vec);
32.
33.     sort(vec.begin(), vec.end(),
34.          [](int a, int b) {return b < a;});          ❻
35.     printVec(vec);
36.
37. }
```

The program prints:

```
2 5 2 9 1 5 7 4
1 5 5 7 9 2 2 4
4 2 2 9 7 5 5 1
9 7 5 5 4 2 2 1
```

Generally, functions returning **bool** values are called **predicates**. They are extensively used by many algorithms, not only sorting ones. For example, function (strictly speaking, a template) **count_if** counts how many elements from a range of a collection defined by two iterators yield **true** when passed as the only argument to a predicate. In the example below, predicate **even** (line 21) returns **true** only for even numbers, so the algorithm will count how many elements in the vector are even:

P212: predic.cpp Predicates

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. #include <iterator>
5. using namespace std;
6.
7. bool even(int a) {
8.     return (a&1) == 0;
9. }
10.
11. int main() {
12.     vector<int> vec;
13.     int d;
14.     while ( cin >> d ) vec.push_back(d);
15.     cin.clear();
16.
17.     cout << "In the series ";
18.     copy(vec.begin(),vec.end(),
19.          ostream_iterator<int>(cout, " "));
20.     cout << "\nthere are "
21.          << count_if(vec.begin(), vec.end(), even)
22.          << " even numbers\n";
23. }
```

The program prints:

```
2 4 -2 7 5 4 31 4 5 -4
^D
In the series 2 4 -2 7 5 4 31 4 5 -4
there are 6 even numbers
```

The standard library contains also many searching algorithms, as, for example, **find** and **find_if**. Function **find** searches for the first occurence of element which is equal to an object passed as argument, while **find_if** searches for the first occurence of element for which a predicate passed as arguments returns **true**. As always, collection is defined by a pair of iterators. If the search succeeds, the iterator to the element found is returned; otherwise the functions returns the iterator **end()**. Examples of these (and others) algorithms can be found in the program below:

P213: search.cpp Searching

```
1. #include <vector>
```

```
2.  #include <iostream>
3.  #include <string>
4.  #include <cctype>    // tolower
5.  #include <algorithm>
6.  #include <iterator>
7.  using namespace std;
8.
9.  bool startsWithA(string& s) {
10.     return s[0] == 'A';
11. }
12.
13. void lowerc(string& name) {
14.     name[0] = tolower(name[0]);
15. }
16.
17. int main() {
18.     vector<string> vs;
19.
20.     vs.push_back("Maggie"); vs.push_back("Ann");
21.     vs.push_back("Monica"); vs.push_back("Agatha");
22.     vs.push_back("Alice");  vs.push_back("Ursula");
23.
24.     vector<string>::iterator k;
25.
26.     k = find(vs.begin(), vs.end(), "Ann");
27.     if ( k != vs.end() )
28.         cout << *k << " found\n";
29.     else
30.         cout << "Ann not found\n";
31.
32.     k = find(vs.begin(), vs.end(), "Kate");
33.     if ( k != vs.end() )
34.         cout << *k << " found\n";
35.     else
36.         cout << "Kate not found\n";
37.
38.
39.     cout << "\nAmong names\n";
40.     copy(vs.begin(),vs.end(),
41.          ostream_iterator<string>(cout," "));
42.     cout << "\nthe following start with \'A\':\n";
43.     k = vs.begin();
44.     while ( k < vs.end() ) {
45.         k = find_if(k, vs.end(), startsWithA);
46.         if ( k != vs.end() ) cout << *k++ << " ";
47.     }
48.     cout << endl;
49.
50.     for_each(vs.begin(),vs.end(),lowerc);
51.     cout << "\nAfter changing to lower case:\n";
52.     copy(vs.begin(),vs.end(),
53.          ostream_iterator<string>(cout," "));
54.     cout << endl;
55. }
```

We use the function **find** on lines 26 and 32. Then we check if the iterator returned is not equal to ***vs.end()***, what would mean that the search failed.

In the next part of the program we select from the vector of names those which start with the letter 'A' (loop on lines 44-47). To achieve that, we use **find_if** with predicate **startsWithA** (which is defined on lines 9-11).

Line 50 illustrates algorithm **for_each**. This is an example of **mutating algorithm**, i.e., an algorithm which modifies elements of a collection (searching algorithms did not). Function **for_each** scans all elements from a given range of a collection, and for each of them calls a user defined function (or function object). Elements of the collection are passed by reference, so the function can modify them. Return values, even if they exist, are ignored. In our example, the function (**lowerc**) changes the first letter of passed strings to lower

case. The printout is as follows:

```
Ann found
Kate not found

Among names
Maggie Ann Monica Agatha Alice Ursula
the following start with 'A':
Ann Agatha Alice

After changing to lower case:
maggie ann monica agatha alice ursula
```

24.2.2 Function objects

As we have seen, user defined functions, e.g., predicates, play very important rôle in various algorithms from the standard library. Actually, these are not necessarily functions; they can be “anything that can be called”. As we remember from the chapter on operator overloading , in any class one can overload the function-invocation operator — **operator()**. Then, if the name of an object is followed by round parentheses with appropriate arguments, this function will be called for the object and arguments will be passed as for any other method. Objects of classes which define method **operator()** are called **callable objects**. Callable objects play the rôle of **function objects**, also known as **functors** — in the standard library they can be used in contexts where a function is expected. As the matter of fact, function objects are more universal than normal functions. One of the reasons is that when we create them, we can pass additional information to their constructors; this information can then be used by the method **operator()**. Therefore, object functions are more flexible. Additionally, they are often more efficient, because they can be inlined, while if a normal function is passed to another function by pointer, the compiler cannot inline it.

Let us consider the following program:

P214: compar.cpp Function objects

```
1. #include <iostream>
2. #include <cmath>      // sqrt
3. #include <vector>
4. #include <algorithm>  // sort, copy
5. #include <iterator>
6. using namespace std;
7.
8. struct Comp {
9.    enum Methods {
10.       by_sum_of_digits,
11.       by_num_of_divisors,
12.       by_value_asc,
13.       by_value_desc
14.    };
15.
16.    Comp(Methods method): method(method) { }
17.
18.    bool operator()(int n1, int n2);
19.
20.    class NoComparator { };
21.
22. private:
23.    Methods method;
24.
25.    static int sum_of_digits(int n);
```

```
26.     static int numb_divs(int n);
27. };
28.
29. bool Comp::operator()(int n1, int n2) {
30.     switch (method) {
31.         case by_sum_of_digits:
32.             return sum_of_digits(n1) < sum_of_digits(n2);
33.         case by_num_of_divisors:
34.             return numb_divs(n1) < numb_divs(n2);
35.         case by_value_asc:
36.             return n1 < n2;
37.         case by_value_desc:
38.             return n2 < n1;
39.         default:
40.             throw NoComparator();
41.                         // should never happen
42.     }
43. }
44.
45. int Comp::sum_of_digits(int n) {
46.     // sum of digits of an integral number
47.     // (all taken with plus sign)
48.     n = n >= 0 ? n : -n;
49.     int s = 0;
50.     while (n) { s += n%10; n /= 10; }
51.     return s;
52. }
53.
54. int Comp::numb_divs(int n) {
55.     // number of positive divisors of an integral
56.     // number (including 1 and the number itself)
57.     n = n > 0 ? n : -n;
58.     if ( n < 3 ) return n;
59.     int sr = (int) sqrt(n+0.5);
60.     int cnt = sr*sr == n ? 1 : 2;
61.     for (int i = 2; i <= sr; ++i) if (n%i == 0) cnt += 2;
62.     return cnt;
63. }
64.
65. int main() {
66.     int tab[] = {7, 4, 8, 12, 13, 119, 16, 6};
67.     vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
68.     v.push_back(64);
69.
70.      // sorting with different methods and displaying results
71.
72.     cout <<   "Sorting by sum of digits\n";
73.     sort(v.begin(),v.end(),Comp(Comp::by_sum_of_digits));
74.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
75.
76.     cout << "\nSorting by number of divisors\n";
77.     sort(v.begin(),v.end(),Comp(Comp::by_num_of_divisors));
78.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
79.
80.
81.     cout << "\nSorting by value in asc. order\n";
82.     sort(v.begin(),v.end(),Comp(Comp::by_value_asc));
83.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
84.
85.     cout << "\nSorting by value in desc. order\n";
86.     sort(v.begin(),v.end(),Comp(Comp::by_value_desc));
87.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
88.
89.     cout << endl;
90. }
```

The program sorts a vector of integers using various sorting criteria. This time, the rôle of a comparator is played not by a function, but rather by an object of class **Comp**. The class defines method **operator()** (lines 18 and 29-43). Its only constructor requires an argument of enumeration type, which then specifies a method of sorting. The value of this enumeration type is held in objects's member ***method***.

In the **main** function we call **sort** passing as the third argument an anonymous object of class **Komp**. Therefore, the object created remembers the required criterion of sorting which has been passed as the argument. The sorting algorithm will invoke **operator()** with elements of the collection as arguments; as this is an object, however, it stores the information on the method of sorting requested in its member ***method*** whose value selects an appropriate branch of the **switch** statement (line 30). The printout is:

```
Sorting by sum of digits
12 4 13 6 7 16 8 64 119
Sorting by number of divisors
13 7 4 6 8 119 16 12 64
Sorting by value in asc. order
4 6 7 8 12 13 16 64 119
Sorting by value in desc. order
119 64 16 13 12 8 7 6 4
```

Function objects can also be used for creating our own manipulators. In section on manipulators we described manipulators with arguments. Several such manipulators are already defined in the standard library and available through the header `iomanip`, but how can we define our own manipulators? Let us look at an example:

P215: manip.cpp Manipulators as function objects

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. struct maniparg {
6.     string str;
7.     maniparg(int cnt, char c) : str(cnt,c) { }
8.     ostream& operator()(ostream& s) const {
9.         return s << str;
10.    }
11. };
12.
13. ostream& operator<<(ostream& s, const maniparg& manip) {
14.     return manip(s);
15. }
16.
17. int main() {
18.     cout << maniparg(7,'*') << "This is maniparg"
19.          << maniparg(3,'!') << maniparg(7,'*') << endl;
20.
21.     maniparg threeexcls(3,'!');
22.     maniparg sevenstars(7,'*');
23.
24.     cout << sevenstars <<  "This is maniparg"
25.          << threeexcls << sevenstars  << endl;
26. }
```

We create class **maniparg** and, in the usual way, we overload operator '<<' for this class (lines 13-15). This function will be invoked when an object of class **maniparg** appears on the right-hand side of operator '<<'. The stream and the object will be passed as arguments. The overloading function calls the object (line 14), i.e., method **operator()** from class **maniparg** (lines 8-10), and returns the obtained value. Method **operator()** gets an output stream through argument, prints the string (constructed from ***cnt*** repetitions of character ***c***) and returns the same string it got. In this way, the function overloading '<<' returns the same string as well, exactly as it should. Why is this construct useful?

Creating the object we can pass any information through a constructor. In our example we pass a character and number of repetitions and this information is held in the object as a string member ***str***. This member is then used by method **operator()** invoked from the function overloading operator '<<'. This overloading function in turn passes information about the stream, so the method has complete information: the string to be printed and the stream to print it into. The program prints

```
*******This is maniparg!!!*******
*******This is maniparg!!!*******
```

Note that equally well we can create in advance named objects of type **maniparg** and then use them as many times and as we wish, as we do with objects ***threeexcls*** and ***sevenstars*** (lines 24-25).

Prev: 24.1 Collections and iterators **Up:** 24. The Standard Library (STL) **Next:** 24.3 Examples

Subsections

- 24.2.1 Algorithms
- 24.2.2 Function objects

24.2 Algorithms and function objects

The standard library provides more than 100 **algorithms**, , i.e., templates of functions operating on collections. Collections as such are never passed as arguments to the algorithms: instead, we always use iterators. Many algorithms take also functions (or function objects) as additional arguments, as we will see below.

24.2.1 Algorithms

We will briefly discuss a few of many algorithms provided by the standard library. They are accessible after including the header `algorithm`.

Sorting algorithms belong to the most useful. Their use is quite simple:

P209: sort.cpp Sorting

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    vector<string> vec{"Paris", "London", "Warsaw",
                       "Berlin", "Lisbon", "Oslo"};
    auto ini = vec.begin(), fin = vec.end();

    list<string> lis(vec.size()-2);                ❶
    copy(ini+1, fin-1, lis.begin());               ❷

    sort(ini,fin);                                 ❸
    lis.sort();                                    ❹

    copy(ini, fin,                                 ❺
         ostream_iterator<string>(cout, " "));
    cout << endl;

    copy(lis.begin(),lis.end(),
         ostream_iterator<string>(cout, " "));
    cout << endl;
}
```

We create a vector of strings. Then we create a list (❶) with initial size by two smaller than the size of the vector. Algorithm **copy** (❷) takes a range (as a pair of iterators) of elements from the source container and iterator of the target container indicating where elements are to be copied. Note that elements are *copied* onto already existing elements of the target container — the algorithm does not create new elements, only overwrites them. That's why we had to ensure that the initial size of the list is sufficient for all the elements from the vector that we want to copy.
We then sort the vector using the algorithm **sort** (❸) — as usual, we pass the range to be sorted as a pair of iterators. Note that **sort** algorithm expects random access iterators. That's why we cannot use it for a list — lists, however, have the *method* **sort** (see ❹). The same mechanism works for other algorithms which require random access iterators; collections providing only bidirectional iterators have corresponding methods defined (e.g., **merge**, **remove**, **reverse** etc.).

Function **copy** was also used in line ❺. This time the third argument is an iterator to a slightly bizarre target collection. This is an iterator created from template **ostream_iterator** (from header `iterator`) by its concretization for type **string**. The constructor of this concretization takes two arguments: an output stream, in our case ***cout***, and a string which will be used as the separator, a space in the example. In this way an output stream can be treated as a collection of elements to be printed (all elements must be of the same type). Iterators of such collections are of type `output iterator' — they are one-directional, single pass iterators which allow only for writing elements to the stream (there are also analogous input iterators). Our example program prints

```
Berlin Lisbon London Oslo Paris Warsaw
Berlin Lisbon London Warsaw
```

As we noted, the **copy** function copies elements from one collection to another overwriting existing elements. That is why, in the program above, we had to create the list (❶) with the appropriate size. This is sometimes not so simple, because very often we don't know how many elements we will copy. In such situations, we can use a special function **back_inserter** which returns an object which behaves like an output iterator, but adds new elements of the collection (by calling **push_back**). For example, in the following program, we copy elements form one vector to another (which is initially empty), but only those which meet some requirement specified by a predicate passed as the last argument to the **copy_if** function:

P210: backins.cpp *back_inserter function*

```
1.  #include <algorithm>    // copy
2.  #include <iostream>
3.  #include <iterator>     // inserters
4.  #include <vector>
5.
6.  int main() {
7.      std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
8.      std::vector<int> emp;
9.      std::copy_if(v.begin(), v.end(),
10.                  std::back_inserter(emp),
11.                  [](auto n){ return n%2 != 0; });
12.     std::copy(emp.begin(), emp.end(),
13.               std::ostream_iterator<int>(std::cout, "\n"));
14. }
```

(there are also similar functions **front_inserter** and **inserter**).

Passing functions to algorithms is rather common. For example, sorting algorithms can take additional argument which determines the way in which elements of a collection are to be compared (normally they are compared using the < operator). This should be a pointer to a function, a lambda or a function object (anything that behaves like a function) and which plays the rôle of a **comparator**, i.e., a function taking two arguments of type corresponding to the type of elements of the collection, and returning a **bool** value which answers the question *is the first argument strictly less than the second?*
For example, in the following program we use comparator **compar** (❶) which states that any odd number is less than any even number, and numbers of the same parity are ordered in the natural way (ascending order). This comparator is used in line ❹ as a function pointer.
We also have a simple structure **Evens** (❷) with overloaded function call operator: objects of these class can also be used as comparators (❺) — this particular comparator considers all even numbers as smaller than odd numbers, and for arguments of the same parity grater values are considered `smaller' (descending order).
Finally, we can also pass a lambda (❻) — it considers greater values as `smaller' so we will get the descending order.
The function (template) **printVec** (❸) just print a vector.

P211: sortev.cpp Comparators

```
1.  #include <iostream>
2.  #include <vector>
3.  #include <algorithm>
4.  #include <iterator>
5.
6.  bool compar(int a, int b) {                          ❶
7.      return (a+b)%2 != 0 ? a%2 != 0 : a < b;
8.  }
9.
10. struct Evens {                                       ❷
11.     bool operator()(int a, int b) {
12.         return (a+b)%2 != 0 ? a%2 == 0 : b < a;
13.     }
14. };
15.
16. template <typename T>
17. void printVec(const std::vector<T>& vec) {           ❸
18.     copy(vec.cbegin(), vec.cend(),
19.         std::ostream_iterator<T>(std::cout, " "));
20.     std::cout << '\n';
21. }
22.
23. int main() {
24.     std::vector<int> vec{2, 5, 2, 9, 1, 5, 7, 4};
25.     printVec(vec);
26.
27.     sort(vec.begin(), vec.end(), compar);            ❹
28.     printVec(vec);
29.
30.     sort(vec.begin(), vec.end(), Evens{});           ❺
31.     printVec(vec);
32.
33.     sort(vec.begin(), vec.end(),
34.          [](int a, int b) {return b < a;});          ❻
35.     printVec(vec);
36.
37. }
```

The program prints:

```
2 5 2 9 1 5 7 4
1 5 5 7 9 2 2 4
4 2 2 9 7 5 5 1
9 7 5 5 4 2 2 1
```

Generally, functions returning **bool** values are called **predicates**. They are extensively used by many algorithms, not only sorting ones. For example, function (strictly speaking, a template) **count_if** counts how many elements from a range of a collection defined by two iterators yield **true** when passed as the only argument to a predicate. In the example below, predicate **even** (line 21) returns **true** only for even numbers, so the algorithm will count how many elements in the vector are even:

P212: predic.cpp Predicates

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. #include <iterator>
5. using namespace std;
6.
7. bool even(int a) {
8.     return (a&1) == 0;
9. }
10.
11. int main() {
12.     vector<int> vec;
13.     int d;
14.     while ( cin >> d ) vec.push_back(d);
15.     cin.clear();
16.
17.     cout << "In the series ";
18.     copy(vec.begin(),vec.end(),
19.          ostream_iterator<int>(cout, " "));
20.     cout << "\nthere are "
21.          << count_if(vec.begin(), vec.end(), even)
22.          << " even numbers\n";
23. }
```

The program prints:

```
2 4 -2 7 5 4 31 4 5 -4
^D
In the series 2 4 -2 7 5 4 31 4 5 -4
there are 6 even numbers
```

The standard library contains also many searching algorithms, as, for example, **find** and **find_if**. Function **find** searches for the first occurence of element which is equal to an object passed as argument, while **find_if** searches for the first occurence of element for which a predicate passed as arguments returns **true**. As always, collection is defined by a pair of iterators. If the search succeeds, the iterator to the element found is returned; otherwise the functions returns the iterator **end()**. Examples of these (and others) algorithms can be found in the program below:

P213: search.cpp Searching

```
1. #include <vector>
```

```
2.  #include <iostream>
3.  #include <string>
4.  #include <cctype>    // tolower
5.  #include <algorithm>
6.  #include <iterator>
7.  using namespace std;
8.
9.  bool startsWithA(string& s) {
10.     return s[0] == 'A';
11. }
12.
13. void lowerc(string& name) {
14.     name[0] = tolower(name[0]);
15. }
16.
17. int main() {
18.     vector<string> vs;
19.
20.     vs.push_back("Maggie"); vs.push_back("Ann");
21.     vs.push_back("Monica"); vs.push_back("Agatha");
22.     vs.push_back("Alice");  vs.push_back("Ursula");
23.
24.     vector<string>::iterator k;
25.
26.     k = find(vs.begin(), vs.end(), "Ann");
27.     if ( k != vs.end() )
28.         cout << *k << " found\n";
29.     else
30.         cout << "Ann not found\n";
31.
32.     k = find(vs.begin(), vs.end(), "Kate");
33.     if ( k != vs.end() )
34.         cout << *k << " found\n";
35.     else
36.         cout << "Kate not found\n";
37.
38.
39.     cout << "\nAmong names\n";
40.     copy(vs.begin(),vs.end(),
41.          ostream_iterator<string>(cout," "));
42.     cout << "\nthe following start with \'A\':\n";
43.     k = vs.begin();
44.     while ( k < vs.end() ) {
45.         k = find_if(k, vs.end(), startsWithA);
46.         if ( k != vs.end() ) cout << *k++ << " ";
47.     }
48.     cout << endl;
49.
50.     for_each(vs.begin(),vs.end(),lowerc);
51.     cout << "\nAfter changing to lower case:\n";
52.     copy(vs.begin(),vs.end(),
53.          ostream_iterator<string>(cout," "));
54.     cout << endl;
55. }
```

We use the function **find** on lines 26 and 32. Then we check if the iterator returned is not equal to ***vs.end()***, what would mean that the search failed.

In the next part of the program we select from the vector of names those which start with the letter 'A' (loop on lines 44-47). To achieve that, we use **find_if** with predicate **startsWithA** (which is defined on lines 9-11).

Line 50 illustrates algorithm **for_each**. This is an example of **mutating algorithm**, i.e., an algorithm which modifies elements of a collection (searching algorithms did not). Function **for_each** scans all elements from a given range of a collection, and for each of them calls a user defined function (or function object). Elements of the collection are passed by reference, so the function can modify them. Return values, even if they exist, are ignored. In our example, the function (**lowerc**) changes the first letter of passed strings to lower

case. The printout is as follows:

```
Ann found
Kate not found

Among names
Maggie Ann Monica Agatha Alice Ursula
the following start with 'A':
Ann Agatha Alice

After changing to lower case:
maggie ann monica agatha alice ursula
```

24.2.2 Function objects

As we have seen, user defined functions, e.g., predicates, play very important rôle in various algorithms from the standard library. Actually, these are not necessarily functions; they can be “anything that can be called”. As we remember from the chapter on operator overloading , in any class one can overload the function-invocation operator — **operator()**. Then, if the name of an object is followed by round parentheses with appropriate arguments, this function will be called for the object and arguments will be passed as for any other method. Objects of classes which define method **operator()** are called **callable objects**. Callable objects play the rôle of **function objects**, also known as **functors** — in the standard library they can be used in contexts where a function is expected. As the matter of fact, function objects are more universal than normal functions. One of the reasons is that when we create them, we can pass additional information to their constructors; this information can then be used by the method **operator()**. Therefore, object functions are more flexible. Additionally, they are often more efficient, because they can be inlined, while if a normal function is passed to another function by pointer, the compiler cannot inline it.

Let us consider the following program:

P214: compar.cpp Function objects

```
1. #include <iostream>
2. #include <cmath>      // sqrt
3. #include <vector>
4. #include <algorithm>  // sort, copy
5. #include <iterator>
6. using namespace std;
7.
8. struct Comp {
9.    enum Methods {
10.       by_sum_of_digits,
11.       by_num_of_divisors,
12.       by_value_asc,
13.       by_value_desc
14.    };
15.
16.    Comp(Methods method): method(method) { }
17.
18.    bool operator()(int n1, int n2);
19.
20.    class NoComparator { };
21.
22. private:
23.    Methods method;
24.
25.    static int sum_of_digits(int n);
```

```
26.     static int numb_divs(int n);
27. };
28.
29. bool Comp::operator()(int n1, int n2) {
30.     switch (method) {
31.         case by_sum_of_digits:
32.             return sum_of_digits(n1) < sum_of_digits(n2);
33.         case by_num_of_divisors:
34.             return numb_divs(n1) < numb_divs(n2);
35.         case by_value_asc:
36.             return n1 < n2;
37.         case by_value_desc:
38.             return n2 < n1;
39.         default:
40.             throw NoComparator();
41.                         // should never happen
42.     }
43. }
44.
45. int Comp::sum_of_digits(int n) {
46.     // sum of digits of an integral number
47.     // (all taken with plus sign)
48.     n = n >= 0 ? n : -n;
49.     int s = 0;
50.     while (n) { s += n%10; n /= 10; }
51.     return s;
52. }
53.
54. int Comp::numb_divs(int n) {
55.     // number of positive divisors of an integral
56.     // number (including 1 and the number itself)
57.     n = n > 0 ? n : -n;
58.     if ( n < 3 ) return n;
59.     int sr = (int) sqrt(n+0.5);
60.     int cnt = sr*sr == n ? 1 : 2;
61.     for (int i = 2; i <= sr; ++i) if (n%i == 0) cnt += 2;
62.     return cnt;
63. }
64.
65. int main() {
66.     int tab[] = {7, 4, 8, 12, 13, 119, 16, 6};
67.     vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
68.     v.push_back(64);
69.
70.      // sorting with different methods and displaying results
71.
72.     cout <<   "Sorting by sum of digits\n";
73.     sort(v.begin(),v.end(),Comp(Comp::by_sum_of_digits));
74.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
75.
76.     cout << "\nSorting by number of divisors\n";
77.     sort(v.begin(),v.end(),Comp(Comp::by_num_of_divisors));
78.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
79.
80.
81.     cout << "\nSorting by value in asc. order\n";
82.     sort(v.begin(),v.end(),Comp(Comp::by_value_asc));
83.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
84.
85.     cout << "\nSorting by value in desc. order\n";
86.     sort(v.begin(),v.end(),Comp(Comp::by_value_desc));
87.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
88.
89.     cout << endl;
90. }
```

The program sorts a vector of integers using various sorting criteria. This time, the rôle of a comparator is played not by a function, but rather by an object of class **Comp**. The class defines method **operator()** (lines 18 and 29-43). Its only constructor requires an argument of enumeration type, which then specifies a method of sorting. The value of this enumeration type is held in objects's member ***method***.

In the **main** function we call **sort** passing as the third argument an anonymous object of class **Komp**. Therefore, the object created remembers the required criterion of sorting which has been passed as the argument. The sorting algorithm will invoke **operator()** with elements of the collection as arguments; as this is an object, however, it stores the information on the method of sorting requested in its member ***method*** whose value selects an appropriate branch of the **switch** statement (line 30). The printout is:

```
Sorting by sum of digits
12 4 13 6 7 16 8 64 119
Sorting by number of divisors
13 7 4 6 8 119 16 12 64
Sorting by value in asc. order
4 6 7 8 12 13 16 64 119
Sorting by value in desc. order
119 64 16 13 12 8 7 6 4
```

Function objects can also be used for creating our own manipulators. In section on manipulators we described manipulators with arguments. Several such manipulators are already defined in the standard library and available through the header `iomanip`, but how can we define our own manipulators? Let us look at an example:

P215: manip.cpp Manipulators as function objects

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. struct maniparg {
6.     string str;
7.     maniparg(int cnt, char c) : str(cnt,c) { }
8.     ostream& operator()(ostream& s) const {
9.         return s << str;
10.    }
11. };
12.
13. ostream& operator<<(ostream& s, const maniparg& manip) {
14.     return manip(s);
15. }
16.
17. int main() {
18.     cout << maniparg(7,'*') << "This is maniparg"
19.          << maniparg(3,'!') << maniparg(7,'*') << endl;
20.
21.     maniparg threeexcls(3,'!');
22.     maniparg sevenstars(7,'*');
23.
24.     cout << sevenstars <<  "This is maniparg"
25.          << threeexcls << sevenstars  << endl;
26. }
```

We create class **maniparg** and, in the usual way, we overload operator '<<' for this class (lines 13-15). This function will be invoked when an object of class **maniparg** appears on the right-hand side of operator '<<'. The stream and the object will be passed as arguments. The overloading function calls the object (line 14), i.e., method **operator()** from class **maniparg** (lines 8-10), and returns the obtained value. Method **operator()** gets an output stream through argument, prints the string (constructed from ***cnt*** repetitions of character ***c***) and returns the same string it got. In this way, the function overloading '<<' returns the same string as well, exactly as it should. Why is this construct useful?

Creating the object we can pass any information through a constructor. In our example we pass a character and number of repetitions and this information is held in the object as a string member ***str***. This member is then used by method **operator()** invoked from the function overloading operator '<<'. This overloading function in turn passes information about the stream, so the method has complete information: the string to be printed and the stream to print it into. The program prints

```
*******This is maniparg!!!*******
*******This is maniparg!!!*******
```

Note that equally well we can create in advance named objects of type **maniparg** and then use them as many times and as we wish, as we do with objects ***threeexcls*** and ***sevenstars*** (lines 24-25).

Prev: 24.1 Collections and iterators **Up:** 24. The Standard Library (STL) **Next:** 24.3 Examples

24.3 Examples

Concluding this short introduction, we will confine ourselves to presenting a couple of examples illustrating some useful features of the standard library.

There is a very useful class (a template of classes, in fact) — **pair** from header `utility`. The template describes simple but useful structures containing a pair of values of different types (built-in or user-defined). If we frequently need pairs of type (**string**, **int**), e.g., to describe name and age of persons, then the appropriate type would be concretization **pair<string,int>**. The structure contains two members named ***first*** (of type **string** in our case) and ***second*** (here of type **int**). The class is implemented in such a way that we do not have to worry about overloading such fundamental operators like '==', '!=' etc. if, of course, they work properly for types of members.

Such a class is created in the following program. Data about persons is read from text file `pairs.dat` containing:

```
Jane  17 Jill  7 Ann 19
Ulla  36 Eve  18
Sue    4
```

After reading (line 46), data is copied to vector whose elements are of type **pair<string,int>**. We then operate on the vector to illustrate algorithms from the standard library and function objects:

P216: pairs.cpp Pairs

```
1. #include <fstream>
2. #include <string>
3. #include <utility>   // pair
4. #include <vector>
5. #include <algorithm> // copy, sort, etc.
6. #include <iostream>
7. using namespace std;
8.
9. typedef pair<string,int> PAIR;
10. typedef vector<PAIR>     VECT;
11. typedef VECT::iterator   VECTIT;
12.
13. template <typename P>
14. class Minor {
15.     int age;
16. public:
17.     Minor(int age) : age(age) { }
18.
19.     bool operator()(const P& p) const {
20.         return p.second < age;
21.     }
22. };
23.
24. template <typename P1, typename P2>
25. ostream& operator<<(ostream& str, const pair<P1,P2>& p) {
```

```
26.     return str << "[" << p.first << "," << p.second << "]";
27. }
28.
29. template <typename P1, typename P2>
30. istream& operator>>(istream& str, pair<P1,P2>& p) {
31.     return str >> p.first >> p.second;
32. }
33.
34. template <typename P>
35. bool comp(const P& p1, const P& p2) {
36.     return p1.second < p2.second;
37. }
38.
39. int main() {
40.     ifstream file("pairs.dat");
41.
42.     PAIR   p;
43.     VECT vec;
44.     VECTIT it, fin;
45.
46.     while (file >> p) vec.push_back(p);
47.
48.     cout << "After reading:\n";
49.     fin=vec.end();
50.     for (it = vec.begin(); it != fin; ++it)
51.         cout << *it << " ";
52.
53.     cout << "\nOldest "
54.          << *max_element(vec.begin(),vec.end(),comp<PAIR>)
55.          << ", youngest "
56.          << *min_element(vec.begin(),vec.end(),comp<PAIR>);
57.
58.     it = remove_if(vec.begin(),fin,
59.                    Minor<PAIR>(18));
60.     vec.erase(it,vec.end());
61.
62.     cout << "\nAfter removing minors:\n";
63.     fin=vec.end();
64.     for (it = vec.begin(); it != fin; ++it)
65.         cout << *it << " ";
66.
67.     sort(vec.begin(),fin,comp<PAIR>);
68.
69.     cout << "\nAfter sorting:\n";
70.     fin=vec.end();
71.     for (it = vec.begin(); it != fin; ++it)
72.         cout << *it << " ";
73.
74.     cout << endl;
75. }
```

Lines 9-11 define aliases for names of types which are then used in the program. The name **PAIR** is a synonym of the name of the type which is concretized from template **pair** with **string** as the type of the member ***first*** and **int** as the type of the member ***second***. Hence, the “true” name of this type is **pair<string,int>**. The name **VECT** (line 10) is an alias of type created by concretization of template **vector** with **pair<string,int>** (i.e., **PAIR**) as the type of its elements. Without **typedef** s we would have to use a rather awkward name **vector<pair<string,int> >**. Note that a space between two ' >' characters is necessary here; otherwise the token '>>' would be interpreted as an operator of extraction from a stream.

We create object ***vec*** of type **VECT** on line 40. This is a vector whose elements are of type **PAIR**. On line 40 we open file input stream ***file*** connected with disk file `pairs.dat` (see section on operations on files). How can we read objects of type **PAIR** from a file? This is a user defined class, so the operator '>>' would not know how to treat object of this type. Therefore, we had to overload this operator; this was done on lines 29-32. The template defined there will work for pairs of values of any two types, as long as these two types are

“printable” (as it is the case for types **string** and **int** used in our example).

Lines 49-51 print elements of the collection ***vec***. They are of type **PAIR**, so we had to overload '<<' operator. This was done on lines 24-27.

Line 54

```
*max_element(vec.begin(),vec.end(),comp<PAIR>)
```

demonstrates algorithm **max_element** which finds the largest element of a collection (defined by two iterators). If the third argument were absent, operator ' <' would be used for comparing elements of the collection. Therefore, we would have to overload this operator for our type **PAIR**. Instead, we use a function (a predicate) which compares the elements — for morte flexibility, the function has been defined in the form of a template (lines 34-37). Note that there is only one parameter of this template (not a pair of types). Consequently, the template will work for any type that has member named ***second*** whose type permits comparisons by ' <' operator. Function **max_element** returns iterator to the maximum element of the collection.

As one can easily guess, algorithm **min_element**, used on line 56, returns iterator to the minimum element of the collection; its use is similar to that of **max_element**. The results of both function can be seen in the printout od the program:

```
After reading:
[Jane,17] [Jill,7] [Ann,19] [Ulla,36] [Eve,18] [Sue,4]
Oldest [Ulla,36], youngest [Sue,4]
After removing minors:
[Ann,19] [Ulla,36] [Eve,18]
After sorting:
[Eve,18] [Ann,19] [Ulla,36]
```

Quite often we want to remove some elements from a collection. For exampel, we want to remove all elements that fulfill a certain condition. This can be effectuated by algorithm **remove_if**, which is used on line 58. Its third element is a function (or a function object) which called with element of the collection answers **true** if, and only if, this element is to be removed. In our case, we use here a function object of class concretizing template **Minor** for type **PAIR**. Creating the object (line 59), we pass to its constructor a number which will be stored in the object as its member ***age*** (see the definition of the template on lines 13-22). We overload the function invocation operator (lines 19-21) in such a way that it returns **true** if the age of the person is less that the value remembered in member ***age***.

Actually, the function **remove_if** does *not* remove anything. It merely relocates “unwanted” elements towards the end of the collection, so when the function exits, all “wanted” elements are before all “unwanted” ones. The function returns the iterator pointing to the first “unwanted” element of the collection (or the “past the end” iterator if there are no elements fulfilling the condition imposed on elements to be removed). We still can remove unwanted elements phsically by invoking **erase** which is a *method* (member function) of the collection (line 60). The method takes two iterators determining a range of elements which are to be erased.

Elements which remained in the collection are sorted on line 67. We call **sort** with the same comparator as that used in functions **max_element** and **min_element**

Another useful type of collection is **map**. Such collections are accessible after including the

header **map**). In order to concretize the template and get a concrete type, one has to pass two types

```
#include <map>
// ...
map<Type1,Type2> aMap;
```

which determine types of the so called keys and values. Individual elements of a map are *pairs* of type **pair<const Type1,Type2>**. Note that the type of keys is modified to become a constant type (**const**), so keys are *unmodifiable* (while values associated with keys *can* be modified). Each key must be unique — there can be no more than one pair with any given key. Maps in C++ correspond roughly to **TreeMap** s in Java, although in C++ both keys and values *can* be of primitive type, like **int** or **double**. Maps (like **TreeMap** s in Java) are implemented as red-black trees. This means that it must be possible to compare keys: for the type of keys the operator '`<`' must work. It work "by itself" for numeric types and **string** s, but if a user-defined type is used as the type of keys, it should overload **operator <** (alternatively, one can pass an appropriate comparator as the third type when concretizing the **map** template).

Every element of a map, being an object of type **pair**, has members ***first*** and ***second*** which correspond to a (key, value) pair. The indexing operator ('`[]`') is overloaded in maps. The value of an expression like '`aMap[key]`' is a reference to member ***second*** of the pair with key (the member ***first***) equal to ***key***. If there is no such a pair, it will be automatically created and the corresponding value (the member ***second***) will be given the default value (zero for numeric types, values provided by default constructor for object types). For example, in the following snippet of code

```
1.    map<string,int> aMap;
2.    aMap["Eve"] = 20;
3.    aMap["Ann"] = ++aMap["Sue"] + 18;
```

line 2 adds to empty map ***aMap*** element with key ***"Eve"*** and value 20. On line 3, element [***"Sue"***, 0] is created, and then its value (zero) is incremented by 1. Hence, after evaluation of expression '`++aMap["Sue"]`', element with key ***"Sue"*** has value 1. The value of the whole right-hand side of the assignment becomes 19.

Evaluation of the left-hand side of the assignment will create a new element [***"Ann"***,0] and then the value of the right-hand side (which is 19) will be assigned to value associated with key ***"Ann"***. After these three lines, the map ***aMap*** will contain three pairs: [***"Eve"***,20], [***"Sue"***,1] and [***"Ann"***,19].

Let us consider another example:

P217: maps.cpp Maps

```
1.  #include <iostream>
2.  #include <iomanip>   // left, setw
3.  #include <string>
4.  #include <map>
5.  #include <utility>   // pair
6.  #include <algorithm>
7.  using namespace std;
8.
9.  using Emp = pair<string,int>;
10. using MAP = map<string,Emp>;
```

```
11.
12. class Range {
13.     int min,max;
14. public:
15.     Range(int min,int max) : min(min), max(max) {}
16.
17.     bool operator()(const pair<const string,Emp>& p) const {
18.         int salary = p.second.second;
19.         return  (min < salary) && (salary < max);
20.     }
21. };
22.
23. void print(const MAP& m) {
24.     for (auto [key, emp] : m) {
25.         auto [name, sal] = emp;
26.         cout << "Key: "    << left << setw(7) << key
27.              << "Name: "   << setw(10) << name
28.              << "Salary: " << sal << endl;
29.     }
30. }
31.
32. int main() {
33.     MAP emp;
34.
35.     emp["sue"]   = Emp("Sue K.",   1900);
36.     emp["jill"]  = Emp("Jill M.",  2100);
37.     emp["eve"]   = Emp("Eve S.",   3100);
38.     emp["boss"]  = Emp("Boss",     9900);
39.     emp["jane"]  = Emp("Jane A.",  1600);
40.     emp["emily"] = Emp("Emily P.", 2600);
41.
42.     print(emp);
43.
44.     int mn = 1100, mx = 2000;
45.     int cnt = count_if(emp.begin(),emp.end(),Range(mn,mx));
46.
47.     cout << cnt << " employees have salary in range "
48.          << mn << " to " << mx << endl;
49. }
```

On lines 9 and 10 we define aliases **Emp** for type **pair<string,int>** and **MAP** for type **map <string,Emp>**. Elements of the maps are pairs with keys of type **const string** and values which are themselves pairs consisting of strings and numbers. An empty map of type **MAP** is created on line 33. Then, on lines 35-40 we add a few elements to the map. For example, on line 35, we create an element with key ***"sue"*** and value which is an object of type **Emp** (a pair of type **pair<string,int>**) with member ***first*** equal to string ***"Sue K."*** and member ***second*** equal to 1900.

The map is then passed to function **print**, which prints all its elements. We could have used `normal' iteration with iterator (say, ***it***) pointing to individual elements of the map, i.e., objects of type **pair<const string, pair<string,int> >**. Hence, '`it->first`' would be a key (e.g., ***"sue"***), and '`it->second`' a pair (**string**, **int**). If we would want to access the value of salary corresponding to an employee represented by an element of the map, we would have to use the syntax '`it->second.second`'. The first selection is by the "arrow" operator, because iterators have semantics of pointers. Epression '`it->second`', however, is not a pointer: it is a reference to object of type **Emp**, so its member ***second*** should be selected by "dot" operator.

However, we don't have to use this syntax here: instead, we can `unpack' pairs using syntax shown in lines 24 and 25. In square brackets we give arbitrary names to elements of the pair, specifying their types by **auto**, as the compiler knows them. In line 24, we unpack elements of the map, so ***key*** corresponds to the key and ***emp*** to the value. This value is itself a **pair**, so we unpack it further to get the name and salary (line 25).

The output of the program could look like this:

```
Key: boss   Name: Boss      Salary: 9900
Key: emily  Name: Emily P.  Salary: 2600
Key: eve    Name: Eve S.    Salary: 3100
Key: jane   Name: Jane A.   Salary: 1600
Key: jill   Name: Jill M.   Salary: 2100
Key: sue    Name: Sue K.    Salary: 1900
2 employees have salary in range 0 to 2000
```

On line 45 we create a callable object of type **Range** passing two numbers (which will define a range) to its constructor. This is a function object which can be called with an element of the map as argument; it can, therefore, play the rôle of a predicate in invocation of function **count_if** from line 45:

```
count_if(emp.begin(),emp.end(),Range(mn,mx));
```

Algorithm **count_if** counts the number of elements of a collection which yield **true** when passed to a predicate given as its third argument. In our case, **operator()** will be called for elements of map ***emp*** and will return **true** if salary of an employee represented by by this element falls into the range defined by members ***min*** and ***max*** of function object ***Range(mn,mx)***.

Note that type **MAP** is really equivalent to **map<string,Emp>**, so its elements will *not* be of type **pair<string,Emp>**, but rather **pair<const string,Emp>** (with **const**). This matches exactly the type of parameter of method **operator()**.

Prev: 24.2 Algorithms and function objects **Up:** 24. The Standard Library (STL) **Next:** 24.4 List of algorithm

24.4 List of algorithm

The standard library provides much more useful tool than those mentioned in this chapter. Its full description can be found in books cited in the bibliography .

Below, we list the algorithms from the standard library without the details — the list may be helpful when we are looking for the algorithm appropriate for a problem we are working on (from en.cppreference.com):

*** **Non-modifying sequence operations** — header `algorithm`

all_of, **any_of**, **none_of** (C++11) checks if a predicate is true for all, any or none of the elements in a range

for_each applies a function to a range of elements

for_each_n (C++17) applies a function object to the first n elements of a sequence

countcount_if returns the number of elements satisfying specific criteria

mismatch finds the first position where two ranges differ

find, **find_if**, **find_if_not** (C++11) finds the first element satisfying specific criteria

find_end finds the last sequence of elements in a certain range

find_first_of searches for any one of a set of elements

adjacent_find finds the first two adjacent items that are equal (or satisfy a given predicate)

search searches for a range of elements

search_n searches a range for a number of consecutive copies of an element

*** **Modifying sequence operations** — header `algorithm`

copy, **copy_if** (C++11) copies a range of elements to a new location

copy_n (C++11) copies a number of elements to a new location

copy_backward copies a range of elements in backwards order

move (C++11) moves a range of elements to a new location

move_backward (C++11) moves a range of elements to a new location in backwards order

fill copy-assigns the given value to every element in a range

fill_n copy-assigns the given value to N elements in a range

transform applies a function to a range of elements

generate assigns the results of successive function calls to every element in a range

generate_n assigns the results of successive function calls to N elements in a range

remove, **remove_if** removes elements satisfying specific criteria

remove_copy, **remove_copy_if** copies a range of elements omitting those that satisfy specific criteria

replace, **replace_if** replaces all values satisfying specific criteria with another value

replace_copy, **replace_copy_if** copies a range, replacing elements satisfying specific criteria with another value

swap swaps the values of two objects

swap_ranges swaps two ranges of elements

iter_swap swaps the elements pointed to by two iterators

reverse reverses the order of elements in a range

reverse_copy creates a copy of a range that is reversed

rotate rotates the order of elements in a range

rotate_copy copies and rotate a range of elements

shift_left, **shift_right** (C++20) shifts elements in a range

shuffle (C++11) randomly re-orders elements in a range

sample (C++17) selects n random elements from a sequence

unique removes consecutive duplicate elements in a range

unique_copy creates a copy of some range of elements that contains no consecutive duplicates

*** **Partitioning operations** — header `algorithm`

is_partitioned (C++11) determines if the range is partitioned by the given predicate

partition divides a range of elements into two groups

partition_copy (C++11) copies a range dividing the elements into two groups

stable_partition divides elements into two groups while preserving their relative order

partition_point (C++11) locates the partition point of a partitioned range

*** **Sorting operations** — header `algorithm`

is_sorted (C++11) checks whether a range is sorted into ascending order

is_sorted_until (C++11) finds the largest sorted subrange

sort sorts a range into ascending order

partial_sort sorts the first N elements of a range

partial_sort_copy copies and partially sorts a range of elements

stable_sort sorts a range of elements while preserving order between equal elements

nth_element partially sorts the given range making sure that it is partitioned by the given element

*** **Binary search operations on sorted ranges** — header `algorithm`

lower_bound returns an iterator to the first element not less than the given value

upper_bound returns an iterator to the first element greater than a certain value

binary_search determines if an element exists in a certain range

equal_range returns range of elements matching a specific key

*** **Other operations on sorted ranges** — header `algorithm`

merge merges two sorted ranges

inplace_merge merges two ordered ranges in-place

*** **Set operations on sorted ranges** — header `algorithm`

includes returns true if one set is a subset of another

set_difference computes the difference between two sets

set_intersection computes the intersection of two sets

set_symmetric_difference computes the symmetric difference between two sets

set_union computes the union of two sets

*** **Heap operations** — header `algorithm`

is_heap (C++11) checks if the given range is a max heap

is_heap_until (C++11) finds the largest subrange that is a max heap

make_heap creates a max heap out of a range of elements

push_heap adds an element to a max heap

pop_heap removes the largest element from a max heap

sort_heap turns a max heap into a range of elements sorted in ascending order

*** **Minimum/maximum operations** — header `algorithm`

max returns the greater of the given values

max_element returns the largest element in a range

min returns the smaller of the given values

min_element returns the smallest element in a range

minmax (C++11) returns the smaller and larger of two elements

minmax_element (C++11) returns the smallest and the largest elements in a range

clamp (C++17) clamps a value between a pair of boundary values

*** **Comparison operations** — header `algorithm`

equal determines if two sets of elements are the same

lexicographical_compare returns true if one range is lexicographically less than another

compare_3way (C++20) compares two values using three-way comparison

lexicographical_compare_3way (C++20) compares two ranges using three-way comparison

*** **Permutation operations** — header `algorithm`

is_permutation (C++11) determines if a sequence is a permutation of another sequence

next_permutation generates the next greater lexicographic permutation of a range of elements

prev_permutation generates the next smaller lexicographic permutation of a range of elements

*** **Numeric operations** — header `numeric`

iota (C++11) fills a range with successive increments of the starting value

accumulate sums up a range of elements

inner_product computes the inner product of two ranges of elements

adjacent_difference computes the differences between adjacent elements in a range

partial_sum computes the partial sum of a range of elements

reduce (C++17) similar to std::accumulate, except out of order

exclusive_scan (C++17) similar to std::partial_sum, excludes the ith input element from the ith sum

inclusive_scan (C++17) similar to std::partial_sum, includes the ith input element in the ith sum

transform_reduce (C++17) applies a functor, then reduces out of order

transform_exclusive_scan (C++17) applies a functor, then calculates exclusive scan

transform_inclusive_scan (C++17) applies a functor, then calculates inclusive scan

*** **Operations on uninitialized memory** — header `memory`

uninitialized_copy copies a range of objects to an uninitialized area of memory

uninitialized_copy_n (C++11) copies a number of objects to an uninitialized area of memory

uninitialized_fill copies an object to an uninitialized area of memory, defined by a range

uninitialized_fill_n copies an object to an uninitialized area of memory, defined by a start and a count

uninitialized_move (C++17) moves a range of objects to an uninitialized area of memory

uninitialized_move_n (C++17) moves a number of objects to an uninitialized area of memory

uninitialized_default_construct (C++17) constructs objects by default-initialization in an uninitialized area of memory, defined by a range

uninitialized_default_construct_n (C++17) constructs objects by default-initialization in an uninitialized area of memory, defined by a start and a count

uninitialized_value_construct (C++17) constructs objects by value-initialization in an uninitialized area of memory, defined by a range

uninitialized_value_construct_n (C++17) constructs objects by value-initialization in an uninitialized area of memory, defined by a start and a count

destroy_at (C++17) destroys an object at a given address

destroy (C++17) destroys a range of objects

destroy_n (C++17) destroys a number of objects in a range

Prev: 24.3 Examples **Up:** 24. The Standard Library (STL) **Next:** 25. Run-time type identification

25. Run-time type identification

As we know, in object-oriented languages there is a distinction between static and dynamic types of variables (see section on polymorphism). Usually, we do not have to worry about it, because the mechanism of polymorphism takes care about it automatically. However, there are situations where we would like to know programmatically what the real type of an object is. This can be done by using the mechanism of **run-time type identification** (RRTI), which can serve two main purposes: comparing true (dynamic) types of two objects, as well as checking validity and performing casts between types from the same class hierarchy. The first of these two tasks is performed by operator **typeid**, while for the casts we use the operator **dynamic_cast**.

The mechanism of RTTI can produce quite substantial overhead that deteriorates the efficiency of code. Therefore, most compilers allow the user to switch it off or on by using appropriate switches: it depends on the compiler whether the RTTI in on or off by default.

Subsections

- 25.1 Operator *typeid*
- 25.2 Operator *dynamic_cast*

25.1 Operator *typeid*

The **typeid** operator (from header `typeinfo`) takes as the argument the name of a type or just a value of a certain type. It returns an object of class **type_info** representing a type. In this class, '`==`' and '`!=`' operators are overloaded providing a possibility of comparing types of objects. For example:

```
#include <typeinfo>
// ...
double x = 1.5;
// ...
if (typeid(x) == typeid(double)) ... // true
if (typeid(x) == typeid(36.0))   ... // true
if (typeid(x) == typeid(3))      ... // false
if (typeid(x) != typeid(3))      ... // true
if (typeid(x) != typeid(int))    ... // true
```

Class **type_info** has method **name** which returns a C-string with the identifier of the type found; the identifier may but does not have to be identical to the standard name of a type. For example

```
typeid(int).name()
```

gives C-string '`int`' in **VC++** environment, but **g++** and Intel's **icpc** return just '`i`'. Usually, we do need these names; all what we care about is to be able to determine it two types are identical or not.

Inheritance makes type identification more interesting and useful. We will demonstrate some details using the following program as an example:

P218: rttid.cpp Dynamic type identification

```
1. #include <iostream>
2. #include <typeinfo>
3. using namespace std;
4.
5. struct Vehicle { };
6. struct Car : Vehicle { };
7.
8. struct Building {
9.     virtual ~Building() { }
10. };
11. struct Station : Building { };
12.
13.
14.
15. int main() {
16.     Vehicle    veh;
17.     Car  car1,car2;
18.     Car* p_car1 = &car1;
19.     Vehicle*   p_car2 = &car2;
20.     Vehicle&   r_car1 = car1;
21.     cout << "    veh: " << typeid(veh).name()      << endl
22.          << "   car1: " << typeid(car1).name()     << endl
23.          << "   car2: " << typeid(car2).name()     << endl
24.          << " p_car1: " << typeid(p_car1).name()   << endl
```

```
25.          << " p_car2: " << typeid(p_car2).name()  << endl
26.          << "*p_car1: " << typeid(*p_car1).name() << endl
27.          << "*p_car2: " << typeid(*p_car2).name() << endl
28.          << " r_car1: " << typeid(r_car1).name()  << endl;
29.
30.     cout << "Types *p_car1 and *p_car2 are "
31.          << (typeid(*p_car1) == typeid(*p_car2) ?
32.                    "the same\n" : "different\n") << endl;
33.
34.     Building  build;
35.     Station   sta1,sta2;
36.     Station*  p_sta1 = &sta1;
37.     Building* p_sta2 = &sta2;
38.     Building& r_sta1 = sta1;
39.     cout << "  build: " << typeid(build).name()   << endl
40.          << "   sta1: " << typeid(sta1).name()    << endl
41.          << "   sta2: " << typeid(sta2).name()    << endl
42.          << " p_sta1: " << typeid(p_sta1).name()  << endl
43.          << " p_sta2: " << typeid(p_sta2).name()  << endl
44.          << "*p_sta1: " << typeid(*p_sta1).name() << endl
45.          << "*p_sta2: " << typeid(*p_sta2).name() << endl
46.          << " r_sta1: " << typeid(r_sta1).name()  << endl;
47.
48.     cout << "Types *p_sta1 and *p_sta2 are "
49.          << (typeid(*p_sta1) == typeid(*p_sta2) ?
50.                    "the same\n" : "different\n");
51. }
```

We have two pairs of classes: **Vehicle** and its subclass **Car**, and analogously **Building** and its subclass **Station**. There is, however, a fundamental difference between these two pairs: **Vehicle** ← **Car** does not contain any virtual functions, so this pair is not polymorphic. On the contrary, **Building** ← **Station** is polymorphic, because the destructor of **Building** is virtual and inheritance is public (what is implicitly implied by using the keyword **struct**, and not **class**). The program prints:

```
     veh: 7Vehicle                          ( 1)
    car1: 3Car                              ( 2)
    car2: 3Car                              ( 3)
  p_car1: P3Car                             ( 4)
  p_car2: P7Vehicle                         ( 5)
 *p_car1: 3Car                              ( 6)
 *p_car2: 7Vehicle                          ( 7)
  r_car1: 7Vehicle                          ( 8)
 Types *p_car1 and *p_car2 are different    ( 9)

   build: 8Building                         (11)
    sta1: 7Station                          (12)
    sta2: 7Station                          (13)
  p_sta1: P7Station                         (14)
  p_sta2: P8Building                        (15)
 *p_sta1: 7Station                          (16)
 *p_sta2: 7Station                          (17)
  r_sta1: 7Station                          (18)
 Types *p_sta1 and *p_sta2 are the same     (19)
```

Lines 1-3 and 11-13 are selfexplanatory: the type printed corresponds exactly to the true type of the objects (leading characters are added to the names of types by the compiler — they are irrelevant to us and could be different for another compiler). Since we refer to the objects by their names, and not by pointers or references, no polymorphism is possible anyway.

Types printed on lines 4-5 (and 14-15) of the printout are types od *pointers*, not of the objects which are pointed to by them (name 'P3Car' is the name given by the compiler to the type *pointer to 3Car*). Comparing lines 4-5 with lines 14-15 one can see that polymorphism has nothing to do with this case; the type reported is the true type of the pointers.

If *p* is a pointer, then typeid(p) determines the type of this pointer, not of the object it points to.

However, we can ask about the type of objects pointed to by these pointers, e.g., about the type of ***p***, where ***p*** is a pointer. We refer to the object by pointer, so polymorphism can work, provided that we deal with polymorphic classes, i.e., classes containing at least one virtual function: it can even be the destructor alone, as it is the case for the pair **Building** ← **Station** in our example.

Let us look at lines 6 and 7 of the printout. The true type of objects pointed to by ***p_car1*** and ***p_car2*** is the derived type **Car**. The static (declared) types of the pointers are **Car*** and **Vehicle***, respectively. The classes are *not* polymorphic, so the reported type of ***p_car1*** and ***p_car2*** will correspond to the *static* type, i.e., **Car** and **Vehicle**, respectively. The situation will be different for types of ***p_sta1*** and ***p_sta2***. The true type of the objects pointed to by these pointers is the derived type **Station**. We refer to these objects by pointer, and classes *are* polymorphic. Therefore, as one can see on lines 16 and 17 of the printout, now their true type has been recognized.

The same holds for references. From lines 8 and 18 we can see that for non-polymorphic classes (**Vehicle**), the static type has been found, while for polymorphic one it was the true (dynamic) type.

Lines 9 and 19 demonstrate that comparing types of objects can give misleading results for non-polymorphic types. Types of objects pointed to by ***p_car1*** and ***p_car2*** has been reported as different (line 9), although they are in fact the same, only the types of pointers are different. For polymorphic classes the types of objects pointed to by ***p_sta1*** and ***p_sta2*** are correctly recognized as being the same (line 19), although the types of these two pointers are different.

Prev: 25. Run-time type identification **Up:** 25. Run-time type identification **Next:** 25.2 Operator dynamic_cast

25.2 Operator *dynamic_cast*

The **dynamic cast opertor** is used when validity of a cast cannot be determined at compile time, because it depends on dynamic type of an object of a polymorphic class.

Suppose we have a class **A** and a derived class **B** which inherits from **A**. The classes have to be polymorphic, i.e., they have to contain at least one virtual method (virtual destructor would be sufficient (see section on inheritance). Next, let us suppose that there is a function with parameter of type **A*** or **A&**. Is it possible to call the function with argument of type **B*** or **B**? Any object of type **B** can be viewed as an object of class **A**, because it contains all members that **A** has (and, possibly, some more). Therefore, such conversion is legal and will be performed automatically. Of course, all this applies to passing arguments by pointers or by references, *not* by values. Values of type **B** cannot play the rôle of values of type **A** on th eprogram stack, because, for example, they have different size (formally, this is possible, but after "slicing" objects — what is pushed on the stack is the subobject of type **A** of object of type **B**, what not necessarily is what we want).

However, the situation is quite different if, on the contrary, we would like to assign a value of type **A*** to variable of type **B***. The object which is pointed to by the pointer may, but does not have to be of type **B** — its type can be **A** or any other type derived from **A**. Therefore, a conversion in this direction, from pointer to a base-class object to pointer to a derived-class object, can fail. Consequenly, we must require it explicitly. Validity of such a conversion cannot be checked by the compiler, because it does not know what the type of an object will be at run time. Hence, static conversion is not a solution here — we have to use **dynamic_cast**. Its syntax is the following:

```
dynamic_cast<Type>(expr)
```

There are two possibilities:

- Type **Type** is a pointer to the polymorphic derived class, **Type** = **B***, and the value of ***expr*** is of type **A***. Cast operator checks if the object pointed to by the pointer is in fact of type **B**; the value of the cast will be:
  - pointer value of type **B*** pointing to the object pointed to by ***expr***, if the test succeeded;
  - zero (empty pointer, **nullptr**), if the test failed (the object is not of type **B**).
- Type **Type** is a reference to the polymorphic derived type, **Type** = **B&**, and the value of ***expr*** is the l-value of an object of type **A**. Cast operator checks if the object referred is in fact of type **B** and then:
  - if test succeded, the cast yields a reference of type **B&** to object which is referred to by ***expr***;
  - if test failed, because the object referred to by ***expr*** is not of type **B**, then an exception of type **bad_cast** (from the header `typeinfo`) is thrown; it can be caught and handled by the program. The information about the failure cannot be conveyed by zero value of the result of the cast, because there is no such thing like "null reference" (but there are null pointers, used in the previous case).

Let us consider an example:

P219: prgr.cpp Dynamic cast

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. class Program {
6. protected:
7.     string name;
8. public:
9.     Program(string n)
10.         : name(n)
11.     { }
12.     virtual void print() = 0;
13.     virtual ~Program() { };
14. };
15.
16. class Freeware : public Program {
17. public:
18.     Freeware(string n)
19.         : Program(n)
20.     { }
21.     void print() {
22.         cout << "Free : " << name << endl;
23.     }
24. };
25.
26. class Shareware : public Program {
27.     int price;
28. public:
29.     Shareware(string n, int c)
30.         : Program(n), price(c)
31.     { }
32.     void print() {
33.         cout << "Share: "  << name
34.              << ", price " << price << endl;
35.     }
36.     int getPrice() {
37.         return price;
38.     }
39. };
40.
41. int total(Program* prgs[], int size) {
42.     Shareware* sh;
43.     int tot = 0;
44.     for (int i = 0; i < size; ++i) {
45.         prgs[i]->print();
46.         if ( sh = dynamic_cast<Shareware*>(prgs[i]) )
47.             tot += sh->getPrice();
48.     }
49.     return tot;
50. }
51.
52. int main() {
53.     Freeware anjuta("Anjuta"); Shareware wz("WinZip",30);
54.     Freeware mysql("MySQL");   Shareware rar("RAR",25);
55.
56.     Program* prgs[] = { &anjuta, &wz, &mysql, &rar };
57.
58.     int tot = total(prgs, sizeof(prgs)/sizeof(prgs[0]));
59.
60.     cout << "\nTotal: $" << tot << endl;
61. }
```

We define an abstract class **Program** and two derived concrete classes: **Freeware** and **Shareware**. Both are polymorphic, because the base class declares (pure) virtual method **print**. Note that **Shareware** has additional members, a field ***price*** and method **getPrice**, which are not present neither in the base class nor in the other derived class. The function **total** takes an array of pointers of type **Program***. It prints information on each object

pointed to by subsequent elements of the array; this works well, because **print** is virtual, so correct overriding method will be invoked depending on the dynamic type of the object. However, the function sums up prices of theprograms. There is a problem here — the function **getPrice** cannot be invoked for pointers of type **Program***, because no such method has been declared in class **Program**. We know, however, that some of the objects are in fact objects of type **Shareware** and in this class price is defined. Therefore, we cast pointers of the base type **Program*** to type **Shareware*** (line 46). If this cast succeedes, then the variable ***sh*** is non-null and points to an object of type **Shareware**, so we can safely call **getPrice** for it (line 47). If the cast fails, then ***sh*** will be **nullptr** and the test in the **if** statement will fail — consequently, the method **getPrice** will not be called, what is what we want, because the object is then of type **Freeware** which does not define any method called **getPrice**

The result of the program is:

```
Free : Anjuta
Share: WinZip, price 30
Free : MySQL
Share: RAR, price 25

Total: $55
```

To illustrate the second possibility, we could have defined the function **total** in the following way:

```
1.      int total(Program* prgs[], int size) {
2.          int tot = 0;
3.          for (int i = 0; i < size; ++i) {
4.              prgs[i]->print();
5.              try {
6.                  Shareware& sh =
7.                      dynamic_cast<Shareware&>(*prgs[i]);
8.                  tot += sh.getPrice();
9.              } catch(bad_cast) { }
10.         }
11.         return tot;
12.     }
```

This time we do not cast a pointer, but the l-value of object which is pointed to by pointer ***prgs[i]***, i.e., ***prgs[i]*** (line 7). Casting is to a reference type **Shareware&**, so polymorphism can be used. If the cast succeeds, then we read the price and add it to the total. If it fails, then it means that the object was not of the type **Shareware**: an exception is thrown and the program does not reach line 8 but enters the **catch** clause where the exception is simply ignored and the loop is continued for the next element of the array.

Index

! (NOT)
see operator, logical NOT
*** (indirection)**
see operator, indirection
*** (multiplication)**
see operator, multiplication
+ (addition)
see operator, addition
::
see operator, scope
(OR)see operator, bitwise OR
~ (NOT)
see operator, bitwise NOT
++
see operator, increment
- -
see operator, decrement
- (subtraction)
see operator, substraction
& (address-of)
see operator, address-of
& (AND)
see operator, bitwise AND
& (reference)
see reference
&& (AND)
see operator, logical AND
% (remainder)
see operator, remainder
__linux__
3.2 Preprocessor directives
_WIN32
3.2 Preprocessor directives
aarrays.cpp (file)
5.6 Arrays of type std::array
abort
22.1 Throwing exceptions
abstract class
21.5 Abstract classes
acc.cpp (file)
14.2 Accessibility of class members
accessibility level
14.2 Accessibility of class members
accout.cpp (file)
14.4 Methods
Ada
see programming languages
addition operator
9.2.5 Arithmetic operators
address-of (&)
9.2.3 Operators of precedence 14
address-of operator
see operator, address-of
adhocforeach.cpp (file)
8.9.4 Foreach loop
adjustfield
16.3.1 Format flags

ADL
15.4 Friend functions
aggreg.cpp (file)
14.8 Creating objects
aggregate
14.8 Creating objects
agrarr.cpp (file)
14.9 Arrays of objects
algorithm
24.2 Algorithms and function objects

Euclid's
11.8 Recursive functions
insertion sort
11.14 Function templates | 12.3 Deallocation of memory — operator | 13.1 C-structures
mutating
24.2.1 Algorithms
sorting
24.2.1 Algorithms

alignment
13.3 Unions
alloc.cpp (file)
12.2 Memory allocation — new operator
Anjuta
1.1.1 Linux
app (opening mode)
16.5 Files
append
17.2.2 Methods and operators
argument
11.1 Introduction | 11.3 Function call

command-line
2.1 Simplest C++ programs
default
11.4 Default arguments | 14.4 Methods | 14.6 Constructors
of command-line
2.4 Command-line arguments
reference
11.6 Reference arguments

arguments.cpp (file)
2.4 Command-line arguments
arr2dim.cpp (file)
5.5.1 Matrices
arr2dim2.cpp (file)
5.5.1 Matrices
arr3dim.cpp (file)
19.4.3 Function call operator
array
8.9.4 Foreach loop

character
5.4 Character arrays (C-strings)
dimension of
5.1 Defining arrays
dynamc

multidimensional
12.4 Dynamic multidimensional arrays

dynamic
12.2 Memory allocation — new operator
multidimensional
5.5.1 Matrices | 12.2 Memory allocation — new operator
of aggregates
14.9 Arrays of objects

of C-strings
5.5.2 Arrays of C-strings
of objects
14.9 Arrays of objects | 14.9 Arrays of objects
size of
5.1 Defining arrays
static
5.1 Defining arrays | 5.1 Defining arrays

arrayfun.cpp (file)
11.12 Function pointers
arrays.cpp (file)
5.2 Array type
arrfunc.cpp (file)
5.2 Array type
arrref.cpp (file)
11.6 Reference arguments
arrstr.cpp (file)
5.5.2 Arrays of C-strings
arythmpoi.cpp (file)
5.3 Pointer arithmetic
ASCII
4.2 Integral types | 4.2 Integral types
assert
11.12 Function pointers | 13.3 Unions
assign
17.2.2 Methods and operators
assign.cpp (file)
9.2.9 Assignment operators
assignment operator
9.2.9 Assignment operators | 19.3 Overloading with methods of
associativity
9.1 Precedence and associativity of
at
17.2.2 Methods and operators | 24.1.1 Vectors
ate (opening mode)
16.5 Files
ate.cpp (file)
24.1.1 Vectors
atof
17.1.3 Conversion functions
atoi
17.1.3 Conversion functions
atol
17.1.3 Conversion functions
atomic variable
7.4.1 Volatile variables
auto
see variable, auto | 4.1 Introductory remarks
autodecl.cpp (file)
4.1 Introductory remarks
autoret.cpp (file)
11.14 Function templates
back
24.1.1 Vectors
back_inserter
24.2.1 Algorithms
backins.cpp (file)
24.2.1 Algorithms
bad
16.6 Handling stream errors
bad_alloc
22.7 Standard exceptions
bad_cast
22.7 Standard exceptions
bad_exception
22.7 Standard exceptions | 22.7 Standard exceptions

bad_typeid
22.7 Standard exceptions
badbit
16.6 Handling stream errors
base class
21.1 Fundamentals of inheritance
basefield
16.3.1 Format flags
begin
17.2.2 Methods and operators | 24.1.2 Iterators
big endian
5.3 Pointer arithmetic
big-endian
16.5 Files
binary (opening mode)
16.5 Files
binary operator
9. Operators
binding

early
21.4 Virtual methods and polymorphism
late
21.4 Virtual methods and polymorphism

bit field
14.10 Bit fields
bitColors.cpp (file)
9.2.7 Bitwise operators
bitfields.cpp (file)
14.10 Bit fields
bits.cpp (file)
9.2.7 Bitwise operators
bitwise AND (&)
9.2.7 Bitwise operators
bitwise NOT (~)
9.2.3 Operators of precedence 14 | 9.2.7 Bitwise operators
bitwise operator
9.2.7 Bitwise operators
bitwise OR (
)
bitwise shift
9.2.7 Bitwise operators
bitwise XOR (^)
9.2.7 Bitwise operators
blocking
2.1 Simplest C++ programs
bool
4.4 Logical type
boolalpha
16.3.1 Format flags
brace-init
4.1 Introductory remarks
byte code
1.1 Compilers
.C
2.1 Simplest C++ programs | 2.1 Simplest C++ programs | 3.2 Preprocessor directives | 23.1.1 Headers and implementation files
C++11
4.1 Introductory remarks | 4.5 Enumeration types | 5.1 Defining arrays | 7.4.2 Constants | 11.2 Declarations and definitions of | 11.5 Variable-length argument lists | 11.12 Function pointers | 11.13 Lambda functions | 15.3.2 Initialization lists
C++11 standard
4.5 Enumeration types | 11.2 Declarations and definitions of
C-string
17.1 C-strings
c_str
17.2.2 Methods and operators

callable object
19.4.3 Function call operator | 24.2.2 Function objects
calling function
11.3 Function call
calloc
12.5 Memory management in C
CamelCase
7.1 Keywords and names
cast
see operator, cast | 9.2.2 Operators of precedence 15 | 20.2 Explicit conversions

dynamic
21.1 Fundamentals of inheritance
static
21.1 Fundamentals of inheritance

cast operator
9.2.3 Operators of precedence 14
cast.cpp (file)
21.1 Fundamentals of inheritance
catch
8.13 Exception handling statements | 22.2 Catching exceptions
cctype
17.1.2 Functions operating on characters
cerr
16.1 Streams | 16.1.1 Predefined streams
cerrno
17.1.3 Conversion functions
char
see type | 4.2 Integral types
char16_t
4.2 Integral types
char32_t
4.2 Integral types
chararr.cpp (file)
5.4 Character arrays (C-strings)
cin
see standard input stream | 2.2 Simplest In/Out operations | 2.2.2 Data input | 16.1 Streams | 16.1.1 Predefined streams
class
11.14 Function templates | 14.1 Introduction

abstract
21.5 Abstract classes
base
21. Inheritance and polymorphism | 21.1 Fundamentals of inheritance
concrete
21.5 Abstract classes
derived
21. Inheritance and polymorphism | 21.1 Fundamentals of inheritance
enclosing
15.5 Nested classes
field
14.3 Fields

static
14.3 Fields

local
15.5 Nested classes
mix-in
21.7 Multiple inheritance
nested
15.5 Nested classes
polymorphic
21.4 Virtual methods and polymorphism
section
14.2 Accessibility of class members
template of

18.1 Templates of classes

classarr.cpp (file)
14.9 Arrays of objects
clear
16.6 Handling stream errors | 17.2.2 Methods and operators
clog
16.1 Streams | 16.1.1 Predefined streams
cnstcnst.cpp (file)
7.4.2 Constants
Code::Blocks
1.1.1 Linux
CodeWarrior
1.1.1 Linux
collection
24.1 Collections and iterators
comma operator
9.2.12 Comma operator
comma.cpp (file)
9.2.12 Comma operator
comp.cpp (file)
9.2.9 Assignment operators
compar.cpp (file)
24.2.2 Function objects
comparator
24.2.1 Algorithms
compare
17.2.2 Methods and operators
compilation
1.1 Compilers
compound assignment operator
9.2.9 Assignment operators
compound statement
8.5 Compound statement
concast.cpp (file)
20.2.1 const_cast conversions
concatenation
17.1.1 Functions on C-strings
concrete class
21.5 Abstract classes
concretization
11.14 Function templates
condes.cpp (file)
21.2 Constructors and destructors of
conditional operator
9. Operators | 9.2.10 Conditional operator
conditional statement
8.7 Conditional statement
confield.cpp (file)
15.4 Friend functions
const
see variable, constant | 5.1 Defining arrays | 7.4.2 Constants
const_cast
20.2.1 const_cast conversions
const_cast conversion
20.2.1 const_cast conversions
const_iterator
24.1.2 Iterators
const_reverse_iterator
24.1.2 Iterators
constants.cpp (file)
7.4.2 Constants
constants1.cpp (file)
7.4.2 Constants
constants2.cpp (file)
7.4.2 Constants
constexpr

5.1 Defining arrays | 7.4.2 Constants
constexpr.cpp (file)
7.4.2 Constants
constit.cpp (file)
24.1.2 Iterators
constmet.cpp (file)
15.1 Constant methods
constpoint.cpp (file)
7.4.2 Constants
constructor
14.6 Constructors

conversion
20.1.1 Conversions to a defined type
copying
15.3.1 Copy constructors
default
14.6 Constructors
delegating
15.3.2 Initialization lists
exception in
22.4 Exceptions in constructors and
move
19.5 Move semantics

continue
8.10 continue and break statements
conv.cpp (file)
10.1 Conversions
conversion
4.6.2 Generic pointers | 9.2.2 Operators of precedence 15 | 9.2.3 Operators of precedence 14 | 10.1 Conversions | 20. More on conversions

const_cast
20.2.1 const_cast conversions
dynamic
20.2.3 Dynamic conversions
forced
20.2.4 Forced conversions
of enumeration
4.5 Enumeration types
static
20.2.2 Static casts
to user-defined type
20.1.1 Conversions to a defined type

conversion method
20.1.2 Conversions from user-defined type
convfrom.cpp (file)
20.1.2 Conversions from user-defined type
convto.cpp (file)
20.1.1 Conversions to a defined type
cop.cpp (file)
21.2 Constructors and destructors of
copy
17.2.2 Methods and operators

deep
15.3.1 Copy constructors
shallow
15.3.1 Copy constructors

copy-constructor
15.3.1 Copy constructors | 21.2 Constructors and destructors of
copy_if
24.2.1 Algorithms
copycon.cpp (file)
15.3.1 Copy constructors

count_if
24.2.1 Algorithms | 24.3 Examples
cout
see standard output stream | 2.2 Simplest In/Out operations | 16.1 Streams | 16.1.1 Predefined streams
.cpp
2.1 Simplest C++ programs | 23.1.1 Headers and implementation files
creatob.cpp (file)
14.8 Creating objects
cstdlib
12.5 Memory management in C | 17.1.3 Conversion functions
cstring
12.6 Functions operating on memory | 17.1.1 Functions on C-strings
cstruct.cpp (file)
13.1 C-structures
ctor
14.8 Creating objects
cvscpp.c (file)
3.2 Preprocessor directives
.cxx
2.1 Simplest C++ programs | 23.1.1 Headers and implementation files
__DATE__
3.3 Predefined preprocessor macros
datetime.cpp (file)
3.3 Predefined preprocessor macros
dec
16.3.1 Format flags
dec (manipulator)
16.3.2.1 Parameterless manipulators
decl.cpp (file)
6.1 Defining “compound” types
decl1.cpp (file)
4.2 Integral types
declaration
4.1 Introductory remarks | 7.1 Keywords and names | 8.3 Declarations

forward
13.1 C-structures

decltype
4.1 Introductory remarks
decrement
9.2.2 Operators of precedence 15
decrement operator
9.2.3 Operators of precedence 14
deep copy
15.3.1 Copy constructors
default argument
11.4 Default arguments | 14.4 Methods
default constructor
14.6 Constructors
#define
3.2 Preprocessor directives
defined
3.2 Preprocessor directives
definition
4.1 Introductory remarks | 7.1 Keywords and names
defval.cpp (file)
11.4 Default arguments
defval1.cpp (file)
11.4 Default arguments
defval1h.h (file)
11.4 Default arguments
delegating constructor
15.3.2 Initialization lists
delegconstr.cpp (file)
15.3.2 Initialization lists
delete

12.3 Deallocation of memory — operator
delete operator
9.2.3 Operators of precedence 14
deleter
19.6.1 Smart pointers of the
delunique.cpp (file)
19.6.1 Smart pointers of the
deque
24.1.2 Iterators
dereference (*)
9.2.3 Operators of precedence 14
dereferencing
see operator, indirection
derived class
21.1 Fundamentals of inheritance
destination
16.1 Streams
destructor
14.7 Destructors

exception in
22.4 Exceptions in constructors and | 22.4 Exceptions in constructors and
virtual
21.6 Virtual destructors

dice.cpp (file)
8.9.2 do-while loop
dista.cpp (file)
24.1.3 Operations on collections
distance
24.1.3 Operations on collections
division operator
9.2.5 Arithmetic operators
.dll
23.1.1 Headers and implementation files
do
8.9.2 do-while loop
do-while
8.9.2 do-while loop
double
see type
downcasting
21.1 Fundamentals of inheritance
dtor
14.8 Creating objects
dynamic array
12.2 Memory allocation — new operator
dynamic conversion
20.2.3 Dynamic conversions
dynamic type
21.4 Virtual methods and polymorphism
dynamic_cast
20.2.3 Dynamic conversions | 22.7 Standard exceptions | 25.2 Operator dynamic_cast
dyncast.cpp (file)
20.2.4 Forced conversions
early binding
21.4 Virtual methods and polymorphism
Eclipse
1.1.1 Linux
#elif
3.2 Preprocessor directives
#else
3.2 Preprocessor directives
emplace_back
24.1.1 Vectors
empty
17.2.2 Methods and operators

emulstr.cpp (file)
17.1.1 Functions on C-strings
enclosing class
15.5 Nested classes
end
17.2.2 Methods and operators | 24.1.2 Iterators
#endif
3.2 Preprocessor directives
endl
2.2.1 Outputting information
endl (manipulator)
16.3.2.1 Parameterless manipulators
ends (manipulator)
16.3.2.1 Parameterless manipulators
enum
4.5 Enumeration types
enumeration
see type
enums.cpp (file)
4.5 Enumeration types
EOF
16.4.1 Unformatted input | 16.6 Handling stream errors
eofbit
16.6 Handling stream errors
epoch
13.1 C-structures
equality operator
9.2.6 Relational operators and comparisons
erase
17.2.2 Methods and operators | 24.1.3 Operations on collections | 24.3 Examples
errno
17.1.3 Conversion functions
#error
3.2 Preprocessor directives
Euclid
11.8 Recursive functions
Euclid's algorithm
11.8 Recursive functions
except.cpp (file)
22.2 Catching exceptions
exception
22. Exceptions

bad_alloc
22.7 Standard exceptions
bad_cast
22.7 Standard exceptions
bad_exception
22.7 Standard exceptions
bad_typeid
22.7 Standard exceptions
handling
22.2 Catching exceptions
hierarchy of
22.3 Hierarchies of exceptions
in constructor
22.4 Exceptions in constructors and
in destructor
22.4 Exceptions in constructors and | 22.4 Exceptions in constructors and
ios::failure
22.7 Standard exceptions
out_of_range
17.2.2 Methods and operators | 24.1.1 Vectors
specification
22.5 Exception specifications
throwing
22.1 Throwing exceptions

exception handling
8.13 Exception handling statements
exception-throw
9.2.11 Exception-throw operator
exceptions
22.7 Standard exceptions
executable
1.1 Compilers | 1.1.1 Linux
exit
22.1 Throwing exceptions
explicit
20.1.1 Conversions to a defined type
expression statement
8.6 Expression statement
exprstat.cpp (file)
8.6 Expression statement
exter1.cpp (file)
7.3.2 External variables
exter2.cpp (file)
7.3.2 External variables
extern
see variable, extern | 7.2 Scope and visibility
external variable
7.3.2 External variables | 23.1 Modules of a program
fail
16.6 Handling stream errors
failbit
16.6 Handling stream errors
false
4.4 Logical type
fibo.cpp (file)
11.8 Recursive functions
Fibonacci
11.8 Recursive functions | 11.13 Lambda functions
Fibonacci sequence
11.13 Lambda functions
field
13.1 C-structures | 14.3 Fields

mutable
15.1.1 mutable fields

figure.cpp (file)
21.4 Virtual methods and polymorphism
__FILE__
3.3 Predefined preprocessor macros

header
23.1.1 Headers and implementation files
header file
2.1 Simplest C++ programs | 2.1 Simplest C++ programs
implementation
23.1.1 Headers and implementation files
internal
16.7 Internal files
opening mode
16.5 Files

filerw.cpp (file)
16.5 Files
fill
16.3.1 Format flags
find
17.2.2 Methods and operators | 24.2.1 Algorithms
find_first_not_of
17.2.2 Methods and operators
find_first_of

17.2.2 Methods and operators
find_if
24.2.1 Algorithms
find_last_not_of
17.2.2 Methods and operators
find_last_of
17.2.2 Methods and operators
fixed
16.3.1 Format flags
fixed (manipulator)
16.3.2.1 Parameterless manipulators
flag

adjustfield
16.3.1 Format flags
basefield
16.3.1 Format flags
boolalpha
16.3.1 Format flags
dec
16.3.1 Format flags
fixed
16.3.1 Format flags
floatfield
16.3.1 Format flags
hex
16.3.1 Format flags
internal
16.3.1 Format flags
left
16.3.1 Format flags
noboolalpha
16.3.1 Format flags
noshowbase
16.3.1 Format flags
noshowpoint
16.3.1 Format flags
noshowpos
16.3.1 Format flags
noskipws
16.3.1 Format flags
nounitbuf
16.3.1 Format flags
nouppercace
16.3.1 Format flags
oct
16.3.1 Format flags
right
16.3.1 Format flags
scientific
16.3.1 Format flags
showbase
16.3.1 Format flags
showpoint
16.3.1 Format flags
showpos
16.3.1 Format flags
skipws
16.3.1 Format flags
unitbuf
16.3.1 Format flags
uppercace
16.3.1 Format flags

flags
16.3.1 Format flags
flags.cpp (file)

16.3.1 Format flags
float
see type
floatfield
16.3.1 Format flags
flush (manipulator)
16.3.2.1 Parameterless manipulators
fmtflags
16.3.1 Format flags
for
8.9.3 for loop
for_each
24.2.1 Algorithms
forced conversion
20.2.4 Forced conversions
foreach.cpp (file)
8.9.4 Foreach loop
format flag
16.3.1 Format flags
formatting
16.2 Operators << and >> | 16.3 Formatting
Fortran
see programming languages
forward iterator
24.1.2 Iterators
free
12.5 Memory management in C
free function
11. Functions
free memory
12.1 Introduction
friend function
15.4 Friend functions
front
24.1.1 Vectors | 24.1.3 Operations on collections
front_inserter
24.2.1 Algorithms
fstream
16.1 Streams | 16.1 Streams | 16.5 Files
ftime
13.1 C-structures
func.cpp (file)
2.3 Functions
__FUNCTION__
2.3 Functions | 3.3 Predefined preprocessor macros

abort
22.1 Throwing exceptions
append
17.2.2 Methods and operators
argument
11.1 Introduction | 11.3 Function call
assert
11.12 Function pointers
assign
17.2.2 Methods and operators
at
17.2.2 Methods and operators | 24.1.1 Vectors
atof
17.1.3 Conversion functions
atoi
17.1.3 Conversion functions
atol
17.1.3 Conversion functions
back
24.1.1 Vectors
back_inserter

24.2.1 Algorithms
begin
17.2.2 Methods and operators | 24.1.2 Iterators
c_str
17.2.2 Methods and operators
call
2.3 Functions | 11.3 Function call
calloc
12.5 Memory management in C
clear
16.6 Handling stream errors | 17.2.2 Methods and operators
compare
17.2.2 Methods and operators
converting
17.1.3 Conversion functions
copy
17.2.2 Methods and operators
copy_if
24.2.1 Algorithms
count_if
24.2.1 Algorithms | 24.3 Examples
definition
11.2 Declarations and definitions of
distance
24.1.3 Operations on collections
emplace_back
24.1.1 Vectors
empty
17.2.2 Methods and operators
end
17.2.2 Methods and operators | 24.1.2 Iterators
erase
17.2.2 Methods and operators | 24.1.3 Operations on collections | 24.3 Examples
exceptions
22.7 Standard exceptions
exit
22.1 Throwing exceptions
fill
16.3.1 Format flags
find
17.2.2 Methods and operators | 24.2.1 Algorithms
find_first_not_of
17.2.2 Methods and operators
find_first_of
17.2.2 Methods and operators
find_last_not_of
17.2.2 Methods and operators
find_last_of
17.2.2 Methods and operators
flags
16.3.1 Format flags
free
11. Functions | 11.1 Introduction | 12.5 Memory management in C
friend
15.4 Friend functions
front
24.1.1 Vectors
front_inserter
24.2.1 Algorithms
ftime
13.1 C-structures
gcount
16.4.1 Unformatted input
get
16.4.1 Unformatted input
getline
17.2.2 Methods and operators

global
2.1 Simplest C++ programs | 11. Functions | 11.1 Introduction
header
11.2 Declarations and definitions of
ignore
16.4.1 Unformatted input
inline
11.2 Declarations and definitions of | 14.4 Methods
inlined
11.10 Inlined functions
insert
17.2.2 Methods and operators | 24.1.3 Operations on collections
inserter
24.2.1 Algorithms
invocation
11.3 Function call
invocation of
2.3 Functions
isalnum
17.1.2 Functions operating on characters
isalpha
17.1.2 Functions operating on characters
iscntrl
17.1.2 Functions operating on characters
isdigit
17.1.2 Functions operating on characters
isgraph
17.1.2 Functions operating on characters
isprint
17.1.2 Functions operating on characters
ispunct
17.1.2 Functions operating on characters
isspace
17.1.2 Functions operating on characters
isxdigit
17.1.2 Functions operating on characters
length
17.2.2 Methods and operators
main
2.1 Simplest C++ programs | 2.1 Simplest C++ programs | 2.4 Command-line arguments | 8.12 return statement
make_shared
19.6.2 Smart pointers of the
make_unique
19.6.1 Smart pointers of the
malloc
12.5 Memory management in C
max_element
24.3 Examples
memchr
12.6 Functions operating on memory
memcmp
12.6 Functions operating on memory
memcpy
12.6 Functions operating on memory
memmove
12.6 Functions operating on memory
memset
12.6 Functions operating on memory
merge
24.2.1 Algorithms
method
14.4 Methods
min_element
24.3 Examples
overloading
11.11 Function overloading | 14.6 Constructors
parameter

11.1 Introduction | 11.2 Declarations and definitions of
peek
16.4.1 Unformatted input
pointer
11. Functions
pop_back
24.1.1 Vectors
precision
16.3.1 Format flags
prototype
11.2 Declarations and definitions of
push_back
17.2.2 Methods and operators | 24.1.1 Vectors
put
16.4.2 Unformatted output
putback
16.4.1 Unformatted input
rbegin
17.2.2 Methods and operators | 24.1.2 Iterators
rdstate
16.6 Handling stream errors
realloc
12.5 Memory management in C
recursive
11.8 Recursive functions
release
19.6.1 Smart pointers of the
remove
24.2.1 Algorithms
remove_if
24.3 Examples
rend
17.2.2 Methods and operators | 24.1.2 Iterators
replace
17.2.2 Methods and operators
reset
19.6.1 Smart pointers of the
resize
17.2.2 Methods and operators
returning reference
11.6 Reference arguments | 11.7 Functions returning references
reverse
24.2.1 Algorithms
rfind
17.2.2 Methods and operators
seekg
16.5 Files
seekp
16.5 Files
set_terminate
22.1 Throwing exceptions
setf
16.3.1 Format flags
setstate
16.6 Handling stream errors
signature
11.2 Declarations and definitions of | 11.11 Function overloading
size
17.2.2 Methods and operators | 24.1.1 Vectors
sort
24.2.1 Algorithms
static
11.9 Static functions | 14.5 Static member functions
str
16.7.2 Internal files represented by
strcat
17.1.1 Functions on C-strings

strchr
17.1.1 Functions on C-strings
strcmp
17.1.1 Functions on C-strings
strcoll
17.1.1 Functions on C-strings
strcpy
17.1.1 Functions on C-strings
strcspn
17.1.1 Functions on C-strings
strlen
17.1.1 Functions on C-strings
strncat
17.1.1 Functions on C-strings
strncmp
17.1.1 Functions on C-strings
strncpy
17.1.1 Functions on C-strings
strpbrk
17.1.1 Functions on C-strings
strrchr
17.1.1 Functions on C-strings
strspn
17.1.1 Functions on C-strings
strstr
17.1.1 Functions on C-strings
strtod
17.1.3 Conversion functions
strtok
17.1.1 Functions on C-strings
strtol
17.1.3 Conversion functions
strtoul
17.1.3 Conversion functions
substr
17.2.2 Methods and operators
swap
17.2.2 Methods and operators
tellg
16.5 Files
tellp
16.5 Files
template of
11.14 Function templates
terminate
22.1 Throwing exceptions
unget
16.4.1 Unformatted input
unsetf
16.3.1 Format flags
virtual
21.4 Virtual methods and polymorphism
what
22.7 Standard exceptions
width
16.3.1 Format flags
with default argument
11.4 Default arguments | 14.4 Methods
with variable-length argument list
11.5 Variable-length argument lists
write
16.4.2 Unformatted output

function call
9.2.2 Operators of precedence 15
function object
11.12 Function pointers | 19.4.3 Function call operator | 24.2.2 Function objects

function pointer
11. Functions
function.cpp (file)
11.3 Function call
functions

operating on C-strings
17.1.1 Functions on C-strings

functor
24.2.2 Function objects
fundefnew.cpp (file)
11.2 Declarations and definitions of
funkcja

find_if
24.2.1 Algorithms
for_each
24.2.1 Algorithms
front
24.1.3 Operations on collections
islower
17.1.2 Functions operating on characters
isupper
17.1.2 Functions operating on characters
lambda
11.13 Lambda functions
pop_front
24.1.3 Operations on collections
push_front
24.1.3 Operations on collections
sort
24.3 Examples
tolower
17.1.2 Functions operating on characters
toupper
17.1.2 Functions operating on characters

funkction

getline
16.4.1 Unformatted input
read
16.4.1 Unformatted input

funpoint.cpp (file)
11.12 Function pointers
funretur.cpp (file)
11.12 Function pointers
gcd.cpp (file)
11.8 Recursive functions
gcount
16.4.1 Unformatted input
Geany
1.1.1 Linux
get
16.4.1 Unformatted input
getline
16.4.1 Unformatted input | 17.2.2 Methods and operators
global function
11. Functions | 11.1 Introduction
global variable
7.2 Scope and visibility
good
16.6 Handling stream errors
goodbit
16.6 Handling stream errors
goto

8.11 goto statement
greeting.cpp (file)
14.2 Accessibility of class members
.h
2.1 Simplest C++ programs | 23.1.1 Headers and implementation files | 23.2 Namespaces
head of the list
13.1 C-structures
header
2.1 Simplest C++ programs | 23.1.1 Headers and implementation files
header file
2.1 Simplest C++ programs
heap
12.1 Introduction | 12.1 Introduction
helloWorld.cpp (file)
2.1 Simplest C++ programs
hex
16.3.1 Format flags
hex (manipulator)
16.3.2.1 Parameterless manipulators
hex.cpp (file)
8.8 Selection (switch)statement (switch)
hidden variable
7.2 Scope and visibility
hide.cpp (file)
7.2 Scope and visibility
hiding
see variable, hidden
hier.cpp (file)
22.3 Hierarchies of exceptions
hier1.cpp (file)
22.3 Hierarchies of exceptions
hierob.cpp (file)
22.3 Hierarchies of exceptions
HUGE_VAL
17.1.3 Conversion functions
identifier
4.1 Introductory remarks | 7.1 Keywords and names | 11.2 Declarations and definitions of
#if
3.2 Preprocessor directives
#ifdef
3.2 Preprocessor directives
#ifndef
3.2 Preprocessor directives
ifstream
16.1 Streams | 16.1 Streams | 16.5 Files
ignore
16.4.1 Unformatted input
in (opening mode)
16.5 Files
#include
2.1 Simplest C++ programs | 3.2 Preprocessor directives | 23.1.1 Headers and implementation files
incomplete
13.1 C-structures
increment
9.2.2 Operators of precedence 15
increment operator
9.2.3 Operators of precedence 14
index
5.1 Defining arrays
indexing
9.2.2 Operators of precedence 15
indirection (*)
9.2.3 Operators of precedence 14
inf
17.1.3 Conversion functions
infix notation
9. Operators

info
1.1.1 Linux
inhas.cpp (file)
21.3 Assignment operator in derived
inhe.cpp (file)
21.3 Assignment operator in derived
inher.cpp (file)
21.1 Fundamentals of inheritance
inheritance
21. Inheritance and polymorphism | 21.1 Fundamentals of inheritance

multibase
21.7 Multiple inheritance
multiple
21.7 Multiple inheritance

inheritance list
21.7 Multiple inheritance
iniagg.cpp (file)
14.8 Creating objects
initalization list
15.3.2 Initialization lists
initialization
4.1 Introductory remarks
initialization list
21.2 Constructors and destructors of
inline function
11.2 Declarations and definitions of | 14.4 Methods
inlined function
11.10 Inlined functions
input iterator
24.1.2 Iterators
input stream
16.1 Streams
input/output
16.1 Streams
input/output operations
2. Let's start, then...
insert
17.2.2 Methods and operators | 24.1.3 Operations on collections
inserter
24.2.1 Algorithms
int
see type | 4.2 Integral types
int16_t
4.2.1 Useful aliases for integral
int32_t
4.2.1 Useful aliases for integral
int64_t
4.2.1 Useful aliases for integral
int8_t
4.2.1 Useful aliases for integral
internal
16.3.1 Format flags
internal (manipulator)
16.3.2.1 Parameterless manipulators
internal file
16.7 Internal files
intfilout.cpp (file)
16.7.1 C-strings as internal files
intstr.cpp (file)
16.7.2 Internal files represented by
invoking function
11.3 Function call
ios
16.1 Streams
ios::failure

22.7 Standard exceptions
ios_base
16.1 Streams | 16.3.1 Format flags | 16.3.1 Format flags
iostream
16.1 Streams
isalnum
17.1.2 Functions operating on characters
isalpha
17.1.2 Functions operating on characters
iscntrl
17.1.2 Functions operating on characters
isdigit
17.1.2 Functions operating on characters
isgraph
17.1.2 Functions operating on characters
isinside.cpp (file)
15.4 Friend functions
islower
17.1.2 Functions operating on characters
isprint
17.1.2 Functions operating on characters
ispunct
17.1.2 Functions operating on characters
isspace
17.1.2 Functions operating on characters
istream
16.1 Streams
istringstream
16.1 Streams | 16.7.2 Internal files represented by
istrstream
16.1 Streams | 16.1 Streams | 16.7.1 C-strings as internal files | 16.7.1 C-strings as internal files
isupper
17.1.2 Functions operating on characters
isxdigit
17.1.2 Functions operating on characters
iterat.cpp (file)
24.1.2 Iterators
iteration statement
8.9 Iteration statements (loops)
iterator
17.2.1 Constructors | 24.1.2 Iterators

bidirectional
24.1.2 Iterators
constant
24.1.2 Iterators
forward
24.1.2 Iterators
input
24.1.2 Iterators
output
24.1.2 Iterators
random access
24.1.2 Iterators
reverse
24.1.2 Iterators

constant
24.1.2 Iterators

Java
see programming languages
john.cpp (file)
2.2.1 Outputting information
KDevelop
1.1.1 Linux
Kernighan, B.

2.1 Simplest C++ programs
keyword
7.1 Keywords and names
Koenig, A.
15.4 Friend functions
l-value
7.5 L-values | 9.2.2 Operators of precedence 15
label
8.2 Labels
labincpp.cpp (file)
8.2 Labels
lambda
see function lambda
lambdagen.cpp (file)
11.13 Lambda functions
lambdamutable.cpp (file)
11.13 Lambda functions
lambdas.cpp (file)
11.13 Lambda functions
late binding
21.4 Virtual methods and polymorphism
lazy evaluation
15.1.1 mutable fields
least astonishment principle
19.1 Introduction | 19.3.2 Unary operators
left
16.3.1 Format flags
left (manipulator)
16.3.2.1 Parameterless manipulators
length
17.2.2 Methods and operators
lengths.cpp (file)
4.1 Introductory remarks
letter.cpp (file)
19.4.2 Indexing operator
__LINE__
3.3 Predefined preprocessor macros
linker
1.1 Compilers
linking
1.1 Compilers | 1.1.1 Linux
list
13.1 C-structures | 24.1.2 Iterators
list.cpp (file)
19.3.1 Binary operators
lists.cpp (file)
13.2 Templates of structures
little endian
5.3 Pointer arithmetic
little-endian
16.5 Files
littlebig.cpp (file)
5.3 Pointer arithmetic
local class
15.5 Nested classes
local variable
7.2 Scope and visibility
logical AND (&&)
9.2.8 Logical operators
logical NOT (!)
9.2.3 Operators of precedence 14 | 9.2.8 Logical operators
long
see type | 4.2 Integral types
long double
see type
long int
4.2 Integral types

long long
see type | 4.2 Integral types
long long int
4.2 Integral types
loop
8.9 Iteration statements (loops)

do-while
8.9.2 do-while loop
for
8.9.3 for loop
foreach
8.9.4 Foreach loop
while
8.9.1 while loop

lval.cpp (file)
7.5 L-values
lvalue
see l-value
main
see function | see function | 2.1 Simplest C++ programs | 8.12 return statement
make_shared
19.6.2 Smart pointers of the
make_unique
19.6.1 Smart pointers of the
malloc
12.5 Memory management in C
man
1.1.1 Linux
mani.cpp (file)
16.3.2.1 Parameterless manipulators
manip.cpp (file)
24.2.2 Function objects
manipulator
2.2.1 Outputting information | 16.3.2 Manipulators | 24.2.2 Function objects

parameterless
16.3.2.1 Parameterless manipulators
with parameters
16.3.2.2 Manipulators with arguments

map
24.3 Examples
maps.cpp (file)
24.3 Examples
masking
9.2.7 Bitwise operators
match.cpp (file)
11.11 Function overloading
matrices.cpp (file)
12.4 Dynamic multidimensional arrays
matrix
5.5.1 Matrices
matrix2dim.cpp (file)
12.4 Dynamic multidimensional arrays
max_element
24.3 Examples
median.cpp (file)
12.3 Deallocation of memory — operator
member
13.1 C-structures | 14.1 Introduction

mutable
15.1.1 mutable fields

memchr
12.6 Functions operating on memory

memcmp
12.6 Functions operating on memory
memcpy
12.6 Functions operating on memory
memmove
12.6 Functions operating on memory
memory

deallocation
12.3 Deallocation of memory — operator

memory allocation
12.2 Memory allocation — new operator
memory functions
12.6 Functions operating on memory
memory leakage
12.2 Memory allocation — new operator
memset
12.6 Functions operating on memory
memstat.cpp (file)
14.5 Static member functions
merge
24.2.1 Algorithms
met.cpp (file)
14.4 Methods
method
14.4 Methods

constant
15.1 Constant methods
conversion
20.1.2 Conversions from user-defined type
overriding
21.4 Virtual methods and polymorphism
pure virtual
21.5 Abstract classes
virtual
21.4 Virtual methods and polymorphism
voaltile
15.2 Volatile methods

Microsoft
1.1.2.1 Visual C++ and Studio
min_element
24.3 Examples
minus operator
9.2.3 Operators of precedence 14
minus.cpp (file)
19.3.2 Unary operators
mix-in class
21.7 Multiple inheritance
mod.cpp (file)
9.2.5 Arithmetic operators
modcon.cpp (file)
20.1.1 Conversions to a defined type
mode
16.5 Files
modifier
7.4 Type modifiers | 8.3 Declarations
modsev.cpp (file)
19.2.1 Binary operators
modsev1.cpp (file)
19.3.1 Binary operators
module
2.1 Simplest C++ programs | 23.1 Modules of a program
move
19.5 Move semantics

move assignment operator
19.5 Move semantics
move constructor
19.5 Move semantics
move semantics
19.5 Move semantics
multbase.cpp (file)
21.7 Multiple inheritance
multiplication operator
9.2.5 Arithmetic operators
mutable
15.1.1 mutable fields
mutable.cpp (file)
15.1.1 mutable fields
mutating algorithm
24.2.1 Algorithms
mySTACK.h (file)
23.2 Namespaces
mySTACKS.h (file)
23.2 Namespaces
mySTACKSImpl.cpp (file)
23.2 Namespaces
namespace
23.2 Namespaces
NDEBUG
11.12 Function pointers
nestcl.cpp (file)
15.5 Nested classes
nested class
15.5 Nested classes
new
12.2 Memory allocation — new operator | 12.7 Placement new operator | 22.7 Standard exceptions
new operator
9.2.3 Operators of precedence 14
nmspc.cpp (file)
23.2 Namespaces
no-op operator
9.2.3 Operators of precedence 14
noboolalpha
16.3.1 Format flags
noexcept
19.5 Move semantics | 22.5 Exception specifications
nonequality operator
9.2.6 Relational operators and comparisons
noshowbase
16.3.1 Format flags
noshowbase (manipulator)
16.3.2.1 Parameterless manipulators
noshowpoint
16.3.1 Format flags
noshowpoint (manipulator)
16.3.2.1 Parameterless manipulators
noshowpos
16.3.1 Format flags
noskipws
16.3.1 Format flags
notation CamelCase
7.1 Keywords and names
nounitbuf
16.3.1 Format flags
nouppercase
16.3.1 Format flags
npos
17.2 Strings in C++
NUL
5.4 Character arrays (C-strings) | 17.1 C-strings
NULL

5.4 Character arrays (C-strings)
null statement
8.4 Null statement
.o
23.1.1 Headers and implementation files
object

callable
19.4.3 Function call operator | 24.2.2 Function objects
function
24.2.2 Function objects

oct
16.3.1 Format flags
oct (manipulator)
16.3.2.1 Parameterless manipulators
ODR
11.2 Declarations and definitions of
ofstream
16.1 Streams | 16.5 Files
one definition rule
11.2 Declarations and definitions of
oneargop.cpp (file)
19.2.2 Unary operators
ope.cpp (file)
19.4.1 Assignment operator
opening mode
16.5 Files
operand
9. Operators
operator
9. Operators

addition
9.2 Overview of operators | 9.2.5 Arithmetic operators
address (&)
13.1 C-structures
address-of (&)
4.6.1 Pointers to variables | 9.2 Overview of operators | 9.2.3 Operators of precedence 14
arithmetic
9.2.5 Arithmetic operators
assignment
9.2 Overview of operators | 9.2.9 Assignment operators | 19.3 Overloading with methods of

compound
9.2.9 Assignment operators
overloading
19.4.1 Assignment operator

associativity
9.1 Precedence and associativity of
binary
9. Operators | 19.3.1 Binary operators

overloading
19.2.1 Binary operators

bitwise
9.2.7 Bitwise operators
bitwise AND (&)
9.2 Overview of operators | 9.2.7 Bitwise operators
bitwise NOT (~)
9.2 Overview of operators | 9.2.3 Operators of precedence 14 | 9.2.7 Bitwise operators
bitwise OR (
) |)
bitwise shift
9.2 Overview of operators | 9.2.7 Bitwise operators
bitwise XOR (^)

9.2 Overview of operators | 9.2.7 Bitwise operators
cast
9.2 Overview of operators | 9.2.2 Operators of precedence 15 | 9.2.3 Operators of precedence 14

const
9.2 Overview of operators | 20.2.1 const_cast conversions
dynamic
9.2 Overview of operators | 20.2.3 Dynamic conversions
forced
20.2.4 Forced conversions
reinterpret
9.2 Overview of operators
static
9.2 Overview of operators | 20.2.2 Static casts

comma
9.2 Overview of operators | 9.2.12 Comma operator
conditional
9. Operators | 9.2 Overview of operators | 9.2.10 Conditional operator
conversion
9.2.2 Operators of precedence 15 | 9.2.3 Operators of precedence 14
decrement
9.2 Overview of operators
delete
9.2 Overview of operators | 9.2.3 Operators of precedence 14 | 12.3 Deallocation of memory — operator

array
9.2 Overview of operators
object
9.2 Overview of operators

dereferencing (*)
4.6.1 Pointers to variables
direct member selection
9.2 Overview of operators
division
9.2 Overview of operators | 9.2.5 Arithmetic operators
dynamic_cast
22.7 Standard exceptions | 25.2 Operator dynamic_cast
equality
9.2 Overview of operators | 9.2.6 Relational operators and comparisons
exception-throw
9.2 Overview of operators | 9.2.11 Exception-throw operator
function call
9.2 Overview of operators | 9.2.2 Operators of precedence 15 | 19.3 Overloading with methods of

overloading
19.4.3 Function call operator

increment
9.2 Overview of operators
indexing
9.2 Overview of operators | 9.2.2 Operators of precedence 15 | 19.3 Overloading with methods of | 24.3 Examples

overloading
19.4.2 Indexing operator

indirect member selection
9.2 Overview of operators | 19.3 Overloading with methods of

overloading
19.4.4 Overloading indirect member selection

indirection (dereference) (*)
4.6.1 Pointers to variables | 9.2 Overview of operators | 9.2.3 Operators of precedence 14
infix notation
9. Operators
logical AND (&&)

9.2 Overview of operators | 9.2.8 Logical operators
logical NOT (!)
9.2 Overview of operators | 9.2.3 Operators of precedence 14 | 9.2.8 Logical operators
logical OR (||)
9.2 Overview of operators
logical OR (||)
9.2.8 Logical operators
multiplication
9.2 Overview of operators | 9.2.5 Arithmetic operators
new
9.2.3 Operators of precedence 14 | 12.2 Memory allocation — new operator | 22.7 Standard exceptions

array
9.2 Overview of operators
object
9.2 Overview of operators

new (placement)
12.7 Placement new operator
new operator
9.2 Overview of operators
no-op
9.2.3 Operators of precedence 14
nonequality
9.2 Overview of operators | 9.2.6 Relational operators and comparisons
of logical alternative (||)
3.2 Preprocessor directives
of logical conjunction (&&)
3.2 Preprocessor directives
of logical negation (!)
3.2 Preprocessor directives
of scope resolution (::)
7.2 Scope and visibility
overloading
19.1 Introduction

with global function
19.2 Overloading with global functions
with method
19.3 Overloading with methods of

postfix

decrement
9.2.2 Operators of precedence 15
increment
9.2.2 Operators of precedence 15
oveloading
19.3.2 Unary operators

precedence
9.1 Precedence and associativity of | 9.2 Overview of operators
prefix

oveloading
19.3.2 Unary operators

prefix decrement
9.2.3 Operators of precedence 14
prefix increment
9.2.3 Operators of precedence 14
prefix notation
9. Operators
relational
9.2 Overview of operators | 9.2.6 Relational operators and comparisons
remainder (%)
9.2 Overview of operators | 9.2.5 Arithmetic operators
scope

14.5 Static member functions
scope resolution
14.2 Accessibility of class members | 14.3 Fields
scope-resolution
9.2.1 Scope-resolution operators | 14.1 Introduction
short-circuited
9.2.8 Logical operators
sizeof
4.1 Introductory remarks | 9.2 Overview of operators | 9.2.3 Operators of precedence 14
stream-extraction (≫)
2.2.2 Data input
stream-extraction (≫)
16.2 Operators << and >>
stream-insertion (≪)
2.2.1 Outputting information
stream-insertion (≪)
16.2 Operators << and >>
stream-insertion (≪)
19.2.1 Binary operators
subscript
9.2 Overview of operators | 9.2.2 Operators of precedence 15
subscripting
19.3 Overloading with methods of
subtraction
9.2 Overview of operators | 9.2.5 Arithmetic operators
ternary
9. Operators | 9.2.10 Conditional operator
throw
22.1 Throwing exceptions
type identification
9.2 Overview of operators | 9.2.2 Operators of precedence 15
typedef
9.2.3 Operators of precedence 14
typeid
11.14 Function templates | 11.14 Function templates | 18.1 Templates of classes | 22.7 Standard exceptions | 25.1 Operator typeid
unary
9. Operators | 19.3.2 Unary operators

overloading
19.2.2 Unary operators

unary minus
9.2 Overview of operators | 9.2.3 Operators of precedence 14
unary plus
9.2 Overview of operators | 9.2.3 Operators of precedence 14
value construction
9.2 Overview of operators | 9.2.2 Operators of precedence 15

order of evaluation
10.2 Order of evaluation
ordeval.cpp (file)
10.2 Order of evaluation
ostream
16.1 Streams
ostream (class)
2.2.1 Outputting information
ostream_iterator
24.2.1 Algorithms
ostringstream
16.1 Streams | 16.7.2 Internal files represented by
ostrstream
16.1 Streams
out (opening mode)
16.5 Files
out_of_range
17.2.2 Methods and operators | 24.1.1 Vectors

output iterator
24.1.2 Iterators
output stream
16.1 Streams
overflow
17.1.3 Conversion functions
overloading
11.11 Function overloading

exact match
11.11 Function overloading
match after promotion
11.11 Function overloading
match after standard conversion
11.11 Function overloading
match after trivial conversion
11.11 Function overloading
match after user-defined conversion
11.11 Function overloading

overloading operators
19.1 Introduction
overriding
21.1 Fundamentals of inheritance | 21.4 Virtual methods and polymorphism
ovrldeq.cpp (file)
19.4.1 Assignment operator
ovrlderr.cpp (file)
19.4.1 Assignment operator
ovrlskl.cpp (file)
19.4.4 Overloading indirect member selection
ownunique.cpp (file)
19.6.1 Smart pointers of the
padding
13.1 C-structures
pair
24.3 Examples
pairs.cpp (file)
24.3 Examples
pairs.dat (file)
24.3 Examples
parameter
11.1 Introduction | 11.2 Declarations and definitions of
Pascal
see programming languages
peek
16.4.1 Unformatted input
person1.cpp (file)
15.3.1 Copy constructors
person2.cpp (file)
15.3.1 Copy constructors
person3.cpp (file)
15.3.1 Copy constructors
pix.cpp (file)
21.2 Constructors and destructors of
plus operator
9.2.3 Operators of precedence 14
pminmax.cpp (file)
4.6.1 Pointers to variables
pnew.cpp (file)
12.7 Placement new operator
pointer
4.6 Pointers | 4.6.1 Pointers to variables

arithmetic
11.12 Function pointers
arithmetic of
5.3 Pointer arithmetic

generic (void*)
4.6.2 Generic pointers | 5.3 Pointer arithmetic
shared
19.6 Smart pointers | 19.6.2 Smart pointers of the
smart
19.6 Smart pointers
to class member
15.6 Pointers to class members
to function
11. Functions | 11.12 Function pointers
unique
19.6 Smart pointers | 19.6.1 Smart pointers of the

pointers.cpp (file)
4.6.1 Pointers to variables
pointmem.cpp (file)
15.6 Pointers to class members
pointref.cpp (file)
4.7 References
polymorphism
21.4 Virtual methods and polymorphism
polysiz.cpp (file)
21.4 Virtual methods and polymorphism
pop_back
24.1.1 Vectors
pop_front
24.1.3 Operations on collections
position
16.5 Files
postdecrementation
see operator, postfix decrement
postincrementation
see operator, postfix increment
pragma once
3.2 Preprocessor directives
precedence
9.1 Precedence and associativity of | 9.2 Overview of operators
precision
16.3.1 Format flags
predecrementation
see operator, prefix decrement
predic.cpp (file)
24.2.1 Algorithms
predicate
24.2.1 Algorithms
prefix notation
9. Operators
preincrementation
see operator, prefix increment
preplog.cpp (file)
3.2 Preprocessor directives
preprocessing
1.1 Compilers
preprocessor
2.1 Simplest C++ programs | 2.1 Simplest C++ programs | 2.1 Simplest C++ programs | 3.1 What is the preprocessor?
preprocessor directive
3.1 What is the preprocessor? | 3.1 What is the preprocessor?

#define
3.2 Preprocessor directives
defined
3.2 Preprocessor directives
#elif
3.2 Preprocessor directives
#else
3.2 Preprocessor directives
#endif

3.2 Preprocessor directives
#error
3.2 Preprocessor directives
#if
3.2 Preprocessor directives
#ifdef
3.2 Preprocessor directives
#ifndef
3.2 Preprocessor directives
#include
3.2 Preprocessor directives | 23.1.1 Headers and implementation files
#undefine
3.2 Preprocessor directives

preprocessor directives

#include
2.1 Simplest C++ programs

prgr.cpp (file)
25.2 Operator dynamic_cast
primatr.cpp (file)
16.3.2.2 Manipulators with arguments
priority
see precedence
private
14.2 Accessibility of class members | 14.2 Accessibility of class members | 21.1 Fundamentals of inheritance | 21.1 Fundamentals of inheritance
programming languages

Ada
1.1 Compilers | 2.3 Functions
C#
19. Operator overloading. Move semantics
Fortran
1.1 Compilers | 2.3 Functions | 5.5 Multidimensional arrays | 11.2 Declarations and definitions of | 12.2 Memory allocation — new operator
Haskell
19. Operator overloading. Move semantics
Java
2.1 Simplest C++ programs | 4.1 Introductory remarks | 4.2 Integral types | 4.3 Floating-point types | 4.6.1 Pointers to variables | 4.6.2 Generic pointers | 5.2 Array type | 7.2 Scope and visibility | 8.6 Expression statement | 8.7 Conditional statement | 8.10 continue and break statements | 9.2.2 Operators of precedence 15 | 11.1 Introduction | 11.6 Reference arguments | 12. Dynamic memory management | 12.2 Memory allocation — new operator | 14.1 Introduction | 14.2 Accessibility of class members | 14.8 Creating objects | 14.9 Arrays of objects | 15.5 Nested classes | 17.2.2 Methods and operators | 19. Operator overloading. Move semantics | 20.2 Explicit conversions | 21. Inheritance and polymorphism | 21.4 Virtual methods and polymorphism | 21.4 Virtual methods and polymorphism | 22. Exceptions
Lisp
12. Dynamic memory management
Modula
21. Inheritance and polymorphism
Pascal
1.1 Compilers | 4.1 Introductory remarks | 11.2 Declarations and definitions of
PHP
2.1 Simplest C++ programs
Python
2.3 Functions | 11.2 Declarations and definitions of | 12. Dynamic memory management | 12.2 Memory allocation — new operator | 19. Operator overloading. Move semantics | 22. Exceptions
Smalltalk
12. Dynamic memory management | 21. Inheritance and polymorphism

promotion
10.1 Conversions
protected
14.2 Accessibility of class members | 14.2 Accessibility of class members | 21.1 Fundamentals of inheritance | 21.1 Fundamentals of inheritance
prototype

11.2 Declarations and definitions of
public
14.2 Accessibility of class members | 14.2 Accessibility of class members | 21.1 Fundamentals of inheritance | 21.1 Fundamentals of inheritance
pure virtual method
21.5 Abstract classes
push_back
17.2.2 Methods and operators | 24.1.1 Vectors
push_front
24.1.3 Operations on collections
put
16.4.2 Unformatted output
putback
16.4.1 Unformatted input
Python
see programming languages | 11.2 Declarations and definitions of
qualification
14.2 Accessibility of class members
qualified name
14.1 Introduction | 14.2 Accessibility of class members
quick sort
8.4 Null statement
r-reference
19.5 Move semantics
r-value
7.5 L-values
random access iterator
24.1.2 Iterators
ranges.cpp (file)
19.4.2 Indexing operator
rbegin
17.2.2 Methods and operators | 24.1.2 Iterators
rdstate
16.6 Handling stream errors
read
16.4.1 Unformatted input
read.cpp (file)
2.2.2 Data input
readarr.cpp (file)
5.5.2 Arrays of C-strings
realloc
12.5 Memory management in C
recursion
11.8 Recursive functions
recursive function
11.8 Recursive functions
ref.cpp (file)
12.2 Memory allocation — new operator
refer.cpp (file)
4.7 References
reference
4.7 References | 4.7 References | 11.6 Reference arguments

as return value
11.7 Functions returning references
to array
11.6 Reference arguments

register
see variable, register
reinterpret_cast
20.2.4 Forced conversions
release
19.6.1 Smart pointers of the
remainder (%)
9.2.5 Arithmetic operators
remove

24.2.1 Algorithms
remove_if
24.3 Examples
rend
17.2.2 Methods and operators | 24.1.2 Iterators
replace
17.2.2 Methods and operators
reset
19.6.1 Smart pointers of the
resetiosflags (manipulator)
16.3.2.2 Manipulators with arguments
resize
17.2.2 Methods and operators
resour.cpp (file)
22.4 Exceptions in constructors and
return
8.12 return statement
return type
11.2 Declarations and definitions of
revers.cpp (file)
8.9.3 for loop
reverse
24.2.1 Algorithms
reverse_iterator
24.1.2 Iterators
revite.cpp (file)
24.1.2 Iterators
rfind
17.2.2 Methods and operators
right
16.3.1 Format flags
right (manipulator)
16.3.2.1 Parameterless manipulators
rmoveassign.cpp (file)
19.5 Move semantics
roots.cpp (file)
11.12 Function pointers
rotat.cpp (file)
13.1 C-structures
rotateE.cpp (file)
12.6 Functions operating on memory
rotLR.cpp (file)
9.2.7 Bitwise operators
RrefMet.cpp (file)
19.5 Move semantics
RTTI
9.2.2 Operators of precedence 15 | 25. Run-time type identification
rttid.cpp (file)
25.1 Operator typeid
run-time type identification
25. Run-time type identification
rvalue
see r-value
RWfile.cpp (file)
16.7.2 Internal files represented by
s-pointer
19.6.2 Smart pointers of the
scientific
16.3.1 Format flags
scientific (manipulator)
16.3.2.1 Parameterless manipulators
scientific notation
4.3 Floating-point types
scope
7.2 Scope and visibility | 9.2 Overview of operators

class

9.2 Overview of operators
file
7.2 Scope and visibility
global
9.2 Overview of operators
namespace
9.2 Overview of operators

scope-resolution operator
9.2.1 Scope-resolution operators
search.cpp (file)
24.2.1 Algorithms
seek_dir
16.5 Files
seekg
16.5 Files
seekp
16.5 Files
selection statement
8.8 Selection (switch)statement (switch)
semidyn.cpp (file)
12.2 Memory allocation — new operator
set_terminate
22.1 Throwing exceptions
setbase (manipulator)
16.3.2.2 Manipulators with arguments
setf
16.3.1 Format flags
setfill (manipulator)
16.3.2.2 Manipulators with arguments
setiosflags (manipulator)
16.3.2.2 Manipulators with arguments
setprecision (manipulator)
16.3.2.2 Manipulators with arguments
setstate
16.6 Handling stream errors
setw (manipulator)
16.3.2.2 Manipulators with arguments
shallow copy
15.3.1 Copy constructors
shared_ptr
19.6 Smart pointers | 19.6.2 Smart pointers of the
sharedcount.cpp (file)
19.6.2 Smart pointers of the
shareddelete.cpp (file)
19.6.2 Smart pointers of the
short
see type | 4.2 Integral types
short int
4.2 Integral types
short-circuited operator
9.2.8 Logical operators
shortc.cpp (file)
9.2.8 Logical operators
showbase
16.3.1 Format flags
showbase (manipulator)
16.3.2.1 Parameterless manipulators
showpoint
16.3.1 Format flags
showpoint (manipulator)
16.3.2.1 Parameterless manipulators
showpos
16.3.1 Format flags
signature
11.2 Declarations and definitions of | 11.11 Function overloading
simplist.cpp (file)

13.1 C-structures
size
17.2.2 Methods and operators | 24.1.1 Vectors
size_t
12.2 Memory allocation — new operator | 12.5 Memory management in C | 16.7.1 C-strings as internal files | 17.1.1 Functions on C-strings
size_type
17.2 Strings in C++ | 24.1.1 Vectors
sizeof
see operator | 9.2.3 Operators of precedence 14
sizes.cpp (file)
9.2.3 Operators of precedence 14
skipws
16.3.1 Format flags
smart pointer
19.6 Smart pointers
.so
23.1.1 Headers and implementation files
sort
24.2.1 Algorithms | 24.3 Examples
sort.cpp (file)
24.2.1 Algorithms
sortev.cpp (file)
24.2.1 Algorithms
sorting
12.3 Deallocation of memory — operator | 13.1 C-structures | 24.2.1 Algorithms
sortinte.h (file)
23.1.1 Headers and implementation files
sortinteApp.cpp (file)
23.1.1 Headers and implementation files
sortintelmpl.cpp (file)
23.1.1 Headers and implementation files
sorttmplt.cpp (file)
11.14 Function templates
sorword.cpp (file)
17.1.1 Functions on C-strings
source
1.1 Compilers | 16.1 Streams
sstream
16.1 Streams | 16.1 Streams | 16.7.2 Internal files represented by
stack
11.3 Function call | 12.1 Introduction

rewinding
12.1 Introduction | 22.1 Throwing exceptions

stacks.cpp (file)
21.5 Abstract classes
stacksApp.cpp (file)
23.2 Namespaces
stackt.cpp (file)
18.1 Templates of classes
StackTmplt.cpp (file)
13.2 Templates of structures
standard C++11
4.5 Enumeration types | 11.2 Declarations and definitions of | 11.5 Variable-length argument lists | 11.13 Lambda functions
standard input stream
2.2 Simplest In/Out operations | 2.2.2 Data input
standard output stream
2.2 Simplest In/Out operations
Standard Template Library
18. Templates
stat.cpp (file)
7.3.1 Static variables
statement
8.1 Categories of statements

catch
8.13 Exception handling statements
compound
2.1 Simplest C++ programs | 8.5 Compound statement
conditional
8.7 Conditional statement
continue
8.10 continue and break statements
declaration
8.3 Declarations
do-while
8.9.2 do-while loop
exception handling
8.13 Exception handling statements
expression
8.6 Expression statement
for
8.9.3 for loop
goto
8.11 goto statement
iteration
8.9 Iteration statements (loops) | 8.9 Iteration statements (loops)
null
8.4 Null statement
return
8.12 return statement
selection
8.8 Selection (switch)statement (switch)
switch
8.8 Selection (switch)statement (switch)
throw
8.13 Exception handling statements
try
8.13 Exception handling statements
while
8.9.1 while loop

static
see variable, static | 11.9 Static functions | 14.3 Fields
static conversion
20.2.2 Static casts
static function
14.5 Static member functions
static type
21.4 Virtual methods and polymorphism
static variable
7.3.1 Static variables
static.cpp (file)
7.3.1 Static variables
static_cast
20.2.2 Static casts
statmem.cpp (file)
14.3 Fields
stderr
16.1 Streams | 16.1.1 Predefined streams
stdexcept
24.1.1 Vectors
stdin
see standard input stream | 16.1 Streams | 16.1.1 Predefined streams
stdout
see standard output stream | 16.1 Streams | 16.1.1 Predefined streams
STL
18. Templates | 24. The Standard Library (STL)
storage class
7.3 Storage classes

specifier

7.3 Storage classes

str
16.7.2 Internal files represented by
str (function)
16.7.1 C-strings as internal files
strcat
17.1.1 Functions on C-strings
strchr
17.1.1 Functions on C-strings
strcmp
17.1.1 Functions on C-strings
strcoll
17.1.1 Functions on C-strings
strcpy
17.1.1 Functions on C-strings
strcspn
17.1.1 Functions on C-strings
stream
16.1 Streams

input
16.1 Streams
output
16.1 Streams
standard

error
16.1.1 Predefined streams
input
16.1.1 Predefined streams
output
16.1.1 Predefined streams

stream insertion operator
19.2.1 Binary operators
stream state information
16.6 Handling stream errors
streamoff
16.5 Files
streamsize
16.3.1 Format flags | 16.4.1 Unformatted input
string (class)
17.2 Strings in C++

concatenation
17.2.2 Methods and operators
constructor
17.2.1 Constructors
methods
17.2.2 Methods and operators
operators
17.2.2 Methods and operators

string (klasa)
24.1.2 Iterators
string literal
11.13 Lambda functions
string.h
12.6 Functions operating on memory
strinsop.cpp (file)
19.2.1 Binary operators
strlen
17.1.1 Functions on C-strings
strncat
17.1.1 Functions on C-strings
strncmp
17.1.1 Functions on C-strings

strncpy
17.1.1 Functions on C-strings
strong typing
4.1 Introductory remarks
Stroustrup, B.
1.2 Literature | 12.2 Memory allocation — new operator
strpbrk
17.1.1 Functions on C-strings
strrchr
17.1.1 Functions on C-strings
strspn
17.1.1 Functions on C-strings
strstr
17.1.1 Functions on C-strings
strstream
16.1 Streams | 16.1 Streams | 16.7.1 C-strings as internal files
strtod
17.1.3 Conversion functions
strtod.cpp (file)
17.1.3 Conversion functions
strtok
17.1.1 Functions on C-strings
strtol
17.1.3 Conversion functions
strtoul
17.1.3 Conversion functions
structure
13.1 C-structures
students.cpp (file)
8.11 goto statement
subclass
21. Inheritance and polymorphism | 21.1 Fundamentals of inheritance
subobject
21.1 Fundamentals of inheritance
subscript
9.2.2 Operators of precedence 15
substr
17.2.2 Methods and operators
subtraction operator
9.2.5 Arithmetic operators
sumpos.cpp (file)
8.10 continue and break statements
sumuntil.cpp (file)
8.10 continue and break statements
superclass
21.1 Fundamentals of inheritance
surp.cpp (file)
10.1 Conversions
swap
17.2.2 Methods and operators
Swift J.
5.3 Pointer arithmetic
switch.cpp (file)
8.8 Selection (switch)statement (switch)
tabinc.cpp (file)
19.3.2 Unary operators
tail of the list
13.1 C-structures
tellg
16.5 Files
tellp
16.5 Files
template
11.14 Function templates | 11.14 Function templates | 18. Templates | 24. The Standard Library (STL)

concretization
11.14 Function templates

of class
18.1 Templates of classes
of function
11.14 Function templates
of structure
13.2 Templates of structures

temporary
7.5 L-values
term.cpp (file)
22.1 Throwing exceptions
terminate
22.1 Throwing exceptions
ternary operator
9. Operators | 9.2.10 Conditional operator
testInst.cpp (file)
1.1 Compilers
Theaetetus
11.8 Recursive functions
this
14.4 Methods
throw
8.13 Exception handling statements | 22.1 Throwing exceptions
throw operator
22.1 Throwing exceptions
tim.cpp (file)
13.1 C-structures
__TIME__
3.3 Predefined preprocessor macros
time_t
13.1 C-structures
tmpl.cpp (file)
11.14 Function templates
tmplt.cpp (file)
11.14 Function templates
tok.cpp (file)
17.1.1 Functions on C-strings
tokenizer
17.1.1 Functions on C-strings
tolower
17.1.2 Functions operating on characters
toupper
17.1.2 Functions operating on characters
trailing return type
11.2 Declarations and definitions of
translation unit
23.1 Modules of a program
triangs.cpp (file)
15.3.2 Initialization lists
true
4.4 Logical type
trunc (opening mode)
16.5 Files
try
8.13 Exception handling statements | 22.2 Catching exceptions
type

bool
4.4 Logical type
C-structure
13.1 C-structures
char
4.2 Integral types | 4.2 Integral types
char16_t
4.2 Integral types
char32_t
4.2 Integral types

double
4.3 Floating-point types | 4.3 Floating-point types
dynamic
21.4 Virtual methods and polymorphism
enumerated
4.5 Enumeration types
float
4.3 Floating-point types | 4.3 Floating-point types
incomplete
13.1 C-structures
int
4.2 Integral types
int16_t
4.2.1 Useful aliases for integral
int32_t
4.2.1 Useful aliases for integral
int64_t
4.2.1 Useful aliases for integral
int8_t
4.2.1 Useful aliases for integral
long
4.2 Integral types
long double
4.3 Floating-point types | 4.3 Floating-point types
long int
4.2 Integral types
long long
4.2 Integral types
long long int
4.2 Integral types
return
11.2 Declarations and definitions of
short
4.2 Integral types
short int
4.2 Integral types
signed
4.2 Integral types | 4.2 Integral types
size_t
12.2 Memory allocation — new operator
static
21.4 Virtual methods and polymorphism
structure
13.1 C-structures
uint16_t
4.2.1 Useful aliases for integral
uint32_t
4.2.1 Useful aliases for integral
uint64_t
4.2.1 Useful aliases for integral
uint8_t
4.2.1 Useful aliases for integral
unsigned
4.2 Integral types | 4.2 Integral types
void
11.2 Declarations and definitions of
wchar_t
4.2 Integral types

type identification operator
9.2.2 Operators of precedence 15
type modifier
7.4 Type modifiers
type_info
25.1 Operator typeid
typedef
6.2 typedef and using specifiers | 9.2.3 Operators of precedence 14 | 11.12 Function pointers

typedef.cpp (file)
6.2 typedef and using specifiers
typedef1.cpp (file)
6.2 typedef and using specifiers
typeid
11.14 Function templates | 11.14 Function templates | 18.1 Templates of classes | 22.7 Standard exceptions | 25.1 Operator typeid
typeinfo
9.2.3 Operators of precedence 14 | 11.14 Function templates | 25.1 Operator typeid
typename
11.14 Function templates
u-pointer
19.6.1 Smart pointers of the
uint16_t
4.2.1 Useful aliases for integral
uint32_t
4.2.1 Useful aliases for integral
uint64_t
4.2.1 Useful aliases for integral
uint8_t
4.2.1 Useful aliases for integral
unary operator
9. Operators
#undefine
3.2 Preprocessor directives
underflow
17.1.3 Conversion functions
unformatted

input
16.4.1 Unformatted input
output
16.4.2 Unformatted output

unfrd.cpp (file)
16.4.1 Unformatted input
unget
16.4.1 Unformatted input
uniform initializer
4.1 Introductory remarks
union
13.3 Unions
unions.cpp (file)
13.3 Unions
unique_ptr
19.6 Smart pointers | 19.6.1 Smart pointers of the
uniquederiv.cpp (file)
19.6.1 Smart pointers of the
uniquereset.cpp (file)
19.6.1 Smart pointers of the
unitbuf
16.3.1 Format flags
Unix epoch
13.1 C-structures
uns.cpp (file)
13.3 Unions
unsetf
16.3.1 Format flags
upcasting
21.1 Fundamentals of inheritance
uppercase
16.3.1 Format flags
upplowe.cpp (file)
17.1.2 Functions operating on characters
using
6.2 typedef and using specifiers
using declaration

23.2 Namespaces
using namespace
23.2 Namespaces
va_arg
11.5 Variable-length argument lists
va_end
11.5 Variable-length argument lists
va_list
11.5 Variable-length argument lists
va_start
11.5 Variable-length argument lists
validat.cpp (file)
16.6 Handling stream errors
vararg.cpp (file)
11.5 Variable-length argument lists
vardecl.cpp (file)
4.1 Introductory remarks
varepoe.cpp (file)
11.7 Functions returning references
variable
4.1 Introductory remarks | 4.1 Introductory remarks

atomic
7.4.1 Volatile variables
constant
7.4.2 Constants
declaring
4.1 Introductory remarks
defining
4.1 Introductory remarks
exported
7.2 Scope and visibility
extern
7.3.2 External variables
external
7.2 Scope and visibility | 23.1 Modules of a program
global
7.2 Scope and visibility
hidden
7.2 Scope and visibility
initialization
4.1 Introductory remarks
local
7.2 Scope and visibility | 11.3 Function call
static
7.3.1 Static variables | 11.3 Function call
volatile
7.4.1 Volatile variables

variable-length argument list
11.5 Variable-length argument lists
vecpoin.cpp (file)
19.2.1 Binary operators
vecsimple.cpp (file)
5.7 Vectors (std::vector)
vector
5.7 Vectors (std::vector) | 24.1.1 Vectors | 24.1.2 Iterators
virdes.cpp (file)
21.6 Virtual destructors
virt.cpp (file)
21.5 Abstract classes
virtual destructor
21.6 Virtual destructors
virtual function
21.4 Virtual methods and polymorphism
visibility
7.2 Scope and visibility

void
11.2 Declarations and definitions of
void*
see pointer, generic
volatile
see variable, volatile | 7.4.1 Volatile variables | 15.2 Volatile methods
vrpe.cpp (file)
11.6 Reference arguments
wchar_t
4.2 Integral types
what
22.7 Standard exceptions
while
8.9.1 while loop
wide character
4.2 Integral types
width
16.3.1 Format flags
wife.cpp (file)
13.1 C-structures
wild card
1.1.1 Linux
wmani.cpp (file)
16.3.2.1 Parameterless manipulators
write
16.4.2 Unformatted output
writer.cpp (file)
17.2.2 Methods and operators
writers.cpp (file)
13.1 C-structures