# CPP

## TABLE OF CONTENTS

- ▽StringStream
- ▷Statements and control flow
  - ▽Range Based For Loop
  - ▽Jump Statements
- ▷Functions
  - ▽Pass By Reference
  - ▽Inline Functions
  - ▽Default Values In The Parameters
  - ▽Declaration Of Functions
- ▷Overloads and templates
  - ▽Overloaded Functions
  - ▽Function Template
  - ▽Non Type Template Arguments
- ▷Name visibility
  - ▽Namesapces
  - ▽Using Keyword
  - ▽Namespace Aliasing
  - ▽The std Namespace
  - ▽Storage Classes
- ▷Arrays
  - ▽Arrays As Parameters
  - ▽Library Arrays
- ▷Character sequences
  - ▽Initialization OF NULL Terminated Character Sequences
  - ▽Strings and null-terminated character sequences
- ▷Pointers
  - ▽Declaration
  - ▽Pointer & Array
  - ▽Pointer Arithmetic
  - ▽Pointer & Constant
  - ▽Pointers & String Literals
  - ▽Pointers To Pointers
  - ▽Void Pointers
  - ▽Invalid & Null Pointers
  - ▽Pointers To Functions

- ▷Dynamic memory
  - ▽Operators new and new[]
  - ▽Operations delete & delete[]
  - ▽Dynamic Memory In C
- ▷Data structures
  - ▽Data Structures
  - ▽Pointers To Structures
  - ▽Nesting Structures
- ▷Other data types
  - ▽Type Aliases (typedef / using)
  - ▽Unions
  - ▽Anonymous Unions
  - ▽Enumerated Types (enum)
  - ▽Enumerated Types With enum Classes
- ▷Classes - i
  - ▽Constructors
  - ▽Overloading Constructors
  - ▽Uniform Initialization
  - ▽Member Initialization In Constructors
  - ▽Pointers To Classes
  - ▽Interpretations
  - ▽Classed Defined With Struct & Union Keywords
- ▷Classes – ii
  - ▽Overloading Operators
  - ▽The Keyword this
  - ▽Static Members
  - ▽Const Member Functions
  - ▽Class Templates
  - ▽Template Specialization
- ▷Special members
  - ▽Default Constructor
  - ▽Destructor
  - ▽Copy Constructor
  - ▽Copy Assignment
  - ▽Move Constructor & Assignment

- ▽Implicit Members
- ▷Friendship and inheritance
  - ▽Friend Function
  - ▽Friend Classes
  - ▽Inheritance Between Classes
  - ▽What Is Inherited From The Base Class?
  - ▽Multiple Inheritence
- ▷Polymorphism
  - ▽Pointer To Base Class
  - ▽Virtual Members
  - ▽Abstract Base Classes
- ▷Type conversions
  - ▽Implicit Conversion
  - ▽Implicit Conversion With Classes
  - ▽Keyword Explicit
  - ▽Type-Casting
  - ▽Dynamic Cast
  - ▽Static_cast
  - ▽Reinterpret_cast
  - ▽Const_cast
  - ▽typeid
  - ▽<cstdint> / (stdint.h)
    - ▸Types
    - ▸Macros
    - ▸Limits of other types
    - ▸Functions like macros
- ▷Exceptions
  - ▽Exception Specifications
  - ▽Standard Exceptions
- ▷Preprocessor directives
  - ▽Macro Definitions
  - ▽Conditional Inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)
  - ▽Line Control (#line)
  - ▽Error Directive (#error)

- ▽Source File Inclusion (#include)
- ▽Pragma Directive (#pragma)
- ▽Predefined macro names

▷Input/output with files

- ▽Open A File
- ▽Closing A File
- ▽Text Files
- ▽Checking State Flags
- ▽Get and Put Stream Positioning
  - ▸tellg() and tellp()
  - ▸seekg() and seekp()
- ▽Binary Files
- ▽Buffers & Synchronization

# INTRODUCTION

- Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (;).
-

```cpp
#include <iostream>
using namespace std;
int main() {
    cout<<"Hello World"<<endl;//Unqualified cout
    std::cout<<"Hello World"<<endl;//Qualified directly within std namespace
    return 0;
}
```

- cout is part of the standard library, and all the elements in the standard C++ library are declared within what is called a namespace: the namespace std.
- Explicit qualification is the only way to guarantee that name collisions never happen.

# VARIABLES AND TYPES

## ▷ Identifiers

▽ A sequence of one or more letters, digits, or underscore characters (_)

▽ Identifiers shall always begin with a letter. They can also begin with an underline character (_), but such identifiers are -on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere.

▽ List of identifiers :

alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq

▽ The c++ is a case sensitive language $\Longrightarrow$ Example $\neq$ example $\neq$ EXAMPLE

## ▷ Fundamental Data-Types

▽ Character data type : They can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.

▽ Numerical integer type : They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be signed or unsigned, depending on whether they support negative values or not.

▽ Floating-point type : They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.

▽ Boolean type : The boolean type, known in C++ as bool, can only represent one of two states, true or false.

▽

Here is the complete list of fundamental types in C++:

| Group | Type names* | Notes on size / precision |
|---|---|---|
| Character types | `char` | Exactly one byte in size. At least 8 bits. |
| | `char16_t` | Not smaller than `char`. At least 16 bits. |
| | `char32_t` | Not smaller than `char16_t`. At least 32 bits. |
| | `wchar_t` | Can represent the largest supported character set. |
| Integer types (signed) | `signed char` | Same size as `char`. At least 8 bits. |
| | *`signed`* `short` *`int`* | Not smaller than `char`. At least 16 bits. |
| | *`signed`* `int` | Not smaller than `short`. At least 16 bits. |
| | *`signed`* `long` *`int`* | Not smaller than `int`. At least 32 bits. |
| | *`signed`* `long long` *`int`* | Not smaller than `long`. At least 64 bits. |
| Integer types (unsigned) | `unsigned char` | (same size as their signed counterparts) |
| | `unsigned short` *`int`* | |
| | `unsigned` *`int`* | |
| | `unsigned long` *`int`* | |
| | `unsigned long long` *`int`* | |
| Floating-point types | `float` | |
| | `double` | Precision not less than `float` |
| | `long double` | Precision not less than `double` |
| Boolean type | `bool` | |
| Void type | `void` | no storage |
| Null pointer | `decltype(nullptr)` | |

▽

```
Char data types :
Size of char 1 bytes
Size of char16_t 2 bytes
Size of char32_t 4 bytes
Size of wchar_t 4 bytes

Size of Signed Int data types :
Size of signed char 1 bytes
Size of signed short int 2 bytes
Size of signed int 4 bytes
Size of signed long int 8 bytes
Size of singed long long int 8 bytes

Size of Unsigned Int data types :
Size of unsigned char 1 bytes
Size of unsigned short int 2 bytes
Size of unsigned int 4 bytes
Size of unsigned long int 8 bytes
Size of unsigned long long int 8 bytes

Size of Floating-point data types :
Size of float 4 bytes
Size of double 8 bytes
Size of long double 16 bytes

Size of boolean data type :
Size of bool 1 bytes

Size of void type
Size of void 1 bytes

Size of decltype(nullptr) :
Size of decltype(nullptr) 8 bytes
```

▽ * The names of certain integer types can be abbreviated without their signed and int components - only the part not in italics is required to identify the type, the part in italics is optional. I.e., signed short int can be abbreviated as signed short, short int, or simply short; they all identify the same fundamental type.

▽ Only char has the exact size of 1 byte. Other fundamental data types do not have an exact size. They have a minimum and atmost sizes.This does not mean that these types are of an undetermined

size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run.

▽

| Size | Unique representable values | Notes |
|---|---|---|
| 8-bit | 256 | = $2^8$ |
| 16-bit | 65 536 | = $2^{16}$ |
| 32-bit | 4 294 967 296 | = $2^{32}$ (~4 billion) |
| 64-bit | 18 446 744 073 709 551 616 | = $2^{64}$ (~18 billion billion) |

▽ The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: void, which identifies the lack of type; and the type nullptr, which is a special type of pointer.

## ▷ Initialization Of Variables

▽ In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

- The first one, known as c-like initialization (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:
  type identifier = initial_value;
  For example,
  int x = 0;
- A second method, known as constructor initialization (introduced by the C++ language), encloses the initial value between parentheses (()):
  type identifier (initial_value);
  For example:
  int x (0);
- A third method, known as uniform initialization, similar to the above, but using curly braces ({}) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):
  type identifier {initial_value};
  For example: int x {0};

## ▷ Type deduction : auto and decltype

▽ When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use auto as the type specifier for the variable:

auto x=33;

▽ Variables that are not initialized can also make use of type deduction with the decltype specifier:

int foo=0;

decltype(foo) bar;

## ▷ Introduction To Strings

▽ It is a compound type.

▽ Defined in header <string>

▽

```cpp
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

▽ Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and change its value during execution.

▷ Note: inserting the endl manipulator ends the line (printing a newline character and flushing the stream).

# CONSTANTS

## ▷ Literals

▽ They are used to express particular values within the source code of a program.

▽ Eg. int a=5; //here 5 is a literal constant

▽ Literal constants can be classified into: integer, floating-point, characters, strings, Boolean, pointers, and user-defined literals.

### ▽ Interger Numerals

- ▸ They are a simple succession of digits representing a whole number in decimal base.
- ▸ In addition to decimal numbers (those that most of us use every day), C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16) as literal constants. For octal literals, the digits are preceded with a 0 (zero) character. And for hexadecimal, they are preceded by the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

  75 //decimal

  0113 //octal

  0×4b //hexadecimal
- ▸ These literal constants have a type, just like variables. By default, integer literals are of type int. However, certain suffixes may be appended to an integer literal to specify a different integer type:

  u or U for unsigned

  l or L for long

  ll or LL for long long
- ▸ Unsigned may be combined with any of the other two in any order to form unsigned long or unsigned long long.
- ▸

```
1 75          // int
2 75u         // unsigned int
3 75l         // long
4 75ul        // unsigned long
5 75lu        // unsigned long
```

▽ Floating Point Numerals

- ▶ They express real values, with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character:

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

- ▶ The default type for floating-point literals is double. Floating-point literals of type float or long double can be specified by adding one of the following suffixes:
  f or F for float
  l or L for long double
  eg. 3.14158L //long double
  6.02e23f //float

▷ Character And String Literals

▽ character literal : ‘x’
string literal : “hello”

▽ Character and string literals can also represent special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). These special characters are all of them preceded by a backslash character (\).

| Escape code | Description |
|---|---|
| \n | newline |
| \r | carriage return |
| \t | tab |
| \v | vertical tab |
| \b | backspace |
| \f | form feed (page feed) |
| \a | alert (beep) |
| \' | single quote (') |
| \" | double quote (") |
| \? | question mark (?) |
| \\ | backslash (\) |

▽ Internally, computers represent characters as numerical codes: most typically, they use one extension of the ASCII character encoding system

The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are representable characters:

| * | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | TAB | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 |  | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ |  |

▽ Because most systems nowadays work with 8bit bytes, which can represent 256 different values, in addition to the 128 standard ASCII codes there are other 128 that are known as extended ASCII, which are platform- and locale-dependent. So there is more than one extended ASCII character set.

**Standard 7-bit ASCII Table** — theascii.com

| Dec | Hex | Oct | Binary | Char | Description | Dec | Hex | Oct | Binary | Char | Dec | Hex | Oct | Binary | Char | Dec | Hex | Oct | Binary | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | NUL | Null character | 32 | 20 | 40 | 100000 | space | 64 | 40 | 100 | 1000000 | @ | 96 | 60 | 140 | 1100000 | ` |
| 1 | 1 | 1 | 1 | SOH | Start of header | 33 | 21 | 41 | 100001 | ! | 65 | 41 | 101 | 1000001 | A | 97 | 61 | 141 | 1100001 | a |
| 2 | 2 | 2 | 10 | STX | Start of text | 34 | 22 | 42 | 100010 | " | 66 | 42 | 102 | 1000010 | B | 98 | 62 | 142 | 1100010 | b |
| 3 | 3 | 3 | 11 | ETX | End of text | 35 | 23 | 43 | 100011 | # | 67 | 43 | 103 | 1000011 | C | 99 | 63 | 143 | 1100011 | c |
| 4 | 4 | 4 | 100 | EOT | End of transmission | 36 | 24 | 44 | 100100 | $ | 68 | 44 | 104 | 1000100 | D | 100 | 64 | 144 | 1100100 | d |
| 5 | 5 | 5 | 101 | ENQ | Enquiry | 37 | 25 | 45 | 100101 | % | 69 | 45 | 105 | 1000101 | E | 101 | 65 | 145 | 1100101 | e |
| 6 | 6 | 6 | 110 | ACK | Acknowledge | 38 | 26 | 46 | 100110 | & | 70 | 46 | 106 | 1000110 | F | 102 | 66 | 146 | 1100110 | f |
| 7 | 7 | 7 | 111 | BEL | Bell ring | 39 | 27 | 47 | 100111 | ' | 71 | 47 | 107 | 1000111 | G | 103 | 67 | 147 | 1100111 | g |
| 8 | 8 | 10 | 1000 | BS | Backspace | 40 | 28 | 50 | 101000 | ( | 72 | 48 | 110 | 1001000 | H | 104 | 68 | 150 | 1101000 | h |
| 9 | 9 | 11 | 1001 | HT | Horizontal tab | 41 | 29 | 51 | 101001 | ) | 73 | 49 | 111 | 1001001 | I | 105 | 69 | 151 | 1101001 | i |
| 10 | 0A | 12 | 1010 | LF | Line feed | 42 | 2A | 52 | 101010 | * | 74 | 4A | 112 | 1001010 | J | 106 | 6A | 152 | 1101010 | j |
| 11 | 0B | 13 | 1011 | VT | Vertical tab | 43 | 2B | 53 | 101011 | + | 75 | 4B | 113 | 1001011 | K | 107 | 6B | 153 | 1101011 | k |
| 12 | 0C | 14 | 1100 | FF | Form feed | 44 | 2C | 54 | 101100 | , | 76 | 4C | 114 | 1001100 | L | 108 | 6C | 154 | 1101100 | l |
| 13 | 0D | 15 | 1101 | CR | Carriage return | 45 | 2D | 55 | 101101 | – | 77 | 4D | 115 | 1001101 | M | 109 | 6D | 155 | 1101101 | m |
| 14 | 0E | 16 | 1110 | SO | Shift out | 46 | 2E | 56 | 101110 | . | 78 | 4E | 116 | 1001110 | N | 110 | 6E | 156 | 1101110 | n |
| 15 | 0F | 17 | 1111 | SI | Shift in | 47 | 2F | 57 | 101111 | / | 79 | 4F | 117 | 1001111 | O | 111 | 6F | 157 | 1101111 | o |
| 16 | 10 | 20 | 10000 | DLE | Data link escape | 48 | 30 | 60 | 110000 | 0 | 80 | 50 | 120 | 1010000 | P | 112 | 70 | 160 | 1110000 | p |
| 17 | 11 | 21 | 10001 | DC1 | Device control 1 | 49 | 31 | 61 | 110001 | 1 | 81 | 51 | 121 | 1010001 | Q | 113 | 71 | 161 | 1110001 | q |
| 18 | 12 | 22 | 10010 | DC2 | Device control 2 | 50 | 32 | 62 | 110010 | 2 | 82 | 52 | 122 | 1010010 | R | 114 | 72 | 162 | 1110010 | r |
| 19 | 13 | 23 | 10011 | DC3 | Device control 3 | 51 | 33 | 63 | 110011 | 3 | 83 | 53 | 123 | 1010011 | S | 115 | 73 | 163 | 1110011 | s |
| 20 | 14 | 24 | 10100 | DC4 | Device control 4 | 52 | 34 | 64 | 110100 | 4 | 84 | 54 | 124 | 1010100 | T | 116 | 74 | 164 | 1110100 | t |
| 21 | 15 | 25 | 10101 | NAK | Negative acknowledge | 53 | 35 | 65 | 110101 | 5 | 85 | 55 | 125 | 1010101 | U | 117 | 75 | 165 | 1110101 | u |
| 22 | 16 | 26 | 10110 | SYN | Synchronize | 54 | 36 | 66 | 110110 | 6 | 86 | 56 | 126 | 1010110 | V | 118 | 76 | 166 | 1110110 | v |
| 23 | 17 | 27 | 10111 | ETB | End transmission block | 55 | 37 | 67 | 110111 | 7 | 87 | 57 | 127 | 1010111 | W | 119 | 77 | 167 | 1110111 | w |
| 24 | 18 | 30 | 11000 | CAN | Cancel | 56 | 38 | 70 | 111000 | 8 | 88 | 58 | 130 | 1011000 | X | 120 | 78 | 170 | 1111000 | x |
| 25 | 19 | 31 | 11001 | EM | End of medium | 57 | 39 | 71 | 111001 | 9 | 89 | 59 | 131 | 1011001 | Y | 121 | 79 | 171 | 1111001 | y |
| 26 | 1A | 32 | 11010 | SUB | Substitute | 58 | 3A | 72 | 111010 | : | 90 | 5A | 132 | 1011010 | Z | 122 | 7A | 172 | 1111010 | z |
| 27 | 1B | 33 | 11011 | ESC | Escape | 59 | 3B | 73 | 111011 | ; | 91 | 5B | 133 | 1011011 | [ | 123 | 7B | 173 | 1111011 | { |
| 28 | 1C | 34 | 11100 | FS | File separator | 60 | 3C | 74 | 111100 | < | 92 | 5C | 134 | 1011100 | \ | 124 | 7C | 174 | 1111100 | \| |
| 29 | 1D | 35 | 11101 | GS | Group separator | 61 | 3D | 75 | 111101 | = | 93 | 5D | 135 | 1011101 | ] | 125 | 7D | 175 | 1111101 | } |
| 30 | 1E | 36 | 11110 | RS | Record separator | 62 | 3E | 76 | 111110 | > | 94 | 5E | 136 | 1011110 | ^ | 126 | 7E | 176 | 1111110 | ~ |
| 31 | 1F | 37 | 11111 | US | Unit separator | 63 | 3F | 77 | 111111 | ? | 95 | 5F | 137 | 1011111 | _ | 127 | 7F | 177 | 1111111 | DEL |

▽ Characters can also be represented in literals using its numerical code by writing a backslash character (\) followed by the code expressed as an octal (base-8) or hexadecimal (base-16) number. For an octal value, the backslash is followed directly by the digits; while for hexadecimal, an x character is inserted

between the backslash and the hexadecimal digits themselves (for example: \x20 or \x4A).

▽ Several string literals can be concatenated to form a single string literal simply by separating them by one or more blank spaces, including tabs, newlines, and other valid blank characters. For example:

"this forms" "a single"  " string "
"of characters"

The above is a string literal equivalent to:

"this formsa single string of characters"

▽ Some programmers also use a trick to include long string literals in multiple lines: In C++, a backslash (\) at the end of line is considered a line-continuation character that merges both that line and the next into a single line. Therefore the following code:

x = "string expressed in \
two lines"

is equivalent to:

x = "string expressed in two lines"

▽ All the character literals and string literals described above are made of characters of type char. A different character type can be specified by using one of the following prefixes:

u for char16_t
U for char32_t
L for wchar_t

▽ Here the prefixes are case sensitive unlike others.

▽ Two additional prefixes also exist :

u8 for encoding the executable in utf8
R for raw string

▽ In raw strings, backslashes and single and double quotes are all valid characters; the content of the literal is delimited by an initial R"sequence( and a final )sequence", where sequence is any sequence of characters (including an empty sequence). The content of the string is what lies inside the parenthesis, ignoring the delimiting sequence itself. For example:

R"(string with \backslash)"
R"&%$(string with \backslash)&%$"

### ▷ Other Literals

▽ Three keyword literals exist in C++: true, false and nullptr:
true and false are the two possible values for variables of type bool.
nullptr is the null pointer value.

### ▷ Typed Constant Expressions

▽ Sometimes, it is just convenient to give a name to a constant value:
const double pi = 3.1415926;
const char tab='\t'

### ▷ Preprocessor Definitions

▽ Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:
#define identifier replacement

▽ This replacement is performed by the preprocessor, and happens before the program is compiled, thus causing a sort of blind replacement: the validity of the types or syntax involved is not checked in any way.

# OPERATORS

## ▷ Assignment

▽ Assignment operations are expressions that can be evaluated. That means that the assignment itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

```
y = 2 + (x = 5);
```

In this expression, y is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

```
x = 5;
y = 2 + x;
```

▽ The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns 5 to the all three variables: x, y and z; always from right-to-left.

## ▷ Arithmetic Operators

▽ The five arithmetical operations supported by C++ are:
addition, subtration, multiplication, division, modulo

## ▷ Compound Assignment

▽ `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `%=` `^=` `|=`

## ▷ Increment & Decrement

▽ Equivalent to +=1 and -=1

▽ In the case that the increase operator is used as a prefix (++x) of the value, the expression evaluates to the final value of x, once it is already increased. On the other hand, in case that it is used as a suffix (x++), the value is also increased, but the expression evaluates to the value that x had before being increased.

## ▷ Relational & Comparison Operators

▽ The result of such an operation is either true or false (i.e., a Boolean value).

▽ == > < >= <= !=

▽ Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables.

## ▷ Logical Operators

▽ ! && ||

▽ When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in the last example ((5==5)||(3>6)), C++ evaluates first whether 5==5 is true, and if so, it never checks whether 3>6 is true or not. This is known as short-circuit evaluation, and works like this for these operators:

| operator | short-circuit |
|---|---|
| && | if the left-hand side expression is `false`, the combined result is `false` (the right-hand side expression is never evaluated). |
| \|\| | if the left-hand side expression is `true`, the combined result is `true` (the right-hand side expression is never evaluated). |

▽ This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ( (i<10) && (++i<n) ) { /* ... */ }   // note that the condition increments i
```

## ▷ Conditional Ternary Operators

▽ `condition ? result1 : result2 //evaluates to result1 if condition is true`

## ▷ Comma Operator

▽ The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

## ▷ Bitwise Operators

▽ & | ^ ~ << >>

▽ Bitwise operators modify variables considering the bit patterns that represent the values they store.

▽

| operator | asm equivalent | description |
|---|---|---|
| & | AND | Bitwise AND |
| \| | OR | Bitwise inclusive OR |
| ^ | XOR | Bitwise exclusive OR |
| ~ | NOT | Unary complement (bit inversion) |
| << | SHL | Shift bits left |
| >> | SHR | Shift bits right |

## ▷ Explicit Type Casting Operator

▽ Precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

▽ Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

## ▷ sizeof

▽ This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object.

▽ The value returned by sizeof is a compile-time constant, so it is always determined before program execution.

## ▷ Precedence Of Operators

▽

From greatest to smallest priority, C++ operators are evaluated in the following order:

| Level | Precedence group | Operator | Description | Grouping |
|---|---|---|---|---|
| 1 | Scope | `::` | scope qualifier | Left-to-right |
| 2 | Postfix (unary) | `++ --` | postfix increment / decrement | Left-to-right |
| | | `()` | functional forms | |
| | | `[]` | subscript | |
| | | `. ->` | member access | |
| 3 | Prefix (unary) | `++ --` | prefix increment / decrement | Right-to-left |
| | | `~ !` | bitwise NOT / logical NOT | |
| | | `+ -` | unary prefix | |
| | | `& *` | reference / dereference | |
| | | `new delete` | allocation / deallocation | |
| | | `sizeof` | parameter pack | |
| | | `(type)` | C-style type-casting | |
| 4 | Pointer-to-member | `.* ->*` | access pointer | Left-to-right |
| 5 | Arithmetic: scaling | `* / %` | multiply, divide, modulo | Left-to-right |
| 6 | Arithmetic: addition | `+ -` | addition, subtraction | Left-to-right |
| 7 | Bitwise shift | `<< >>` | shift left, shift right | Left-to-right |
| 8 | Relational | `< > <= >=` | comparison operators | Left-to-right |
| 9 | Equality | `== !=` | equality / inequality | Left-to-right |
| 10 | And | `&` | bitwise AND | Left-to-right |
| 11 | Exclusive or | `^` | bitwise XOR | Left-to-right |
| 12 | Inclusive or | `\|` | bitwise OR | Left-to-right |
| 13 | Conjunction | `&&` | logical AND | Left-to-right |
| 14 | Disjunction | `\|\|` | logical OR | Left-to-right |
| 15 | Assignment-level expressions | `= *= /= %= += -= >>= <<= &= ^= \|=` | assignment / compound assignment | Right-to-left |
| | | `?:` | conditional operator | |
| 16 | Sequencing | `,` | comma separator | Left-to-right |

▽ When an expression has two operators with the same precedence level, grouping determines which one is evaluated first: either left-to-right or right-to-left.

▽ Parts of the expressions can be enclosed in parenthesis to override this precedence order, or to make explicitly clear the intended effect.

# BASIC INPUT / OUTPUT

▷ C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen, the keyboard or a file. A stream is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

▷ Streams provided by standard library:

cin : standard input

cout : standard output

cerr : standard error(output)

clog : standard logging(output)

▷ Standard Output (cout)

▽ Its a c++ stream object.

▽ endl : Is a manipulator

▽ \n : Is a character

▽ The endl manipulator produces a newline character, exactly as the insertion of '\n' does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly fully buffered streams, and cout is (generally) not a fully buffered stream. Still, it is generally a good idea to use endl only when flushing the stream would be a feature and '\n' when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.

▷ Standard Input (cin)

▽ The extraction operation on cin uses the type of the variable after the >> operator to determine how it interprets the characters read from the input; if it is an integer, the format

expected is a series of digits, if a string a sequence of characters, etc.

▽ The extraction operation fails if the type of value inserted by the user does not match the type specified by the programmer. And this, by default, lets the program continue without setting a value for variable i, producing undetermined results if the value of i is used later.
Stringstreams can be used to keep a check.

## ▷ Cin and Strings

▽ The extraction operator can be used on cin to get strings of characters in the same way as with fundamental data types:
string mystring;
cin >> mystring;

▽ However, cin extraction always considers spaces (whitespaces, tabs, new-line ... ) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

▽ getline(cin,string_variable_name) : To get an entire line from cin, there exists a function, called getline, that takes the stream (cin) as first argument, and the string variable as second.

## ▷ StringStream

▽ The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on cin and cout. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint; //numerical value 1204 is stored in myint variable
```

▽

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
  string mystr;
  float price=0;
  int quantity=0;

  cout << "Enter price: ";
  getline (cin,mystr);
  stringstream(mystr) >> price;
  cout << "Enter quantity: ";
  getline (cin,mystr);
  stringstream(mystr) >> quantity;
  cout << "Total price: " << price*quantity << endl;
  return 0;
}
```

```
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

In this example, we acquire numeric values from the standard input indirectly: Instead of extracting numeric values directly from cin, we get lines from it into a string object (mystr), and then we extract the values from this string into the variables price and quantity. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

▽ With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user expects, and at the same time gaining more control over the transformation of its content into useful data by the program.

▷ A for loop with no condition is equivalent to a loop with true as condition (i.e., an infinite loop). Eg. for(int i=0; ;i++){ s1;}

▷ We can use multiple expression in the for loop. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```
for ( n=0, i=100 ; n≠i ; ++n, --i ){
 // whatever here ...
}
```

▷ Range Based For Loop

▽ The for-loop has another syntax, which is used exclusively with ranges:

for ( declaration : range ) statement;

▽ This kind of for loop iterates over all the elements in range, where declaration declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end.

```
1  // range-based for loop                                    [H][e][l][l][o][!]
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main ()
7  {
8    string str {"Hello!"};
9    for (char c : str)
10   {
11     cout << "[" << c << "]";
12   }
13   cout << '\n';
14 }
```

▽ Range based loops usually also make use of type deduction for the type of the elements with auto. Typically, the range-based loop above can also be written as:

```
1 for (auto c : str)
2   cout << "[" << c << "]";
```

## ▷ Jump Statements

▽ Break : break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.

▽ Continue : The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

▽ The Goto Statement : goto allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

```
// goto loop example
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
mylabel:
  cout << n << ", ";
  n--;
  if (n>0) goto mylabel;
  cout << "liftoff!\n";
}
```

▽ Switch : The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating if-else statements, but limited to constant expressions. It uses labels instead of blocks. Its most typical syntax is:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

# FUNCTIONS

▷ If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement: return 0; //this applies only to main

▷ The values for main that are guaranteed to be interpreted in the same way on all platforms are:

| value | description |
|---|---|
| 0 | The program was successful |
| EXIT_SUCCESS | The program was successful (same as above).<br>This value is defined in header <cstdlib>. |
| EXIT_FAILURE | The program failed.<br>This value is defined in header <cstdlib>. |

EXIT_SUCCESS & EXIT_FAILURE are macros that expand to system-dependent integral expression that, when used with the function exit {void exit(int status)} signifies that the application succeeded or failed. Eg. exit(EXIT_SUCCESS)

▷ Pass By Refreence

▽

```
// passing parameters by reference                    x=2, y=6, z=14
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

▽ To gain access to its arguments, the function declares its parameters as references.

▽ In C++, references are indicated with an ampersand (&) following the parameter type.

▽ The parameters of the function becomes the aliases of the arguments passed by the calling function.

▽ On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.
The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

▽

```
string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

## ▷ Inline Functions

▽ Calling a function generally causes a certain overhead (stacking arguments, jumps, etc ... ), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

▽ Preceding a function declaration with the inline specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

▽

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

▽ inline is only specified in the function declaration, not when it is called.

▽ This specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise.

## ▷ Default Values In The Parameters

▽ In C++, functions can also have optional parameters, for which no arguments are required in the call, in such a way that, for example, a function with three parameters may be called with only two. For this, the function shall include a default value for its last parameter, which is used by the function when called with fewer arguments. For example:

▽

```cpp
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

## ▷ Declaration Of Functions

▽ The parameter list does not need to include the parameter names, but only their types. Parameter names can nevertheless be specified, but they are optional, and do not need to necessarily match those in the function definition.

▽

```cpp
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int x);
void even (int x);

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

```
Please, enter number (0 to exit): 9
It is odd.
Please, enter number (0 to exit): 6
It is even.
Please, enter number (0 to exit): 1030
It is even.
Please, enter number (0 to exit): 0
It is even.
```

# OVERLOADS AND TEMPLATES

## ▷ Overloaded Functions

▽ In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```cpp
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

▽ Two functions with the same name are generally expected to have -at least- a similar behavior.

▽ Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

## ▷ Function Template

▽ Defining a function template follows the same syntax as a regular function, except that it is preceded by the template keyword and a series of template parameters enclosed in angle-brackets <>:
template <template-parameters> function-declaration

▽ The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the class or typename keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```cpp
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

▽ It makes no difference whether the generic type is specified with keyword class or keyword typename in the template argument list (they are 100% synonyms in template declarations).

▽ The generic type “sometype” can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

▽ Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the function template, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:
name <template-arguments> (function-arguments)

```
x = sum<int>(10,20);
```

▽ When the generic type T is used as a parameter for a generic function, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets.
Eg. x=sum(10,20) //T is deduced as int by compiler
x=sum(10.5,20.5) //T is deduced as double by compiler

▽ Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

```
// function templates                          x and y are equal
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

## ▷ Non Type Template Arguments

▽ The template parameters can not only include types introduced by class or typename, but can also include expressions of a particular type:

```cpp
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

```
20
30
```

- The second argument of the fixed_multiply function template is of type int. It just looks like a regular function parameter, and can actually be used just like one.
- But there exists a major difference: the value of template parameters is determined on compile-time to generate a different instantiation of the function fixed_multiply, and thus the value of that argument is never passed during runtime: The two calls to fixed_multiply in main essentially call two versions of the function: one that always multiplies by two, and one that always multiplies by three. For that same reason, the second template argument needs to be a constant expression (it cannot be passed a variable).

# NAME VISIBILITY

## ▷ Namesapces

▽ But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

▽ Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.

▽ The syntax to declare a namespaces is:

```
namespace identifier
{
  named_entities
}
```

▽ Where identifier is any valid identifier and named_entities is the set of variables, types and functions that are included within the namespace. For example:

```
1 namespace myNamespace
2 {
3   int a, b;
4 }
```

▽ These variables can be accessed from within their namespace normally, with their identifier (either a or b), but if accessed from outside the myNamespace namespace they have to be properly qualified with the scope operator ::. For example, to access the previous variables from outside myNamespace they should be qualified like:

```
1 myNamespace::a
2 myNamespace::b
```

▽

```
1  // namespaces                          5
2  #include <iostream>                    6.2832
3  using namespace std;                   3.1416
4
5  namespace foo
6  {
7    int value() { return 5; }
8  }
9
10 namespace bar
11 {
12   const double pi = 3.1416;
13   double value() { return 2*pi; }
14 }
15
16 int main () {
17   cout << foo::value() << '\n';
18   cout << bar::value() << '\n';
19   cout << bar::pi << '\n';
20   return 0;
21 }
```

▽ Namespaces can be split: Two segments of a code can be declared in the same namespace:

```
namespace foo { int a; }
namespace bar { int b; }
namespace foo { int c; }
```

## ▷ Using Keyword

▽ The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example :

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

```
5
2.7183
10
3.1416
```

▽ The keyword using can also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```
5
10
3.1416
2.7183
```

▽ using and using namespace have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

```
5
3.1416
```

▷ Namespace Aliasing

▽ Existing namespaces can be aliased with new names, with the following syntax:

namespace new_name = current_name;

▷ The std Namespace

▽ All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the std namespace.

▽ Whether the elements in the std namespace are introduced with using declarations or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

▷ Storage Classes

▽ The storage for variables with global or namespace scope is allocated for the entire duration of the program. This is known as static storage, and it contrasts with the storage for local variables (those declared within a block). These use what is known as automatic storage. The storage for local variables is only available during the block in which they are declared; after that,

that same storage may be used for a local variable of some other function, or used otherwise.

▽ Global variables are automatically initialized to 0.

# ARRAYS

▷ Declaration : type name [elements];

▷ The elements field within square brackets [], representing the number of elements in the array, must be a constant expression, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

▷ Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

▷ Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension.

▷ Arrays As Parameters

▽ In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument.

▽ To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:
void procedure(int arg[])

▽

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

Edit & Run

▽ In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:
base_type[][depth][depth]
Eg. void procedure(int myarray[][2][3])

- Notice that the first brackets [] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.
- In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer.

## Library Arrays

- The arrays explained above are directly implemented as a language feature, inherited from the C language. They are a great feature, but by restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization.
- To overcome some of these issues with language built-in arrays, C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header <array>.
- They operate in a similar way to built-in arrays, except that they allow being copied (an actually expensive operation that copies the entire block of memory, and thus to use with care) and decay into pointers only when explicitly told to do so (by means of its member data).
-

**language built-in array**

```
#include <iostream>

using namespace std;

int main()
{
  int myarray[3] = {10,20,30};

  for (int i=0; i<3; ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

**container library array**

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

int main()
{
  array<int,3> myarray {10,20,30};

  for (int i=0; i<myarray.size(); ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

# CHARACTER SEQUENCES

▷ Because strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.

▷ By convention, the end of strings represented in character sequences is signalled by a special character: the null character, whose literal value can be written as '\0' (backslash, zero).

## ▷ Initialization of NULL Terminated Character Sequences

▽ Sequences of characters enclosed in double-quotes (") are literal constants. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

▽

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword[] = "Hello";
```

▽ Because string literals are regular arrays, they have the same restrictions as these, and cannot be assigned values.Expressions (once myword has already been declared as above), such as:

```
1 myword = "Bye";
2 myword[] = "Bye";
```

would **not** be valid, like neither would be:

```
myword = { 'B', 'y', 'e', '\0' };
```

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

```
1 myword[0] = 'B';
2 myword[1] = 'y';
3 myword[2] = 'e';
4 myword[3] = '\0';
```

## ▷ Strings and null-terminated character sequences

▽ Plain arrays with null-terminated sequences of characters are the typical types used in the C language to represent strings (that is why they are also known as C-strings). In C++, even though the standard library defines a specific type for strings (class string), still, plain arrays with null-terminated sequences of characters (C-strings) are a natural way of representing strings

in the language; in fact, string literals still always produce null-terminated character sequences, and not string objects.

▽ In the standard library, both representations for strings (C-strings and library strings) coexist, and most functions requiring strings are overloaded to support both.

▽

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  return 0;
}
```

```
What is your name? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

Edit & Run

▽ Arrays have a fixed size that needs to be specified either implicit or explicitly when declared; while strings are simply strings, no size is specified. This is due to the fact that strings have a dynamic size determined during runtime, while the size of arrays is determined on compilation, before the program runs.

▽ Null-terminated character sequences can be transformed into strings implicitly, and strings can be transformed into null-terminated character sequences by using either of string's member functions c_str or data (both are equivalent):

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to string
cout << mystring;          // printed as a library string
cout << mystring.c_str();  // printed as a c-string
```

# POINTERS

▷ The actual address of a variable in memory cannot be known before runtime.

▷ DECLARATION

▽ Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

▽ The size in memory of a pointer depends on the platform where the program runs.

▷ POINTER & ARRAY

▽ The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int myarray[20];
int * mypointer;
mypointer=myarray;//valid statement
```

▽ The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything. Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.

▽

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

▽ [] brackets are a dereferencing operator known as offset operator. They dereference the variable they follow just as *

does, but they also add the number between brackets to the address being dereferenced.

▽

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed to by (a+5) = 0
```

## ▷ Pointer Arithmetic

▽ Only addition and subtraction operations are allowed, They behave according to the size of the data type to which they point.

▽ Postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (*). Therefore, the following expression:
*p++ is equivlent to *(p++).

▽ Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
*p++    // same as *(p++): increment pointer, and dereference unincremented address
*++p    // same as *(++p): increment pointer, and dereference incremented address
++*p    // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

▽

```
*p++ = *q++;
```

Because ++ has a higher precedence than *, both p and q are incremented, but because both increment operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are incremented. And then both are incremented. It would be roughly equivalent to:

```
*p = *q;
++p;
++q;
```

## ▷ Pointer & Constant

▽ Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough

with qualifying the type pointed to by the pointer as const. For example:

▽

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

▽ Note also, that the expression &y is of type int*, but this is assigned to a pointer of type const int*. This is allowed: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around! As a safety feature, pointers to const are not implicitly convertible to pointers to non-const.

▽ One of the use cases of pointers to const elements is as function parameters: a function that takes a pointer to non-const as parameter can modify the value passed as argument, while a function that takes a pointer to const as parameter cannot.

▽ const data_type * pointers point to constant content they cannot modify, but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.

▽ Pointers can also be themselves const. And this is specified by appending const to the pointed type (after the asterisk):

```
int x;
      int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
      int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

▽ The const qualifier can either precede or follow the pointed type, with the exact same meaning:

```
const int * p2a = &x;  //       non-const pointer to const int
int const * p2b = &x;  // also non-const pointer to const int
```

## ▷ Pointers & String Literals

▽ String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type const char (as literals, they can never be modified). For example:

const char * foo = "hello";

## ▷ Pointers To Pointers

▽ The syntax simply requires an asterisk (*) for each level of indirection in the declaration of the pointer:

▽

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

## ▷ Void Pointers

▽ void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

▽ This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters.

▽ They have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

▽ One of its possible uses may be to pass generic parameters to a function. For example:

```
// increaser                                    y, 1603
#include <iostream>
using namespace std;

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

## ▷ Invalid & Null Pointers

▽ Pointers can actually point to any address, including addresses that do not refer to any valid element. In contrast to pointing to a variable or to an element of an array.

▽ Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

```
int * p;                  // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

▽ Neither p nor q point to addresses known to contain a value, but none of the above statements causes an error. In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not. What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to). Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

▽ But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address. For such cases, there exists a special value that any pointer type can take: the null pointer value. This value can be expressed in C++ in two ways: either with an integer value of zero, or with the nullptr keyword:

```
int * p = 0;
int * q = nullptr;
```

▽ All null pointers compare equal to all other null pointers.

▽ Do not confuse null pointers with void pointers! A null pointer is a value that any pointer can take to represent that it is pointing to "nowhere", while a void pointer is a type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer, and the other to the type of data it points to.

## ▷ Pointers To Functions

▽ C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```
// pointer to functions                                    8
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

▽ In the example above, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function subtraction:

*int* (* minus)(*int*,*int*) = subtraction;

# DYNAMIC MEMORY

## ▷ Operators new and new[]

▽ Returns the address of the beginning of the new block of memory allocated.

▽

```
pointer = new type
pointer = new type [number_of_elements]
```

▽ The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

▽ C++ provides two standard mechanisms to check if the allocation was successful:

- ▸ One is by handling exceptions. Using this method, an exception of type bad_alloc is thrown when the allocation fails.
  if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.This exception method is the method used by default by new.
- ▸ The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a *null pointer*, and the program continues its execution normally.
  This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:
  foo=new (nothrow) int[5];

▽ This nothrow method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations.

## ▷ Operations delete & delete[]

- ▽ The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

## ▷ Dynamic Memory In C

- ▽ It uses functions malloc, calloc, realloc and free, defined in the header <cstdlib>.
- ▽ Malloc : void* malloc (size_t size);//size_t is an unsigned integral type and size if the size of memory block
  - ▸ Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.
  - ▸ Always returns a pointer of type void* and null pointer if allocation fails.
  - ▸ The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.
  - ▸ If *size* is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.
- ▽ Calloc : void* calloc (size_t num, size_t size);//num is the no of elements to allocate, size if the size of each element, size_t is an unsigned integral type
  - ▸ Allocates a block of memory for an array of num elements, each of them size bytes long, and initializes all its bits to zero.
  - ▸ On success, a pointer to the memory block allocated by the function.
  - ▸ The type of this pointer is always void*, which can be cast to the desired type of data pointer in order to be dereferenceable.
  - ▸ If the function failed to allocate the requested block of memory, a null pointer is returned.
  - ▸ The effective result is the allocation of a zero-initialized memory block of (num*size) bytes.
  - ▸ If size is zero, the return value depends on the particular library implementation (it may or may not be a null pointer), but the returned pointer shall not be dereferenced.

▽ Realloc : void* realloc (void* ptr, size_t size);//ptr is pointer to a memory block previously allocated with malloc, calloc or realloc. Alternatively, this can be a null pointer, in which case a new block is allocated (as if malloc was called).

- ▸ Changes the size of the memory block pointed to by *ptr*.
- ▸ The function may move the memory block to a new location (whose address is returned by the function).
- ▸ The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new *size* is larger, the value of the newly allocated portion is indeterminate.
- ▸ Otherwise, if size is zero, the memory previously allocated at ptr is deallocated as if a call to free was made, and a null pointer is returned.
- ▸ If size is zero, the return value depends on the particular library implementation: it may either be a null pointer or some other location that shall not be dereferenced.
- ▸ If the function fails to allocate the requested block of memory, a null pointer is returned, and the memory block pointed to by argument ptr is not deallocated (it is still valid, and with its contents unchanged).

▽ Free : void free (void* ptr);//ptr is pointer to the block of memory previously allocated with malloc, calloc or realloc

- ▸ Deallocates a block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.
- ▸ If ptr does not point to a block of memory allocated with the above functions, it causes undefined behavior.
- ▸ If ptr is a null pointer, the function does nothing.
- ▸ Notice that this function does not change the value of ptr itself, hence it still points to the same (now invalid) location.
- ▸ It does not returns anything.

# DATA STRUCTURES

## ▷ Data Structures

▽ A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures can be declared in C++ using the following syntax:

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

▽ Where type_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

▽ Declaring Objects :

▶

```
1 struct product {
2   int weight;
3   double price;
4 } ;
5
6 product apple;
7 product banana, melon;
```

```
1 struct product {
2   int weight;
3   double price;
4 } apple, banana, melon;
```

▽ The members of an object can be accessed directly using a “.”.

▽ Because structures are types, they can also be used as the type of arrays to construct tables or databases of them.

## ▷ Pointers To Structures

▽ Structures can be pointed to by its own type of pointers.

▽ The arrow operator (→) is a dereference operator that is used exclusively with pointers to objects that have members. This operator serves to access the member of an object directly from its address.

## ▷ Nesting Structures

▽ Structures can also be nested in such a way that an element of a structure is itself another structure.

# OTHER DATA TYPES

## ▷ Type Aliases (typedef / using)

▽ A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

▽ Two ways :

- ▸ typedef(Inherited from c language)

  typedef existing_type new_type_name ;

  where existing type is any type, Either fundamental or compound.

- ▸ using(defined in c++)

  using new_type_name = existing_type ;

▽ Both aliases defined with typedef and aliases defined with using are semantically equivalent. The only difference being that typedef has certain limitations in the realm of templates that using has not. Therefore, using is more generic, although typedef has a longer history and is probably more common in existing code.

## ▷ Unions

▽ Unions allow one portion of memory to be accessed as different data types. Its declaration and use is similar to the one of structures, but its functionality is totally different:

```
union type_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

▽ This creates a new union type, identified by type_name, in which all its member elements occupy the same physical space in memory. The size of this type is the one of the largest member element. For example:

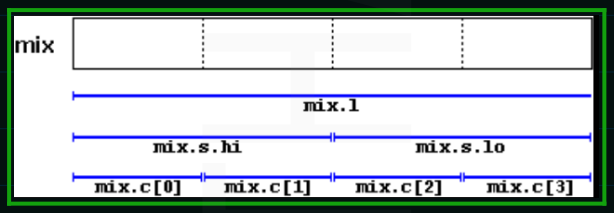
```
1 union mytypes_t {
2   char c;
3   int i;
4   float f;
5 } mytypes;
```

Each of the members(c,i,f) is of a different data type. But since all of them are referring to the same location in memory, the

modification of one of the members will affect the value of all of them. It is not possible to store different values in them in a way that each is independent of the others.

▽ One of the uses of a union is to be able to access a value either in its entirety or as an array or structure of smaller elements. For example:

```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```




▽ The exact alignment and order of the members of a union in memory depends on the system, with the possibility of creating portability issues.

## ▷ Anonymous Unions

▽ When unions are members of a class (or structure), they can be declared with no name. In this case, they become anonymous unions, and its members are directly accessible from objects by their member names. For example, see the differences between these two structure declarations:

| structure with regular union | structure with anonymous union |
|---|---|
| `struct book1_t {`<br>`  char title[50];`<br>`  char author[50];`<br>`  union {`<br>`  float dollars;`<br>`  int yen;`<br>`  } price;`<br>`} book1;` | `struct book2_t {`<br>`  char title[50];`<br>`  char author[50];`<br>`  union {`<br>`  float dollars;`<br>`  int yen;`<br>`  };`<br>`} book2;` |

▽ Accessing :

```
book1.price.dollars
book1.price.yen
```

```
book2.dollars
book2.yen
```

## ▷ Enumerated Types (enum)

▽ Enumerated types are types that are defined with a set of custom identifiers, known as enumerators, as possible values. Objects of these enumerated types can take any of these enumerators as value.

▽ Syntax :

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

▽ This creates the type type_name, which can take any of value1, value2, value3, ... as value. Objects (variables) of this type can directly be instantiated as object_names.

▽ Eg.

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

This declaration includes no other type, neither fundamental nor compound, in its definition. To say it another way, somehow, this creates a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type color_t may take are the enumerators listed within braces.

```
1 colors_t mycolor;
2
3 mycolor = blue;
4 if (mycolor == green) mycolor = red;
```

▽ Values of enumerated types declared with enum are implicitly convertible to an integer type. In fact, the elements of such an enum are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to. If it is not specified otherwise, the integer value equivalent to the first possible value is 0, the equivalent to the second is 1, to the third is 2, and so on... Therefore, in the data type colors_t defined above, black would be equivalent to 0, blue would be equivalent to 1, green to 2, and so on...

▽ A specific integer value can be specified for any of the possible values in the enumerated type. And if the constant value that follows it is itself not given its own value, it is automatically assumed to be the same value plus one. For example:

▽ Here january is represented by 1 and december by 12.

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

## ▷ Enumerated Types With enum Classes

▽ In C++, it is possible to create real enum types that are neither implicitly convertible to int and that neither have enumerator values of type int, but of the enum type itself, thus preserving type safety. They are declared with enum class (or enum struct) instead of just enum:

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Each of the enumerator values of an enum class type needs to be scoped into its type (this is actually also possible with enum types, but it is only optional). For example:

```
Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```

▽ Enumerated types declared with enum class also have more control over their underlying type; it may be any integral data type, such as char, short or unsigned int, which essentially serves to determine the size of the type. This is specified by a colon and the underlying type following the enumerated type. For example:

```
enum class EyeColor : char {blue, green, brown};
```

Here, Eyecolor is a distinct type with the same size of a char (1 byte).

# CLASSES - I

▷ Like data structures, they can contain data members, but they can also contain functions as members.

▷ Classes are defined using either keyword class or keyword struct, with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

▷ The body of the declaration can contain members, which can either be data or function declarations, and optionally access specifiers.

▷ An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights for the members that follow them:

▽ Private : members of a class are accessible only from within other members of the same class (or from their "friends").

▽ Protected : members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.

▽ Public : members are accessible from anywhere where the object is visible.

▷ By default, all members of a class declared with the class keyword have private access for all its members.

▷ Eg.

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
area: 12
```

Edit & Run

▷ Here the scope resolution operator is used in the definition of function to define a member of a class outside the class itself.

▷ The only difference between defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class, is that in the first case the function is automatically considered an inline member function by the compiler, while in the second it is a normal (not-inline) class member function. This causes no differences in behavior, but only on possible compiler optimizations.

▷ Classes allow programming using object-oriented paradigms: Data and functions are both members of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called.

▷ CONSTRUCTORS

▽ A class can include a special function called its constructor, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

▽ This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

Edit & Run

▽ Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.

▽ Constructors never return values, they simply initialize the object.

## ▷ Overloading Constructors

▽ A constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types.

▽

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

Edit & Run

▽ The default constructor : The default constructor is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. Empty parentheses cannot be used to call the default constructor. This is because the empty set of parentheses would make of rectc a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type Rectangle.

▽

```
Rectangle rectb;   // ok, default constructor called
Rectangle rectc(); // oops, default constructor NOT called
```

## ▷ Uniform Initialization

▽ The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as functional form. But constructors can also be called with other syntaxes:

- ▸ First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument): class_name object_name = initialization_value;
- ▸ More recently, C++ introduced the possibility of constructors to be called using uniform initialization, which essentially is the

same as the functional form, but using braces ({}) instead of parentheses (()): class_name object_name { value, value, ... } or class_name object_name = {value, value, ...}

▽ Different ways of initialization :

```cpp
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum() << '\n';
  return 0;
}
```

```
foo's circumference: 62.8319
```

Edit & Run

▽ An advantage of uniform initialization over functional form is that, unlike parentheses, braces cannot be confused with function declarations, and thus can be used to explicitly call default constructors.

```cpp
Rectangle rectb;   // default constructor called
Rectangle rectc(); // function declaration (default constructor NOT called)
Rectangle rectd{}; // default constructor called
```

▽ Uniform Initialization has preference for initializer_list as its type.

Initialzer List

This type is used to access the values in a C++ initialization list, which is a list of elements of type `const T`.

Objects of this type are automatically constructed by the compiler from initialization list declarations, which is a list of comma-separated elements enclosed in braces:

```cpp
auto il = { 10, 20, 30 };  // the type of il is an initializer_list
```

Notice though that this template class is not implicitly defined and the header `<initializer_list>` shall be included to access it, even if the type is used implicitly.

`initializer_list` objects are automatically constructed as if an array of elements of type `T` was allocated, with each of the elements in the list being copy-initialized to its corresponding element in the array, using any necessary non-narrowing implicit conversions.

The `initializer_list` object refers to the elements of this array without containing them: copying an `initializer_list` object produces another object referring to the same underlying elements, not to new copies of them (reference semantics).

The lifetime of this temporary array is the same as the `initializer_list` object.

Constructors taking only one argument of this type are a special kind of constructor, called *initializer-list constructor*. Initializer-list constructors take precedence over other constructors when the initializer-list constructor syntax is used:

```cpp
struct myclass {
  myclass (int,int);
  myclass (initializer_list<int>);
  /* definitions ... */
};

myclass foo {10,20};  // calls initializer_list ctor
myclass bar (10,20);  // calls first constructor
```

## ▷ Member Initialization In Constructors

▽ When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members.

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

▽ For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.

▽ Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list. For example:

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

```
foo's volume: 6283.19
```

▽ In this example, class Cylinder has a member object whose type is another class (base's type is Circle). Because objects of class Circle can only be constructed with a parameter, Cylinder's constructor needs to call base's constructor, and the only way to do this is in the member initializer list.

▽ These initializations can also use uniform initializer syntax, using braces {} instead of parentheses ():

```
Cylinder::Cylinder (double r, double h) : base{r}, height{h} { }
```

## ▷ Pointers To Classes

▽ Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer. For example:

Rectangle * prect;

is a pointer to an object of class Rectangle.

## ▷ Interpretations

▽

| expression | can be read as |
|---|---|
| *x | pointed to by x |
| &x | address of x |
| x.y | member y of object x |
| x->y | member y of object pointed to by x |
| (*x).y | member y of object pointed to by x (equivalent to the previous one) |
| x[0] | first object pointed to by x |
| x[1] | second object pointed to by x |
| x[n] | (n+1)th object pointed to by x |

## ▷ Classed Defined Wtih Struct & Union Keywords

▽ Classes can be defined not only with keyword class, but also with keywords struct and union.

▽ The keyword struct, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword class. The only difference between both is that members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access by default. For all other purposes both keywords are equivalent in this context.

▽ Conversely, the concept of unions is different from that of classes declared with struct and class, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is public.

# CLASSES - II

## ▷ Overloading Operators

▽ C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes. Here is a list of all the operators that can be overloaded:

| Overloadable operators | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + | - | * | / | = | < | > | += | -= | *= | /= | << | >> |
| <<= | >>= | == | != | <= | >= | ++ | -- | % | & | ^ | ! | \| |
| ~ | &= | ^= | \|= | && | \|\| | %= | [] | () | , | ->* | -> | new |
| delete | | new[] | | delete[] | | | | | | | | |

▽ Operators are overloaded by means of operator functions, which are regular functions with special names: their name begins by the operator keyword followed by the operator sign that is overloaded. The syntax is:

type operator sign (parameters) { /*... body ...*/ }

▽ Eg. Overloading + operator for cartesian vectors

```
1 // overloading operators example                    4,3
2 #include <iostream>
3 using namespace std;
4
5 class CVector {
6   public:
7     int x,y;
8     CVector () {};
9     CVector (int a,int b) : x(a), y(b) {}
10    CVector operator + (const CVector&);
11 };
12
13 CVector CVector::operator+ (const CVector& param) {
14   CVector temp;
15   temp.x = x + param.x;
16   temp.y = y + param.y;
17   return temp;
18 }
19
20 int main () {
21   CVector foo (3,1);
22   CVector bar (1,2);
23   CVector result;
24   result = foo + bar;
25   cout << result.x << ',' << result.y << '\n';
26   return 0;
27 }
```

▽ The function operator+ of class CVector overloads the addition operator (+) for that type. Once declared, this function can be called either implicitly using the operator, or explicitly using its functional name:

c = a + b;

c = a.*operator*+ (b);

▽ The parameter expected for a member function overload for operations such as operator+ is naturally the operand to the right hand side of the operator. This is common to all binary operators (those with an operand to its left and one operand to its right).

▽ Summary of the parameters needed for each of the different operators than can be overloaded (please, replace @ by the operator in each case):

| Expression | Operator | Member function | Non-member function |
|---|---|---|---|
| @a | + - * & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A,int) |
| a@b | + - * / % ^ & \| < > == != <= >= << >> && \|\| , | A::operator@(B) | operator@(A,B) |
| a@b | = += -= *= /= %= ^= &= \|= <<= >>= [] | A::operator@(B) | - |
| a(b,c...) | () | A::operator()(B,C...) | - |
| a->b | -> | A::operator->() | - |
| (TYPE) a | TYPE | A::operator TYPE() | - |

Where a is an object of class A, b is an object of class B and c is an object of class C. TYPE is just any type (that operators overloads the conversion to type TYPE).

▽ Some operators may be overloaded in two forms: either as a member function or as a non-member function.

▽ When any operator is overloaded as non-member functions; In this case, the operator function takes an object of the proper class as first argument.

```
// non-member operator overloads                                    4,3
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {}
    CVector (int a, int b) : x(a), y(b) {}
};


CVector operator+ (const CVector& lhs, const CVector& rhs) {
  CVector temp;
  temp.x = lhs.x + rhs.x;
  temp.y = lhs.y + rhs.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

## ▷ The Keyword this

▽ The keyword this represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer to the object itself.

▽ One of its uses can be to check if a parameter passed to a member function is the object itself. For example:

```
// example on this                                    yes, &a is b
#include <iostream>
using namespace std;

class Dummy {
  public:
    bool isitme (Dummy& param);
};

bool Dummy::isitme (Dummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  Dummy a;
  Dummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b\n";
  return 0;
}
```

▽ It is also frequently used in operator overloaded member functions that return objects by reference. Following with the examples on cartesian vector seen before, its operator= function could have been defined as:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

## ▷ Static Members

▽ A class can contain static members, either data or functions.

▽ A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value is not different from one object of this class to another.

▽ It may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```cpp
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
  public:
    static int n;
    Dummy () { n++; };
};

int Dummy::n=0;

int main () {
  Dummy a;
  Dummy b[5];
  cout << a.n << '\n';
  Dummy * c = new Dummy;
  cout << Dummy::n << '\n';
  delete c;
  return 0;
}
```

▽ In fact, static members have the same properties as non-member variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it. As in the previous example:

*int* Dummy::n=0;

▽ Because it is a common variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

cout << a.n;

cout << Dummy::n;

▽ Classes can also have static member functions. These represent the same: members of a class that are common to all object of that class, acting exactly as non-member functions but being accessed like members of the class. Because they are like non-member functions, they cannot access non-static members of the class (neither member variables nor member functions). They neither can use the keyword this.

## ▷ Const Member Functions

▽ When an object of a class is qualified as a const object:

*const* MyClass myobject;

The access to its data members from outside the class is restricted to read-only, as if all its data members were const for those accessing them from outside the class. Note though, that the

constructor is still called and is allowed to initialize and modify these data members:

```
// constructor on const object                                    10
#include <iostream>
using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {
  const MyClass foo(10);
// foo.x = 20;           // not valid: x cannot be modified
  cout << foo.x << '\n';  // ok: data member x can be read
  return 0;
}
```

▽ The member functions of a const object can only be called if they are themselves specified as const members. To specify that a member is a const member, the const keyword shall follow the function prototype, after the closing parenthesis for its parameters:

*int* get() const {*return* x;}

▽ Note that const can be used to qualify the type returned by a member function. This const is not the same as the one which specifies a member as const. Both are independent and are located at different places in the function prototype:

```
int get() const {return x;}       // const member function
const int& get() {return x;}      // member function returning a const&
const int& get() const {return x;} // const member function returning a const&
```

▽ Member functions specified to be const cannot modify non-static data members nor call other non-const member functions. In essence, const members shall not modify the state of an object.

▽ const objects are limited to access only member functions marked as const, but non-const objects are not restricted and thus can access both const and non-const member functions alike.

▽ Most functions taking classes as parameters actually take them by const reference, and thus, these functions can only access their const members:

```cpp
// const objects
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
};

void print (const MyClass& arg) {
  cout << arg.get() << '\n';
}

int main() {
  MyClass foo (10);
  print(foo);

  return 0;
}
```

▽ Member functions can be overloaded on their constness: i.e., a class may have two member functions with identical signatures except that one is const and the other is not: in this case, the const version is called only when the object is itself const, and the non-const version is called when the object is itself non-const.

▽

```cpp
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
    int& get() {return x;}
};

int main() {
  MyClass foo (10);
  const MyClass bar (20);
  foo.get() = 15;         // ok: get() returns int&
// bar.get() = 25;        // not valid: get() returns const int&
  cout << foo.get() << '\n';
  cout << bar.get() << '\n';

  return 0;
}
```

## ▷ Class Templates

▽ Defination :

```cpp
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

▽ Call :

```cpp
mypair<int> myobject (115, 36);
```

▽ In case that a member function is defined outside the defintion of the class template, it shall be preceded with the template < ... > prefix:

```
// class templates                                    100
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

▽

```
template <class T>
T mypair<T>::getmax ()
```

The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

## ▷ Template Specialization

▽ It is possible to define a different implementation for a template when a specific type is passed as template argument.

▽

```
// template specialization                            8
#include <iostream>                                   J
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

▽ Syntax :

```
template <> class mycontainer <char> { ... };
```

▽ The empty <> signifies that all types are known and no template arguments are required for this specialization.
The <char> specialization parameter itself identifies the type for which the template class is being specialized (char).

▽

```
1 template <class T> class mycontainer { ... };
2 template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

▽ When we declare specializations for a template class, we must also define all its members, even those identical to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

# SPECIAL MEMBERS

▷ Special member functions are member functions that are implicitly defined as member of classes under certain circumstances. There are six:

| Member function | typical form for class C: |
|---|---|
| Default constructor | `C::C();` |
| Destructor | `C::~C();` |
| Copy constructor | `C::C (const C&);` |
| Copy assignment | `C& operator= (const C&);` |
| Move constructor | `C::C (C&&);` |
| Move assignment | `C& operator= (C&&);` |

▷ DEFAULT CONSTRUCTOR

▽ The default constructor is the constructor called when objects of a class are declared, but are not initialized with any arguments.

▽ If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor. Therefore, after declaring a class like this:

```
class Example {
  public:
    int total;
    void accumulate (int x) { total += x; }
};
```

The compiler assumes that this class has a default constructor. Declaring objects of this class :

Example ex;

▽ But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments.

```
class Example2 {
  public:
    int total;
    Example2 (int initial_value) : total(initial_value) { };
    void accumulate (int x) { total += x; };
};
```

Decalring objects of this class :

Example ex{100};//valid

Example ex;//Invalid

Therefore, if objects of this class need to be constructed without arguments, the proper default constructor also needs to be declared in the class.

▽

```
// classes and default constructors                    bar's content: Example
#include <iostream>
#include <string>
using namespace std;

class Example3 {
    string data;
  public:
    Example3 (const string& str) : data(str) {}
    Example3() {}
    const string& content() const {return data;}
};

int main () {
  Example3 foo;
  Example3 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

## ▷ Destructor

▽ They are responsible for the necessary cleanup needed by a class when its lifetime ends (If there is dynamic allocation).

▽ it would be very useful to have a function called automatically at the end of the object's life in charge of releasing the allocated dynamic memory. To do this, we use a destructor. A destructor is a member function very similar to a default constructor: it takes no arguments and returns nothing, not even void. It also uses the class name as its own name, but preceded with a tilde sign (~).

```
// destructors                                          bar's content: Example
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

## ▷ Copy Constructor

▽ When an object is passed a named object of its own type as argument, its copy constructor is invoked in order to construct a copy.

▽ A copy constructor is a constructor whose first parameter is of type reference to the class itself (possibly const qualified) and which can be invoked with a single argument of this type. For

example, for a class MyClass, the copy constructor may have the following signature:

MyClass::MyClass (*const* MyClass&);

▽ If a class has no custom copy nor move constructors (or assignments) defined, an implicit copy constructor is provided. This copy constructor simply performs a copy of its own members. For example, for a class such as:

```
class MyClass {
  public:
    int a, b; string c;
};
```

An implicit copy constructor is automatically defined. The definition assumed for this function performs a shallow copy, roughly equivalent to:

MyClass::MyClass(*const* MyClass& x) : a(x.a), b(x.b), c(x.c) {}

▽ shallow copies only copy the members of the class themselves.

▽ For classes that contains pointers of which it handles its storage. For that class, performing a shallow copy means that the pointer value is copied, but not the content itself; This means that both objects (the copy and the original) would be sharing a single string object (they would both be pointing to the same object), and at some point (on destruction) both objects would try to delete the same block of memory, probably causing the program to crash on runtime. This can be solved by defining the following custom *copy constructor* that performs a deep copy:

```
// copy constructor: deep copy                          bar's content: Example
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

## ▷ Copy Assignment

▽ Objects are not only copied on construction, when they are initialized: They can also be copied on any assignment operation. See the difference:

```
MyClass foo;
MyClass bar (foo);       // object initialization: copy constructor called
MyClass baz = foo;       // object initialization: copy constructor called
foo = bar;               // object already initialized: copy assignment called
```

▽ The copy assignment operator is an overload of operator= which takes a value or reference of the class itself as parameter. The return value is generally a reference to *this (although this is not required). For example, for a class MyClass, the copy assignment may have the following signature:

MyClass& *operator*= (*const* MyClass&);

▽ The copy assignment operator is also a special function and is also defined implicitly if a class has no custom copy nor move assignments (nor move constructor) defined.

▽ The implicit version performs a shallow copy which is not suitable for classes with pointers to objects they handle its storage.

▽ Not only the class incurs the risk of deleting the pointed object twice, but the assignment creates memory leaks by not deleting the object pointed by the object before the assignment. These issues could be solved with a copy assignment that deletes the previous object and performs a deep copy:

```
Example5& operator= (const Example5& x) {
  delete ptr;                      // delete currently pointed string
  ptr = new string (x.content());  // allocate space for new string, and copy
  return *this;
}

```

## ▷ Move Constructor & Assingment

▽ Unlike copying, the content is actually transferred from one object (the source) to the other (the destination): the source loses that content, which is taken over by the destination. This moving only happens when the source of the value is an unnamed object.

▽ Unnamed objects are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of unnamed objects are return values of functions or type-casts.

▽ The move constructor is called when an object is initialized on construction using an unnamed temporary. Likewise, the move assignment is called when an object is assigned the value of an unnamed temporary:

```
MyClass fn();             // function returning a MyClass object
MyClass foo;              // default constructor
MyClass bar = foo;        // copy constructor
MyClass baz = fn();       // move constructor
foo = bar;                // copy assignment
baz = MyClass();          // move assignment
```

▽ Both the value returned by fn and the value constructed with MyClass are unnamed temporaries. In these cases, there is no need to make a copy, because the unnamed object is very short-lived and can be acquired by the other object when this is a more efficient operation.

▽ The move constructor and move assignment are members that take a parameter of type rvalue reference to the class itself:

```
MyClass (MyClass&&);               // move-constructor
MyClass& operator= (MyClass&&);    // move-assignment
```

▽ An rvalue reference is specified by following the type with two ampersands (&&). As a parameter, an rvalue reference matches arguments of temporaries of this type.

▽ The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete. In such objects, copying and moving are really different operations:

- Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B.
- Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage.

▽

```
// move constructor/assignment                          foo's content: Example
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
  public:
    Example6 (const string& str) : ptr(new string(str)) {}
    ~Example6 () {delete ptr;}
    // move constructor
    Example6 (Example6&& x) : ptr(x.ptr) {x.ptr=nullptr;}
    // move assignment
    Example6& operator= (Example6&& x) {
      delete ptr;
      ptr = x.ptr;
      x.ptr=nullptr;
      return *this;
    }
    // access content:
    const string& content() const {return *ptr;}
    // addition:
    Example6 operator+(const Example6& rhs) {
      return Example6(content()+rhs.content());
    }
};


int main () {
  Example6 foo ("Exam");
  Example6 bar = Example6("ple");   // move-construction

  foo = foo + bar;                  // move-assignment

  cout << "foo's content: " << foo.content() << '\n';
  return 0;
}
```

▽ Compilers already optimize many cases that formally require a move-construction call in what is known as Return Value Optimization. Most notably, when the value returned by a function is used to initialize an object. In these cases, the move constructor may actually never get called.

## ▷ Implicit Members

▽ The six special members functions described above are members implicitly declared on classes under certain circumstances:

| Member function | implicitly defined: | default definition: |
|---|---|---|
| Default constructor | if no other constructors | does nothing |
| Destructor | if no destructor | does nothing |
| Copy constructor | if no move constructor and no move assignment | copies all members |
| Copy assignment | if no move constructor and no move assignment | copies all members |
| Move constructor | if no destructor, no copy constructor and no copy nor move assignment | moves all members |
| Move assignment | if no destructor, no copy constructor and no copy nor move assignment | moves all members |

▽ Each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords default and delete, respectively. The syntax is either one of:

```
function_declaration = default;
function_declaration = delete;
```

▽

```
// default and delete implicit members                bar's area: 200
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int x, int y) : width(x), height(y) {}
    Rectangle() = default;
    Rectangle (const Rectangle& other) = delete;
    int area() {return width*height;}
};

int main () {
  Rectangle foo;
  Rectangle bar (10,20);

  cout << "bar's area: " << bar.area() << '\n';
  return 0;
}
```

Here, Rectangle can be constructed either with two int arguments or be default-constructed (with no arguments). It cannot however be copy-constructed from another Rectangle object, because this function has been deleted. Therefore, assuming the objects of the last example, the following statement would not be valid:

```
Rectangle baz (foo);
```

It could, however, be made explicitly valid by defining its copy constructor as:

```
Rectangle::Rectangle (const Rectangle& other) = default;
```

▽ The keyword default does not define a member function equal to the default constructor (i.e., where default constructor means constructor with no parameters), but equal to the constructor that would be implicitly defined if not deleted.

▽ In general, and for future compatibility, classes that explicitly define one copy/move constructor or one copy/move assignment but not both, are encouraged to specify either delete or default on the other special member functions they don't explicitly define.

# FRIENDSHIP AND INHERITANCE

## ▷ Friend Function

▽ In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".

▽ Friends are functions or classes declared with the friend keyword.

▽

```
// friend functions                                    24
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

▽ Function duplicate is not considered a member of class Rectangle. It simply has access to its private and protected members without being a member.

▽ Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

## ▷ Friend Classes

▽

```cpp
// friend class                                    16
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

▽ Empty declaration of class Square is necessary because class Rectangle uses Square (as a parameter in member convert), and Square uses Rectangle (declaring it a friend).

▽ Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around.

▽ Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.

## ▷ Inheritance Between Classes

▽ Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a base class and a derived class: The derived class inherits the members of the base class, on top of which it can add its own members.

▽ The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

class derived_class_name: public base_class_name
{ /* ... */ };

▽ The public access specifier may be replaced by any one of the other access specifiers (protected or private). This access specifier limits the most accessible level for the members inherited from the base class: The members with a less restrictive level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

▽

```
// derived classes                                    20
#include <iostream>                                   10
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

▽ Access :

| Access | public | protected | private |
|---|---|---|---|
| members of the same class | yes | yes | yes |
| members of derived class | yes | yes | no |
| not members | yes | no | no |

▽ If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword class and public for those declared with struct.

## ▷ What Is Inherited From The Base Class?

▽ In principle, a publicly derived class inherits access to every member of a base class except:
its constructors and its destructor
its assignment operator members (operator=)
friends
its private members

▽ Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class

▽ Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments). Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

▽

```
// constructors and derived classes
#include <iostream>
using namespace std;

class Mother {
  public:
    Mother ()
      { cout << "Mother: no parameters\n"; }
    Mother (int a)
      { cout << "Mother: int parameter\n"; }
};

class Daughter : public Mother {
  public:
    Daughter (int a)
      { cout << "Daughter: int parameter\n\n"; }
};

class Son : public Mother {
  public:
    Son (int a) : Mother (a)
      { cout << "Son: int parameter\n\n"; }
};

int main () {
  Daughter kelly(0);
  Son bud(0);

  return 0;
}
```

```
Mother: no parameters
Daughter: int parameter

Mother: int parameter
Son: int parameter
```

## ▷ Multiple Inheritence

▽ A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class's base classes (i.e., after the colon).

▽

```
// multiple inheritance                                    20
#include <iostream>                                        10
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
};

class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```

# POLYMORPHISM

## ▷ Pointer To Base Class

▽ One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.

▽

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

▽ But because the type of both ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

## ▷ Virtual Members

▽ A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the virtual keyword:

```
// virtual members                          20
#include <iostream>                         10
using namespace std;                        0

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

▽ Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

▽ A class that declares or inherits a virtual function is called a polymorphic class.

## ▷ Abstract Base Classes

▽ Abstract base classes are something very similar to the Polygon class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). The syntax is to replace their definition by =0 (an equal sign and a zero):

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

- Classes that contain at least one pure virtual function are known as abstract base classes.
- Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of Polygon could not be used to declare objects like:

```
Polygon mypolygon;// not working if Polygon is abstract base class
```

- But an abstract base class is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

```
Polygon * ppoly1;
Polygon * ppoly2;
```

-

```
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

```
20
10
```

- In this example, objects of different but related types are referred to using a unique type of pointer (Polygon*) and the proper member function is called every time, just because they are virtual. This can be really useful in some circumstances.
- It is even possible for a member of the abstract base class Polygon to use the special pointer this to access the proper virtual members, even though Polygon itself has no implementation for this function:

▽

```
// pure virtual members can be called                    20
// from the abstract base class                          10
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

▽ Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for object-oriented projects.

▽

```
// dynamic allocation and polymorphism                   20
#include <iostream>                                      10
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

# TYPE CONVERSIONS

## ▷ Implicit Conversion

▽ Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of a is promoted from short to int without the need of any explicit operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow the conversions between numerical types (short to int, int to float, double to int ... ), to or from bool, and some pointer conversions.

▽ Converting to int from some smaller integer type, or to double from float is known as promotion, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ... ).

The conversions from/to bool consider false equivalent to zero (for numeric types) and to null pointer (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.

If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes undefined behavior.

Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is implementation-specific (and may not be portable).

▽ Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

▽ For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

Null pointers can be converted to pointers of any type

Pointers to any type can be converted to void pointers.

Pointer upcast: pointers to a derived class can be converted to a pointer of an accessible and unambiguous base class, without modifying its const or volatile qualification.

## ▷ Implicit Conversion With Classes

▽ Implicit conversions can be controlled by means of three member functions:

Single-argument constructors: allow implicit conversion from a particular type to initialize an object.

Assignment operator: allow implicit conversion from a particular type on assignments.

Type-cast operator: allow implicit conversion to a particular type.

▽

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;     // calls constructor
  bar = foo;       // calls assignment
  foo = bar;       // calls type-cast operator
  return 0;
}
```

▽ The type-cast operator uses a particular syntax: it uses the operator keyword followed by the destination type and an empty set of parentheses. Notice that the return type is the destination type and thus is not specified before the operator keyword.

## ▷ Keyword Explicit

▽ On a function call, C++ allows one implicit conversion to happen for each argument. This may be somewhat problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

*void* fn (B arg) {}

This function takes an argument of type B, but it could as well be called with an object of type A as argument:

fn (foo);

▽ It can be prevented by marking the affected constructor with the explicit keyword:

```
// explicit:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  explicit B (const A& x) {}
  B& operator= (const A& x) {return *this;}
  operator A() {return A();}
};

void fn (B x) {}

int main ()
{
  A foo;
  B bar (foo);
  bar = foo;
  foo = bar;

//  fn (foo);  // not allowed for explicit ctor.
  fn (bar);

  return 0;
}
```

▽ Additionally, constructors marked with explicit cannot be called with the assignment-like syntax; In the above example, bar could not have been constructed with: B bar = foo;

▽ Type-cast member functions (those described in the previous section) can also be specified as explicit. This prevents implicit conversions in the same way as explicit-specified constructors do for the destination type.

## ▷ Type-Casting

▽ C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as type-casting. There exist two main syntaxes for generic type-casting: functional and c-like:

```
double x = 10.3;
int y;
y = int (x);    // functional notation
y = (int) x;    // c-like cast notation
```

▽ These operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that -while being syntactically correct- can cause runtime errors.

```
// class type-casting
#include <iostream>
using namespace std;

class Dummy {
    double i,j;
};

class Addition {
    int x,y;
  public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};

int main () {
  Dummy d;
  Addition * padd;
  padd = (Addition*) &d;
  cout << padd->result();
  return 0;
}
```

This code compiles without errors. Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.

▽ In order to control these types of conversions between classes, we have four specific casting operators: dynamic_cast, reinterpret_cast, static_cast and const_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const cast <new type> (expression)
```

## ▷ Dynamic Cast

▽ dynamic_cast can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

▽ This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion.

▽ But dynamic_cast can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual

members) if -and only if- the pointed object is a valid complete object of the target type. For example:

```
// dynamic_cast                                    Null pointer on second type-cast.
#include <iostream>
#include <exception>
using namespace std;

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
  try {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast.\n";

  } catch (exception& e) {cout << "Exception: " << e.what();}
  return 0;
}
```

Compatibility note: This type of dynamic_cast requires Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This needs to be enabled for runtime type checking using dynamic_cast to work properly with these types.

▽ The code above tries to perform two dynamic casts from pointer objects of type Base* (pba and pbb) to a pointer object of type Derived*, but only the first one is successful. Notice their respective initializations:

```
Base * pba = new Derived;
Base * pbb = new Base;
```

▽ Even though both are pointers of type Base*, pba actually points to an object of type Derived, while pbb points to an object of type Base. Therefore, when their respective type-casts are performed using dynamic_cast, pba is pointing to a full object of class Derived, whereas pbb is pointing to an object of class Base, which is an incomplete object of class Derived.

▽ When dynamic_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is thrown instead.

- dynamic_cast can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a void* pointer.

## STATIIC_CAST

- static_cast can perform conversions between pointers to related classes, not only upcasts (from pointer-to-derived to pointer-to-base), but also downcasts (from pointer-to-base to pointer-to-derived)
- No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of dynamic_cast.
-

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

  This would be valid code, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.
- Therefore, static_cast is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.
- static_cast is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can :
  - Convert from void* to any pointer type. In this case, it guarantees that if the void* value was obtained by converting from that same pointer type, the resulting pointer value is the same.
  - Convert integers, floating-point values and enum types to enum types.
- Additionally, static_cast can also perform the following:
  - Explicitly call a single-argument constructor or a conversion operator.

- Convert to rvalue references.
- Convert enum class values into integers or floating-point values.
- Convert any type to void, evaluating and discarding the value.

### Reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as intptr_t), is guaranteed to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable. For example:

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This code compiles, although it does not make much sense, since now b points to an object of a totally unrelated and likely incompatible class. Dereferencing b is unsafe.

### Const_cast

This type of casting manipulates the constness of the object pointed by a pointer, either to be set or to be removed. For example, in order to pass a const pointer to a function that expects a non-const argument:

```
// const_cast                                    sample text
#include <iostream>
using namespace std;

void print (char * str)
{
  cout << str << '\n';
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

▽ The example above is guaranteed to work because function print does not write to the pointed object. Note though, that removing the constness of a pointed object to actually write to it causes undefined behavior.

## ▷ TYPEID

▽ typeid allows to check the type of an expression:

typeid (expression)

▽ This operator returns a reference to a constant object of type type_info that is defined in the standard header <typeinfo>. A value returned by typeid can be compared with another value returned by typeid using operators == and ≠ or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

▽

```cpp
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

```
a and b are of different types:
a is: int *
b is: int
```

▽ When typeid is applied to classes, typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object :

```cpp
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }
  return 0;
}
```

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

▽ The string returned by member name of type_info depends on the specific implementation of your compiler and library.

▽ If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.

▷ <CSTDINT> / (STDINT.H)

▽ This header defines a set of integral type aliases with specific width requirements, along with macros specifying their limits and macro functions to create values of these types.

▽ Types :

▸ The following are typedefs of fundamental integral types or extended integral types.

| signed type | unsigned type | description |
|---|---|---|
| intmax_t | uintmax_t | Integer type with the maximum width supported. |
| int8_t | uint8_t | Integer type with a width of exactly 8, 16, 32, or 64 bits.<br>For signed types, negative values are represented using 2's complement.<br>No padding bits.<br>**Optional:** These typedefs are not defined if no types with such characteristics exist.* |
| int16_t | uint16_t | |
| int32_t | uint32_t | |
| int64_t | uint64_t | |
| int_least8_t | uint_least8_t | Integer type with a minimum of 8, 16, 32, or 64 bits.<br>No other integer type exists with lesser size and at least the specified width. |
| int_least16_t | uint_least16_t | |
| int_least32_t | uint_least32_t | |
| int_least64_t | uint_least64_t | |
| int_fast8_t | uint_fast8_t | Integer type with a minimum of 8, 16, 32, or 64 bits.<br>At least as fast as any other integer type with at least the specified width. |
| int_fast16_t | uint_fast16_t | |
| int_fast32_t | uint_fast32_t | |
| int_fast64_t | uint_fast64_t | |
| intptr_t | uintptr_t | Integer type capable of holding a value converted from a void pointer and then be converted back to that type with a value that compares equal to the original pointer.<br>**Optional:** These typedefs may not be defined in some library implementations.* |

▸ Some of these typedefs may denote the same types. Therefore, function overloads should not rely on these being different.

▸ Some types are optional (and thus, with no portability guarantees). A particular library implementation may also define additional types with other widths supported by its system. In any case, if either the signed or the unsigned version is defined, both the signed and unsigned versions are defined.

▽ Macros :

▸ Limits of cstdint types

| Macro | description | defined as |
|---|---|---|
| INTMAX_MIN | Minimum value of intmax_t | $-(2^{63}-1)$, or lower |
| INTMAX_MAX | Maximum value of intmax_t | $2^{63}-1$, or higher |
| UINTMAX_MAX | Maximum value of uintmax_t | $2^{64}-1$, or higher |
| INT*N*_MIN | Minimum value of exact-width signed type | Exactly $-2^{(N-1)}$ |
| INT*N*_MAX | Maximum value of exact-width signed type | Exactly $2^{(N-1)}-1$ |
| UINT*N*_MAX | Maximum value of exact-width unsigned type | Exactly $2^{N}-1$ |
| INT_LEAST*N*_MIN | Minimum value of minimum-width signed type | $-(2^{(N-1)}-1)$, or lower |
| INT_LEAST*N*_MAX | Maximum value of minimum-width signed type | $2^{(N-1)}-1$, or higher |
| UINT_LEAST*N*_MAX | Maximum value of minimum-width unsigned type | $2^{N}-1$, or higher |
| INT_FAST*N*_MIN | Minimum value of fastest minimum-width signed type | $-(2^{(N-1)}-1)$, or lower |
| INT_FAST*N*_MAX | Maximum value of fastest minimum-width signed type | $2^{(N-1)}-1$, or higher |
| UINT_FAST*N*_MAX | Maximum value of fastest minimum-width unsigned type | $2^{N}-1$, or higher |
| INTPTR_MIN | Minimum value of intptr_t | $-(2^{15}-1)$, or lower |
| INTPTR_MAX | Maximum value of intptr_t | $2^{15}-1$, or higher |
| UINTPTR_MAX | Maximum value of uintptr_t | $2^{16}-1$, or higher |

- Where N is one in 8, 16, 32, 64, or any other type width supported by the library.
  Only the macros corresponding to types supported by the library are defined.

▽ Limits of other types :

- Limits of other standard integral types:

| Macro | description | defined as |
|---|---|---|
| SIZE_MAX | Maximum value of size_t | $2^{64}-1$, or higher |
| PTRDIFF_MIN | Minimum value of ptrdiff_t | $-(2^{16}-1)$, or lower |
| PTRDIFF_MAX | Maximum value of ptrdiff_t | $2^{16}-1$, or higher |
| SIG_ATOMIC_MIN | Minimum value of sig_atomic_t | if sig_atomic_t is signed: -127, or lower<br>if sig_atomic_t is unsigned: 0 |
| SIG_ATOMIC_MAX | Maximum value of sig_atomic_t | if sig_atomic_t is signed: 127, or higher<br>if sig_atomic_t is unsigned: 255, or higher |
| WCHAR_MIN | Minimum value of wchar_t | if wchar_t is signed: -127, or lower<br>if wchar_t is unsigned: 0 |
| WCHAR_MAX | Maximum value of wchar_t | if wchar_t is signed: 127, or higher<br>if wchar_t is unsigned: 255, or higher |
| WINT_MIN | Minimum value of wint_t | if wint_t is signed: -32767, or lower<br>if wint_t is unsigned: 0 |
| WINT_MAX | Maximum value of wint_t | if wint_t is signed: 32767, or higher<br>if wint_t is unsigned: 65535, or higher |

▽ Functions like macros :

- These function-like macros expand to integer constants suitable to initialize objects of the types above:

| Macro | description |
|---|---|
| INTMAX_C | expands to a value of type intmax_t |
| UINTMAX_C | expands to a value of type uintmax_t |
| INT*N*_C | expands to a value of type int_least*N*_t |
| UINT*N*_C | expands to a value of type uint_least*N*_t |

- eg. INTMAX_C(2012)  // expands to 2012LL or similar

# EXCEPTIONS

- Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
- To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.
- An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

```
// exceptions                                          An exception occurred. Exception Nr. 20
#include <iostream>
using namespace std;

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

- A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.
- The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught by that handler.
- Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.
- If an ellipsis ( ... ) is used as the parameter of catch, that handler will catch any exception no matter what the type of the

exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

▷ After an exception has been handled the program, execution resumes after the try-catch block, not after the throw statement!.

▷ It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
  try {
    // code here
  }
  catch (int n) {
    throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

▷ Exception Specifications

▽ Older code may contain dynamic exception specifications. They are now deprecated in C++, but still supported. A dynamic exception specification follows the declaration of a function, appending a throw specifier to it. For example:

*double* myfunction (*char* param) *throw* (*int*);

▽ This declares a function called myfunction, which takes one argument of type char and returns a value of type double. If this function throws an exception of some type other than int, the function calls std::unexpected instead of looking for a handler or calling std::terminate.

▽ If this throw specifier is left empty with no type, this means that std::unexpected is called for any exception. Functions with no throw specifier (regular functions) never call std::unexpected, but follow the normal path of looking for their exception handler.

▽

```
int myfunction (int param) throw(); // all exceptions call unexpected
int myfunction (int param);         // normal exception handling
```

## ▷ Standard Exceptions

▽ The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called std::exception and is defined in the <exception> header. This class has a virtual member function called what that returns a null-terminated character sequence (of type char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

▽

```
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

```
My exception happened.
```

▽ All exceptions thrown by components of the C++ Standard library throw exceptions derived from this exception class. These are:

| exception | description |
|---|---|
| bad_alloc | thrown by new on allocation failure |
| bad_cast | thrown by dynamic_cast when it fails in a dynamic cast |
| bad_exception | thrown by certain dynamic exception specifiers |
| bad_typeid | thrown by typeid |
| bad_function_call | thrown by empty function objects |
| bad_weak_ptr | thrown by shared_ptr when passed a bad weak_ptr |

▽ Also deriving from exception, header <exception> defines two generic exception types that can be inherited by custom exceptions to report errors:

| exception | description |
|---|---|
| logic_error | error related to the internal logic of the program |
| runtime_error | error detected during runtime |

▽ Eg.

```cpp
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

# PREPROCESSOR DIRECTIVES

▷ These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

▷ Macro Definitions

▽ To define preprocessor macros we can use #define. Its syntax is:
#define identifier replacement

▽ This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of identifier by replacement.

▽ #define can work also with parameters to define function macros:
#define getmax(a,b) a>b?a:b

▽ Defined macros are not affected by block structure. A macro lasts until it is undefined with the #undef preprocessor directive:

```
1 #define TABLE_SIZE 100
2 int table1[TABLE_SIZE];
3 #undef TABLE_SIZE
4 #define TABLE_SIZE 200
5 int table2[TABLE_SIZE];
```

▽ Function macro definitions accept two special operators (# and ##) in the replacement sequence:

▸ The operator #, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):
#define str(x) #x
cout << str(test);
is equivalent to : cout << "test";

▸ The operator ## concatenates two arguments leaving no blank spaces between them:
#define glue(a,b) a##b
glue(c,out)<<endl; is equivalent to : cout<<endl;

▽ Preprocessor replacements happen before any C++ syntax check.

## ▷ Conditional Inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

▽ These directives allow to include or discard part of the code of a program if a certain condition is met.

▽ #ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is.

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

▽ #ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined.

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

▽ The #if, #else and #elif (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows #if or #elif can only evaluate constant expressions, including macro expressions.

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

▽ The behavior of #ifdef and #ifndef can also be achieved by using the special operators defined and !defined respectively in any #if or #elif directive:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

## ▷ Line Control (#line)

▽ When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a

line number, so it is easier to find the code generating the error.

▽ The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

#line number "filename"

▽ Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

"filename" is an optional parameter that allows to redefine the file name that will be shown. For example:

```
1 #line 20 "assigning variable"
2 int a?;
```

## ▷ Error Directive (#error)

▽ This directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter:

```
1 #ifndef __cplusplus
2 #error A C++ compiler is required!
3 #endif
```

▽ This example aborts the compilation process if the macro name __cplusplus is not defined (this macro name is defined by default in all C++ compilers).

## ▷ Source File Inclusion (#include)

▽ When the preprocessor finds an #include directive it replaces it by the entire content of the specified header or file. There are two ways to use #include:

```
1 #include <header>
2 #include "file"
```

▽ <> is used to include headers provided by the implementation, such as the headers that compose the standard library (iostream, string, ... ). Whether the headers are actually files or exist in some other form is implementation-defined, but in any case they shall be properly included with this directive.

▽ "" includes a file. The file is searched for in an implementation-defined manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes ("") were replaced by angle-brackets (<>).

## ▷ Pragma Directive (#pragma)

▽ This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma.

▽ If the compiler does not support a specific argument for #pragma, it is ignored - no syntax error is generated.

## ▷ PREDEFINED MACRO NAMES

▽ The following macro names are always defined (they all begin and end with two underscore characters, _):

| macro | value |
|---|---|
| `__LINE__` | Integer value representing the current line in the source code file being compiled. |
| `__FILE__` | A string literal containing the presumed name of the source file being compiled. |
| `__DATE__` | A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began. |
| `__TIME__` | A string literal in the form "hh:mm:ss" containing the time at which the compilation process began. |
| `__cplusplus` | An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler:<br>• `199711L`: ISO C++ 1998/2003<br>• `201103L`: ISO C++ 2011<br>Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above. |
| `__STDC_HOSTED__` | `1` if the implementation is a *hosted implementation* (with all standard headers available)<br>`0` otherwise. |

▽ The following macros are optionally defined, generally depending on whether a feature is available:

| macro | value |
|---|---|
| `__STDC__` | In C: if defined to `1`, the implementation conforms to the C standard.<br>In C++: Implementation defined. |
| `__STDC_VERSION__` | In C:<br>• `199401L`: ISO C 1990, Ammendment 1<br>• `199901L`: ISO C 1999<br>• `201112L`: ISO C 2011<br>In C++: Implementation defined. |
| `__STDC_MB_MIGHT_NEQ_WC__` | `1` if multibyte encoding might give a character a different value in character literals |
| `__STDC_ISO_10646__` | A value in the form `yyyymmL`, specifying the date of the Unicode standard followed by the encoding of `wchar_t` characters |
| `__STDCPP_STRICT_POINTER_SAFETY__` | `1` if the implementation has *strict pointer safety* (see `get_pointer_safety`) |
| `__STDCPP_THREADS__` | `1` if the program can have more than one thread |

▽ Particular implementations may define additional constants.

Eg.

```cpp
// standard macro names
#include <iostream>
using namespace std;

int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " << __FILE__ << ".\n";
  cout << "Its compilation began " << __DATE__;
  cout << " at " << __TIME__ << ".\n";
  cout << "The compiler gives a __cplusplus value of " << __cplusplus;
  return 0;
}
```

```
This is the line number 7 of file /home/jay/stdmacronames.cpp.
Its compilation began Nov  1 2005 at 10:12:29.
The compiler gives a __cplusplus value of 1
```

Edit & Run

# INPUT/OUTPUT WITH FILES

- C++ provides the following classes to perform output and input of characters to/from files:
  - ofstream: Stream class to write on files
  - ifstream: Stream class to read from files
  - fstream: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream.
-

```
1 // basic file operations                          [file example.txt]
2 #include <iostream>                              Writing this to a file.
3 #include <fstream>
4 using namespace std;
5
6 int main () {
7   ofstream myfile;
8   myfile.open ("example.txt");
9   myfile << "Writing this to a file.\n";
10  myfile.close();
11  return 0;
12 }
```

This code creates a file called example.txt and inserts a sentence into it

- Open A File
  - The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream (i.e., an object of one of these classes) and any input or output operation performed on this stream object will be applied to the physical file associated to it.
  - In order to open a file with a stream object we use its member function open:

    open (filename, mode);

    Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

| | |
|---|---|
| ios::in | Open for input operations. |
| ios::out | Open for output operations. |
| ios::binary | Open in binary mode. |
| ios::ate | Set the initial position at the end of the file.<br>If this flag is not set, the initial position is the beginning of the file. |
| ios::app | All output operations are performed at the end of the file, appending the content to the current content of the file. |
| ios::trunc | If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one. |

▽ All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

▽ Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

| class | default mode parameter |
|---|---|
| ofstream | ios::out |
| ifstream | ios::in |
| fstream | ios::in \| ios::out |

▽ For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).

▽ For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

▽ File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

▽ These three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member.

Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

▽ To check if a file stream was successful opening a file, you can do it by calling to member is_open. This member function returns a bool value of true in the case that indeed the stream object is

associated with an open file, or false otherwise:
*if* (myfile.is_open()) { */* ok, proceed with output */* }

## ▷ Closing A File

▽ close member is called to close a file. It flushes associated buffers and closes the file specified.
eg. myfile.close();

▽ Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.

▽ In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

## ▷ Text Files

▽ ios::binary flag is not included in the opening mode.

▽ These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

▽

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

▽

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
      cout << line << '\n';
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

```
This is a line.
This is another line.
```

▽ The value returned by getline is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

## ▷ Checking State Flags

▽ The following member functions exist to check for specific states of a stream (all of them return a bool value):

`bad()`
: Returns `true` if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

`fail()`
: Returns `true` in the same cases as `bad()`, but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

`eof()`
: Returns `true` if a file open for reading has reached the end.

`good()`
: It is the most generic state flag: it returns `false` in the same cases in which calling any of the previous functions would return `true`. Note that `good` and `bad` are not exact opposites (`good` checks more state flags at once).

▽ The member function clear() can be used to reset the flags.

## ▷ Get and Put Stream Positioning

▽ All i/o streams objects keep internally -at least- one internal position:

- ifstream, like istream, keeps an internal get position with the location of the element to be read in the next input operation.
- ofstream, like ostream, keeps an internal put position with the location where the next element has to be written.

- Finally, fstream, keeps both, the get and the put position, like iostream.

▽ These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

- tellg() and tellp()
  - These two member functions with no parameters return a value of the member type streampos, which is a type representing the current get position (in the case of tellg) or the put position (in the case of tellp).
- seekg() and seekp()
  - These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:

```
seekg ( position );
seekp ( position );
```

Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.
The other form for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the get or put position is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an enumerated type that determines the point from where offset is counted from, and that can take any of the following values:

| | |
|---|---|
| ios::beg | offset counted from the beginning of the stream |
| ios::cur | offset counted from the current position |
| ios::end | offset counted from the end of the stream |

- Eg.

```
// obtaining file size                                    size is: 40 bytes.
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin", ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

- streampos is a specific type used for buffer and file positioning and is the type returned by file.tellg(). Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.
- These stream positioning functions use two particular types: streampos and streamoff. These types are also defined as member types of the stream class:

| Type | Member type | Description |
|---|---|---|
| streampos | ios::pos_type | Defined as fpos<mbstate_t>. It can be converted to/from streamoff and can be added or subtracted values of these types. |
| streamoff | ios::off_type | It is an alias of one of the fundamental integral types (such as int or long long). |

- Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.

## ▷ Binary Files

▽ For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data and data is likely not formatted in lines.

▽ File streams include two member functions specifically designed to read and write binary data sequentially: write and read. The first one (write) is a member function of ostream (inherited by ofstream). And read is a member function of istream (inherited by ifstream). Objects of class fstream have both. Their prototypes

are:

```
write ( memory_block, size );
read ( memory_block, size );
```

▽ Where memory_block is of type char* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

▽

```
// reading an entire binary file                    the entire file content is in memory
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

▽ First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file.

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

memblock = *new char*[size];

Right after that, we proceed to set the get position at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

## ▷ Buffers & Synchronization

▽ When we operate with file streams, these are associated to an internal buffer object of type streambuf. This buffer object may

represent a memory block that acts as an intermediary between the stream and the physical file. For example, with an ofstream, each time the member function put (which writes a single character) is called, the character may be inserted in this intermediate buffer instead of being written directly to the physical file with which the stream is associated.

The operating system may also define other layers of buffering for reading and writing to files.

▽ When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream). This process is called synchronization and takes place under any of the following circumstances:

- When the file is closed: before closing a file, all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
- When the buffer is full: Buffers have a certain size. When the buffer is full it is automatically synchronized.
- Explicitly, with manipulators: When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
- Explicitly, with member function sync(): Calling the stream's member function sync() causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.