

C++.cpp

Eileen Rheinboldt-Tran

2021

**Index**

[Syntax 1](#_Toc75593356)

[Omitting Namespace 1](#_Toc75593357)

[Comments 2](#_Toc75593358)

[Declaration 2](#_Toc75593359)

[auto keyword 2](#_Toc75593360)

[decltype 2](#_Toc75593361)

[Variable initialization 2](#_Toc75593362)

[Constants 3](#_Toc75593363)

[Output (print text) 3](#_Toc75593364)

[User input 3](#_Toc75593365)

[while loop 3](#_Toc75593366)

[do/while loop 4](#_Toc75593367)

[for loop 4](#_Toc75593368)

[if/else statements 4](#_Toc75593369)

[Short hand if…else 4](#_Toc75593370)

[Another way to write 5](#_Toc75593371)

[Directives to the processor 5](#_Toc75593372)

[using 5](#_Toc75593373)

[Basic types in C++ 5](#_Toc75593374)

[integer 6](#_Toc75593375)

[floating types 6](#_Toc75593376)

[Character 7](#_Toc75593377)

[Boolean 8](#_Toc75593378)

[volatile keyword 8](#_Toc75593379)

[Operators and expressions 8](#_Toc75593380)

[implicit conversions 9](#_Toc75593381)

[Short circuit in the evaluation of && and ¦¦ 11](#_Toc75593382)

[The ordinary assignment operator 11](#_Toc75593383)

[The cast operator 11](#_Toc75593384)

[Conditional operator 11](#_Toc75593385)

[The sizeof operator 12](#_Toc75593386)

[switch statement 12](#_Toc75593387)

[break, continue and goto 13](#_Toc75593388)

[the break instruction 13](#_Toc75593389)

[The continue instruction 13](#_Toc75593390)

[The goto instruction 13](#_Toc75593391)

[Functions 13](#_Toc75593392)

[Passing arguments by value 14](#_Toc75593393)

[Passing arguments by reference 15](#_Toc75593394)

[Transmission by reference of a return value 16](#_Toc75593395)

[Global variable 17](#_Toc75593396)

[Local variables 17](#_Toc75593397)

[Default arguments 18](#_Toc75593398)

[Over-definition of functions 18](#_Toc75593399)

[The reference in general 20](#_Toc75593400)

[Functions with variable arguments 21](#_Toc75593401)

# Syntax



Line 1 : #include <iostream> is a directive taken into account before compiling the program. It’s a header file library that lets us work with input and output objects (flows), therefore, in particular to cin and cout .

Line 2 : using namespace std ; means that we can use names for objects and variables from the standard library.

Line 3 : A blank line. C++ ignores white space.

Line 4 : int main(), always appear in a C++ program, it’s a function. Any code inside its curly brackets { } will be executed.

Line 6 : cout (pronounced “see-out”) is an object used together with the *insertion operator* (<<) to output/print text. In our example it’ll output “Hello World!”.

Line 7 : return 0 ends the main function.

Line 8 : Do not forget to add the closing bracket } .

Note 1 : Every C++ statements ends with a semicolon ; .

Note 2 : the body of int main() could also been written as:

int main () { cout << ”Hello World!” ; return 0 ; }

## Omitting Namespace

The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for some object.



# Comments

Single line comments start with 2 forward slashes // .

Multi-line comments start with /\* and ends with \*/ .

# Declaration

The declarations of the types of variables are mandatory . Variables must be declared before they can be used.



## auto keyword

The type of variable is deduced from the expression used for its initialization.



The const and constexpr are not considered :



## decltype

We can also declare that a variable is of the same type as another :



This time the const and constexpr are kept.

# Variable initialization

We can initialize a variable in different ways :



Or with the parenthesized notation :



Or notation with braces :



⚠Doesn’t allow degrading conversions

Forms of initialization that we use the equal sign are called copy initialization and others direct initialization.

## Constants

A value of a variable can be unmodifiable and for that we use either const or the constexpr modifier.

The primary difference between const and constexpr variables is that the initialization of a const variable can be deferred until run time. A constexpr must be initialized at compile time. All constexpr variables are const.

# Output (print text)

The cout object, together with the << operator, is used to output values/print text.



# User input

cin is a predefined variable that reads data from the keyboard with the extraction operator >> .



# while loop

The while loop loops through a block of code as long as a specified condition is true.



## do/while loop

Variant of the while loop, it executes the code block once before checking if the condition is true, then it’ll repeat as long as the condition is true.



# for loop

When we know exactly how many times we want to loop through a block of code, we use the for loop instead of a while loop.



Statement 1 : is executed (1 time) before the execution of the code block.

Statement 2 : condition for executing the code block.

Statement 3 : is executed (every time) after the code block has been executed.

# if/else statements

Syntax :



## Short hand if…else

Syntax : variable = (condition ) ? expressionTrue : expressionFalse ;

## Another way to write



⚠ An else always refers to the last if encountered to which an else has not yet been assigned.

# Directives to the processor



These are directives that are taken into account before the translation (compilation) of the program, by a program called “processor” (or precompiler).

They must be written one per line and start at the beginning of the line and are usually placed at the top of the program.

These directives ask to introduce (before compilation) instruction (in C++).

For the moment we note that:

* iostream contains statements relating to streams, therefore, in particular, to cin and cout, as well as the operators << and >>.
* cmath contains declarations for mathematical functions (inherited from C).

# using



The using statements is used precisely to indicate that we are going “in this std namespace”. We could do without this instruction, provided we prefix all the symbols concerned by std:: (ex: std::cout or std::cin ), but make the program less readable.

# Basic types in C++

The types char, int, float are said to be “scalar” or “simple”, because, at a given moment, a variable of such a type contains a single value. They are opposed to “structured” type (we also say “aggregated”) which correspond to variables which, at a given moment, contain several values ( of the same type or not).

Here, we’ll study the types of bases, therefore scalar from which all the other can be constructed, called “derived types”, whether it is :

* structured types like string, vector, array and class
* other simple types like pointers or enumeration

4 types of bases :

* integer number (int)
* floating number (float or double)
* character (char)
* Boolean values, true or false (bool)

## integer

There can be up to 4 different integer sizes :

* short int (abbreviate in “short”)
* int
* long int (abbreviate in “long”)
* long long int (abbreviate in “long long”)

### unsigned integer types

Unsigned integers only hold non-negative whole number. We use the unsigned qualification and these types are often reserved for the direct manipulation of a “binary pattern” and it is advisable to avoid mixing signed and unsigned integers in the same expression.

### notation of integer literal constants

The easiest way is to write it in decimal form, with or without sign:

8 -1394 +50

Otherwise, we can also write in octal (base 8) ox hexadecimal (base 16) notation.

## floating types

The floating types allow to represent, in an approximative way, a part of the real number by being inspired by scientific notation.

There are 3 types of floats (corresponding to different sizes):

* float
* double
* long double

### notation of floating literal constants

They can be written according to one of the two notation: decimal or exponential.

Decimal notation must include a period : ex= 12.43 -0.38 -.95 4. .27

Exponential notations uses the letter e (or E) to introduce an integer exponent (power of 10): ex= 4.25e+4 527.7E-33 48e13 48.0E13

By default, all floating-point literal constants are created by the compiler in type *double*. It’s however possible to impose on a floating constant :

* to be a *float* type, by following his writing with the letter F (or f) :

+memory space -precision

* to be a *long double* type, by following his writing with the letter L (or l) :

-memory space +precision

## Character

The char data type is used to store a single character who must be surrounded by a single quote. ex= ‘a’ ‘2’ ‘C

We can use ASCII values to display certain characters, either in octal or hexadecimal form.

Ex= ‘A’ or ‘\x41’ or ‘\101’

Some non-printable characters have a conventional representation using the backslash character “\” :

|  |  |  |  |
| --- | --- | --- | --- |
| Notation in C++ | Code ASCII | abbreviation | Usual meaning |
| \a | 07 | Bel | Alert or audible bell |
| \b | 08 | BS | Backspace |
| \f | 0C | FF | Page breach (form feed) |
| \n | 0A | LF | Line feed |
| \r | 0D | CR | Carriage return |
| \t | 09 | HT | Horizontal Tab |
| \v | 0B | VT | Vertical tab |
| \\ | 5C | \ |  |
| \’ | 2C | ‘ |  |
| \” | 2Z | “ |  |
| \? | 3F | ? |  |

## Boolean

A Boolean is declared with the bool keyword and has 2 values : true and false.

When the value is returned true=1 and false=0.

# volatile keyword

Not very used. Is manly used when the environment outside the program can act directly on memory location.

The volatile keyword prevents the compiler from performing optimization.



If the value j is not modified in the loop, the compiler will translate these instructions as if we had written:



On the other hand, if the variable has been declared volatile, the compiler will keep the affection in question in the loop.

# Operators and expressions

|  |  |  |
| --- | --- | --- |
| category | operators | associativity |
| Scope resolution | :: (global scope-unary), :: (class scope-binary), :: (namespace scope-binary) | 🡪 |
| reference | ( ), [ ], 🡪, . | 🡪 |
| unary | ++ (postfixed), -- (postfixed), typeid, cast, dynamic\_cast, static\_cast, reinterpret\_cast, const\_cast | 🡨 |
| unary | +, -, ++ (prefixed), --(prefixed), !, ~, \*, &, sizeof, new, new[], delete, delete[], noexcept | 🡨 |
| selection | 🡪\*, .\* | 🡪 |
| arithmetic | \*, /, % | 🡪 |
| arithmetic | +, - | 🡪 |
| shift | <<, >> | 🡪 |
| relational | <, <=, >, >= | 🡪 |
| relational | ==, != | 🡪 |
| bit manipulation | & | 🡪 |
| bit manipulation | ^ | 🡪 |
| bit manipulation | ¦ | 🡪 |
| logic | && | 🡪 |
| logic | ¦¦ | 🡪 |
| Conditional (ternary) | ?, : | 🡪 |
| assignment | =, +=, -=, \*=, /=, %=, &=, ^=, ¦=, <<=, >>= | 🡨 |
| exception | throw | 🡨 |
| sequential | , | 🡪 |

This is a table of the complete list of operators of the C++ language in order of decreasing priority.

⚠Difference ++i and i++:

++i = increment the i’s value and return the incremented value

i++ = increment the i’s value but return the original value of that i held before incremented

## implicit conversions

### The usual type conversion

A conversion such as int 🡪 float is called a <<type fit conversion>>. Such a conversion is done by following a hierarchy of such to respect the integrity of the data, namely : int 🡪 long 🡪 long long 🡪 float 🡪 double 🡪 long double

We can of course directly convert an int to a double ; however, we cannot convert a double to float or int.

The choice of conversion to be implemented is made by considering the operands concerned one by one and not the expression as a whole. For example, if n is of type int, p of type long and x of type float, the expression: n \* p + x

will be evaluated according to this scheme:



### Usual digital promotions

Arithmetic operators are not defined for short, char, and bool types. So C++ excepts that any values of any of these 3 types appearing in an expression is first convert to an int. We then speak of “digital promotion” (or “systematic conversion”)

#### char type case=

a value of type char can be considered in 2 ways:

* like the character concerned: a, z, end of line
* like the code of this character, that is to say the pattern of 8 bits; for example, in the ASCII code, the character E is represented by the binary pattern 01000101, to which the number 65 can be matched.

⚠All machines do not use the same code for character, the integer associated with a character will not always be the same!

#### bool type case=

the type bool provides an integer value of 0 or 1.

### Conversion in presence of unsigned type

It’s strongly recommended not to mix signed and unsigned types.

## Short circuit in the evaluation of && and ¦¦

The 2 operators && and ¦¦ have an interesting property in C++ under the name “short-circuit”: their second operand (the one to the right of the operator) is only evaluated if the knowledge of its value is required to decide whether the expression is true or false.

Knowledge of this property is essential in order to master “constructions” in which one of the operands performs an action ( in addition to having a value) :



## The ordinary assignment operator

### concept of lvalue

lvalue (left value) = represents the address of a location. (see in green =)



rvalue (right value) = represents a value. (see in green =)



### Right to left affection

This assignment operator has right-to-left associativity which allows :



## The cast operator

We can force the conversion of any expression into a type of our choice, using the cast operator :



## Conditional operator

The following operation :



can be translate into = 

## The sizeof operator

Provides the size in bytes.



# switch statement

Syntax:



If a label corresponds to the value obtained, we connect to the instruction appearing after this label.

If no value would have been encountered before, the instruction connected to the default keyword will be run.



⚠**Here the break instruction is fundamental !!!**

In the example that follows some break statements are intentionally omitted :



# break, continue and goto

## the break instruction

It’s used to interrupt the progress of the loop, by passing to the instruction which follows this loop.

## The continue instruction

It allows us to go to the next loop prematurely.

## The goto instruction

It allows connection to any location in the program.



⚠It’s strongly recommended to only use this instruction in exceptional circumstances and to avoid any branching inside a block.

# Functions

A function is a block of code which only runs when it’s called.

We can pass data known as parameters, into a function.



The return instruction specifies the value that the function will provide at the end of the job. If the type is different from the type in the header the compiler will automatically set up conversion instruction.

When a function does not return a result, it is specified, both in its declaration and its header, using the void keyword.

Example of the declaration:



Example of the header:



C++ allows recursion.

## Passing arguments by value

This transmission mode does not allow a function to modify the value of an argument, the function works on copies of the values passed as arguments.

This is the default mode used in C++.



## Passing arguments by reference

The reference of a variable is its address and not is value.



The int &a notation means that a is an information of type int transmitted by reference.

Transmission by reference has 2 distinct properties :

* it avoids the copying of information 🡪 saving time
* It allows a function to work directly on the effective argument transmitted and, therefore, to modify its value.

⚠Passing by reference forces an effective argument to be a modifiable lvalue of the type intended for the dumb argument.





If we don’t modify the value :



The following calls are correct:



## Transmission by reference of a return value



A call to f will no longer cause the return transmission of not more of a value, but of the reference ( address) of n .

Here again we avoid a copy so we save time.

⚠However n cannot be a local variable to f, otherwise it will be destroyed when f exists. And n must be an editable lvalue.





## Global variable

Several functions can share common variables that we qualify as global. But this practice is not recommended.

To do this, it suffices to declare the variable outside the functions. Their scope (or range of validity) is limited to the part of the source program which follows their declaration.

## Local variables

They are defined within a function so their scope is limited to this one.

Local variables have a lifetime limited to that of an execution of the function in which they appear. More precisely, their location isn’t permanently defined. We translate this by saying that the allocation class of the variables is automatic.

### Static local variables

It is possible to assign a permanent location to a local variable and thus keep its value from one call to another. We just have to declare it using the static keyword.



⚠Do not confuse a static local variable and a global variable, the scope of such a variable remains limited to the function !!!

## Default arguments



⚠Default values must be set in the function declaration and not in its definition !

Inside : void fct(int, int=12);

int=12 specifies to the compiler that, in the absence of this second argument, it will be necessary to act as if the call had been made with this value.

⚠The default values must be the last of the list !

## Over-definition of functions

To be able to use several functions with the same name, we need another criterion (other than the name): the type of the argument





### Example of choice of an over-defined function

The previous example had exactly one of the types provided in the prototypes as argument, what if we called the function with different types on multiple arguments.

#### Example 1



#### Example 2



Given its ambiguity, the last call leads to a compilator error.

#### Example 3





#### Example 4



Compile error because C++ didn’t plan to distinguish int from const int.

#### Example 5



This time, the distinction between *int &* and *const int &* is justified.

Indeed, we can imagine that trial I modifies the value of the lvalue of which it receive the reference, while trial II does nothing.

We can also:



## The reference in general

It’s possible to declare an identifier as a reference of another variable:



⚠It is not possible to declare a reference without initializing it.



Once declared (and initialize) a reference can no longer be modified.

If, having declared *int &p=n*; we write *p=q*, it’s obligatory a question of the affection of the value of q at the location of reference p, and not of the modification of the reference q.



We can’t initialize a reference with a constant : int &n = 3; ✘

In other hand, it’s possible to define references to constant which can then be initiated by constants: const int &n = 3; ✔

## Functions with variable arguments

### Initializer\_list type

C++11 defines a so-called “generic” type, denoted initializer\_list<T>, T being any type. A variable of such type is formatted by a list of non-modifiable values of a type implicitly compatible with T.



### Application to a function with variable number of arguments

We can therefore use initializer\_list to pass to a function a variable number of arguments of a type compatible with the given type.

So, a header function:



can be called with a list of values compatible with float (char, int, long,…)





## Separate compilation

Separate compilation is the process in which one brings together several independently compiled source file designed by the user.

Syntax: #include “filename.h”

⚠” “ and not < >

⚠usually the .h extension

⚠Do not confuse the header files which contain the function declarations with the object module which, in turn, contain the corresponding executable code !!!

### Mechanism of over-definition of function

We have previously seen that the compiler chooses the “right function”, reasoning on one file at the time. But it’s quite possible:

* to first compile a source file containing the different definitions of a function ; we can even explode these over definitions in multiple sources files.
* to use these functions later in another source file by simply providing the prototypes.

However, for this, the link editor must be able to make the link between the choice made by the compiler and the “good function” appearing in another object module. This recognition is based on the modification, by the compiler, of the “external” names of the functions. This mechanism applies to all functions (over defined or not).

We therefore see that a problem arises, as soon as we want to use in a program in C++ a function written and compiled in C (or in another language using the same function calling conventions such as Assembly or Fortran). Such a function will not have its name modified according to the mechanism mentioned.

However, there is a solution: declare such a function to precede its prototype with the extern “C” mention. For example, if we wrote and compiled in C a header function:

double fct(int n, char c);

and we want to use it in a C++program, we just need to provide its prototype as follow:

extern “C” double fct(int,char);

There is a « collective » form of the extern “C” declaration which look like this:



### Separate compilation and global variable

The scope of a global variable seems limited to the source file in which it was defined. So, suppose we compile these 2 sources file separately :

 

It doesn’t seem possible, in function *fct2* and *fct3* to refer to the global variable x declared in the first source file.

In fact, C++ provides a declaration allowing to specify that a global variable has already been defined in another source file. This is done using the extern keyword. So, by preceding our second source file with the declaration: extern int x ; , it becomes possible to mention the global variable x in the function fct2 and fct3.

⚠This extern declaration doesn’t perform a variable slot reservation! It only specifies that the variable x is defined by elsewhere and it indicates its type!

To make a global variable inaccessible outside the source file where it was defined, we just use a static declaration.



## The inline specification

As we would expect, the executable code corresponding to a function is generated only once by the compiler. However, for each call to this function, the compiler must provide not only the connection to the corresponding executable code, but also useful instructions for establishing communication between the calling program and the function.

When code efficiency becomes an important criterion, C++ allows us to save time in calling functions, at the expense of code size thanks to the inline specification. But this will consume an increasing amount of memory with the number of calls.



The purpose of the function norm is to calculate the norm of a vector with 3 components that we provide to it as an argument.

( The 3 components v1= 0 1 2

v2= -1 2 3 )

## completing a program

We have already seen that we can stop the execution of the main function with a return statement, accompanied by an integer value indicating whether the execution took place correctly (value 0) or not (non-zero value).

However, certain situation, in general in the event of important error, we would like to be able, from a function, to end the execution of the program, without having to “go up” the stack of various calls. It’s then possible to call the standard exit function (prototype in cstdlib), to which we provide, here again, an integer that can be used as a report of the progress of the program.

There is another abort function (prototype in cstdlib), used for anomalous purpose. This terminate the program abruptly, without any intervention at the level of files or object.

Finally, it is possible to request that one or more function be executed at the end of program, using the atexit function. If more than one atexit functions are registered, they are executed in the reverse order.



# The type string

## Declaration and initialization

During its declaration, we can initialize a variable of type string in different ways:

* with a literal string :



* By repeating a given number of time a given character :



* With a series of characters :



* With another string :



The use of the type string requires the inclusion of the header file <string> which is automatically included in <iostream>.

⚠Be careful with the use of auto.

## The size and empty functions

size allows us to know the number of characters at a given moment in a string and empty to know if the string contained in a variable of type string is empty.



## String concatenation

The concatenation of 2 operands is the string obtained by putting the characters of the first- and second-string end to end.



## Access to the characters of a string

We can access the characters in a string by referring to its index number inside square brackets [ ].



These notations are modifiable lvalues (unless the string has been declared constant).



For reason of efficiency, no control over the value of the index is made during execution.



There is a form of the for statement applicable to sequences:



The loop instruction, written here for a c element, will be applied to each of the characters in s. c designates a copy of a character from the string. Thus, the instruction:



would have no effect on the content of s since it would only modifies copies. But it’s still possible to ask to work directly on each character of the string using a transmission by reference :

