Introduction to C++

Emanuele Giona
Department of Computer Science, Sapienza University of Rome

Internet of Things A.Y. 2021/22

Prof. Chiara Petrioli
Department of Computer, Control and Management Engineering, Sapienza University of Rome

Brief History

- Bjarne Stroustrup invented C++ as “C with classes” in 1979, at Bell Labs
- Main focus: efficient and flexible language similar to C, while also providing high-level features for program organization
- It was renamed to C++ in 1983
- Many standards: C++98, C++03, C++11, C++14, C++17, C++20, *C++23*

Feb 2022	Feb 2021	Change	Programming Language	Ratings	Change
1	3		Python	15.33%	+4.47%
2	1		C	14.08%	-2.26%
3	2		Java	12.13%	+0.84%
4	4		C++	8.01%	+1.13%
5	5		C#	5.37%	+0.93%
6	6		Visual Basic	5.23%	+0.90%
7	7		JavaScript	1.83%	-0.45%
8	8		PHP	1.79%	+0.04%
9	10		Assembly language	1.60%	-0.06%
10	9		SQL	1.55%	-0.18%

TIOBE index

Applications of C++

C++ is widely used in several applications such as desktop applications, video games, databases, web servers, although its original scope was intended to be systems programming and embedded systems.

Its popularity can be attributed to its stability, solid performance, high compatibility and scalability, as well as the many useful features provided by the language.

Main features comprise:

- Middle-level programming language (both high-level and low-level capabilities)
- Multi-paradigm programming
  - Imperative: programs expressed via statements that change the internal state
  - Object-oriented: programs expressed via concepts of classes, objects, polymorphism, inheritance, abstraction, encapsulation
  - Functional: programs expressed via application and composition of functions (see Declarative Programming)
- Direct memory manipulation (explicit pointers, dynamic memory allocation, etc.)

Why C++ for the Internet of Things?

Why C++ for the Internet of Things?

- Internet of Things applications often make use of embedded systems
- Such devices usually are resource-constrained (processing power, memory limits, etc.)
- Access to low-level features of a device allows for full exploitation of its capabilities

2. Working with C++

Compilers

C++ is a compiled programming language: what does this mean?

Computes can only execute programs expressed in machine language, in which instructions consist of *0*s and *1*s.
In order to obtain code in this way, programming languages can either be compiled or interpreted.

Compiled Languages

Programs are translated to machine language all at once.

Compilation is only needed once, unless source code has to be modified.

Capable of detecting some errors before execution occurs.

RAM only contains program state and no source code at all.

Compiled programs usually execute faster.

Interpreted Languages

Programs are translated to machine language one instruction at a time.

Program execution occurs by loading and translating instructions at the moment of execution, every time.

All errors can only be detected at execution time.

Part of the source code is stored in RAM for the interpreter to translate.

Interpreted programs usually run slower.

Compilers and Operating Systems

Windows

Install an Integrated Development Environment (IDE), such as *Dev-C++*, *C++Builder*, *Visual Studio Code*, *Codelite*, etc. Compile a program simply using the visual interface within the chosen IDE.

Mac

Install *Xcode* with gcc/clang compilers, or *Visual Studio Code*, etc. Compile a program by launching a Terminal and using one of the following commands:

```
g++ -std=c++11 example.cpp -o example_exec
clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_exec
```

Linux

Most Linux distributions ship with GNU Compiler Collection (gcc) already installed. Compile a program by launching a Terminal and using the following command:

```
g++ -std=c++11 example.cpp -o example_exec
```

3. Basic Syntax

Generic syntax

- Source code is usually contained in files with .cpp extension
- Comments in C++ can either be single lines or span multiple lines

```
// This is a single-line comment
/* This comment spans
multiple lines instead */
```

- Each instruction must end in a semicolon ;
- Source code is case sensitive: hello ≠ hEllo
- An identifier is the name for variables, functions, classes, or any other user-defined item
  - Consists of a sequence of alphanumeric characters
  - Cannot start with a digit
  - Cannot contain punctuation, special characters, or spaces
  - Should not start with single or double underscore (_) characters

Valid	Error
`Count`	`0count`
`C0unt`	`C@unt`
`c__`	`count+`

Generic syntax

Additionally, there are some reserved keywords that cannot be used as identifiers, despite fulfilling the syntax rules.

A - C	D - P	R - Z
alignas (since C++11)	decltype (since C++11)	reflexpr (reflection TS)
alignof (since C++11)	default (1)	register (2)
and	delete (1)	reinterpret_cast
and_eq	do	requires (since C++20)
asm	double	return
atomic_cancel (TM TS)	dynamic_cast	short
atomic_commit (TM TS)	else	signed
atomic_noexcept (TM TS)	enum	sizeof (1)
auto (1)	explicit	static
bitand	export (1) (3)	static_assert (since C++11)
bitor	extern (1)	static_cast
bool	false	struct (1)
break	float	switch
case	for	synchronized (TM TS)
catch	friend	template
char	goto	this
char8_t (since C++20)	if	thread_local (since C++11)
char16_t (since C++11)	inline (1)	throw
char32_t (since C++11)	int	true
class (1)	long	try
compl	mutable (1)	typedef
concept (since C++20)	namespace	typeid
const	new	typename
consteval (since C++20)	noexcept (since C++11)	union
constexpr (since C++11)	not	unsigned
constinit (since C++20)	not_eq	using (1)
const_cast	nullptr (since C++11)	virtual
continue	operator	void
co_await (since C++20)	or	volatile
co_return (since C++20)	or_eq	wchar_t
co_yield (since C++20)	private	while
	protected	xor
	public	xor_eq

Note that and, bitor, or, xor, compl, bitand, and_eq, or_eq, xor_eq, not, and not_eq (along with the digraphs <%, %>, <:, :>, %:, and %:%:) provide an alternative way to represent standard tokens.

In addition to keywords, there are *identifiers with special meaning*, which may be used as names of objects or functions, but have special meaning in certain contexts.

final (C++11)
override (C++11)
transaction_safe (TM TS)
transaction_safe_dynamic (TM TS)
import (C++20)
module (C++20)

The following tokens are recognized by the preprocessor when in context of a preprocessor directive:

if	ifdef	include	defined	export (C++20)
elif	ifndef	line	__has_include (since C++17)	import (C++20)
else	define	error	__has_cpp_attribute (since C++20)	module (C++20)
endif	undef	pragma		

They have special meanings associated to them, indicating specific behavior to the compiler.

Variables

A variable is a portion of memory to store a value, to which a name and a type are associated. Names are used to distinguish among multiple variables, whereas types determine the meaning of the stored values as well as operations performed on them.

Fundamental data types are basic types directly implemented by C++ that represent the lowest level storage units natively supported by most systems and be classified into:

- Character types (e.g. `char`)
  Can store a single character – `‘A’` or `‘f’`
- Numerical integer types (e.g. `int`)
  Can store a whole number – 42 or `65535`
- Floating-point types (e.g. `float`)
  Can store a single-precision real number – `3.14` or `-2.71`
- Boolean type (e.g. `bool`)
  Can store a logical value – `true` or `false`

Types and size in memory

The type of a variable determines the size of the memory portion used to store the value of the variable itself.

Group	Type	Minimum size
Character	`char`	At least 8 bits
Numerical integer	`short`	At least 16 bits
	`int`	At least 16 bits
	`long`	At least 32 bits
	`long long`	At least 64 bits
Floating point	`float`	At least 32 bits, of which >6 significant
	`double`	At least 64 bits, of which >15 significant
	`long double`	At least 64 bits, of which 15/18/33 significant (depends on size)
Boolean	`bool`	N/A

Signed and unsigned types

Unless specified otherwise, numeric integer types are signed: this allows to store both negative and positive values in variables of these types. By explicitly marking a variable unsigned, it will only be capable of representing positive values (≥ 0).
This is simply done by preceding the type with the keyword (`unsigned short`, etc.).

Quick quiz

- Can you store `-1` in a variable of type `unsigned short`?
- Can you store `65535 + 1` in a variable of type `int`?
- Can you store `65535 + 1` in a variable of type `unsigned int`?
- Can you define a variable of type `unsigned float`?

Signed and unsigned types

Unless specified otherwise, numeric integer types are signed: this allows to store both negative and positive values in variables of these types. By explicitly marking a variable unsigned, it will only be capable of representing positive values (≥ 0).
This is simply done by preceding the type with the keyword (`unsigned short`, etc.).

Quick quiz

- Can you store `-1` in a variable of type `unsigned short`? → No, 65535 will be stored instead.
- Can you store `65535 + 1` in a variable of type `int`? → Theoretically no, but usually yes.
- Can you store `65535 + 1` in a variable of type `unsigned int`? → Yes.
- Can you define a variable of type `unsigned float`? → No, compile-time error!

Type-related errors may be spotted at compile time and others at runtime, but silent failures can happen too!

Literals

Literals are data used for representing fixed values, to which no other value can be assigned.

- Integer literals
  Decimal: `7`, `-36`, … | Octal: `021`, `054`, … | Hexadecimal: `0xff`, `0x345`, …
- Floating-point literals
  `42.72f`, `0.000003435`, `-0.3E6`
- Boolean literals
- Character literals
  `'e'`, `'G'`, `'9'`, `'\n'`, `'\t'`
- String literals
  `"hello world"`, `"x"`, `"input:\n"`

Escape Sequences	Characters
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null Character

Literals are used during variable assignments or instruction evaluation.

Creating variables

A simple variable definition consists of:

```
int     x, y, z = 5     ;
float   dist            ;
bool    outcome = false ;
char    newLine = '\n'  ;
```

Type specifier

1+ names separated by commas

Ends with a semicolon

Assignment (optional)

Quick quiz

➢ What is stored in variables x, y, and z?

Creating variables

A simple variable definition consists of:

```
int     x, y, z = 5      ;
float   dist             ;
bool    outcome = false  ;
char    newLine = '\n'   ;
```

Type specifier

1+ names separated by commas

Ends with a semicolon

Assignment (optional)

Quick quiz

- What is stored in variables x, y, and z? → z contains 5, but values in x and y are compiler-dependant.

In order to avoid undefined behavior, make sure to initialize variables before using them!

Constants and details on variables creation

Constant variables are used whenever the value stored in a variable must not be changed after *definition*, and creating them is simply done by preceding the type specifier with the `const` keyword:

```
const double pi = 3.14f;
```

Declaration, definition, and initialization

These concepts are distinct in C++ and they refer to variables as well as functions, classes, and more. In the scope of variables initialization is equivalent to a value assignment through the = operator, while definition has no real meaning. Declaration of a variable instead refers to the introduction of a *new name* in the program, without necessarily specifying a value in the same instruction.
Unless particular cases, constant variables must be declared and initialized otherwise a compile error will be raised.

Differences between declaring and defining something are more important in the case of functions and classes.

Variables of unknown type?

Starting from C++11, the language supports the creation of variables without explicitly using a type specifier. This is done through type deduction and the `auto` keyword has been introduced to use as a type placeholder.

```
int counter = 0;
auto missing = counter;
auto next;
```

Variable `missing` will be created of the same type of its initialization value, in this case `int`, and will be assigned the value stored in `counter`.

However this program will not compile: variable `next` is declared as `auto` but no initialization has taken place and thus the compiler does not know how much memory it should allocate for it.

4. Basic Input/Output

Streams

In C++, a stream is a flow of data into or out of a program in a sequential fashion: operations regarding screen output, keyboard input, or I/O from/to files are implemented by means of streams.

The standard `iostream` library is needed to leverage such features:

```
#include <iostream>
```

This library provides classes and operators to effectively implement the necessary I/O operations.

Standard output

By default, the standard output provides data to the screen:

```
std::cout << “This string literal will be printed on screen”;
```

Standard output | Insertion operator | String that will be printed on screen

All kinds of literals and variables can be passed to the standard output. The insertion operator << can be used with all kinds of streams: writing to a file is done in the same way as printing characters to screen. Multiple insertions are allowed too:

```
std::cout << “X: “ << x << “\nY: “ << y;
std::cout << std::endl;
```

Both the ‘\n’ literal and std::endl add new line characters, but std::endl should be used carefully

Potential output:

```
X: 4.5
Y: 2
```

Standard input

By default, the standard input retrieves data from the keyboard:

```
int counter;
std::cin >> counter;
```

Standard input — `std::cin`
Extraction operator — `>>`
Variable storing the extracted value — `counter`

The program execution will be stopped until the user presses on the Enter / Return key; when execution starts again, data will be transferred to the program.

Data coming from user input should be used carefully and only after an appropriate data validation phase.

5. Program Structure

Hello World in C++

Copy-pasting a C++ Hello World example in your favorite IDE will probably look like this:

```
#include <iostream>

int main(){
    std::cout << “Hello World!” << std::endl;
    return 0;
}
```

Hello World in C++

```
#include <iostream>

int main(){
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Library/file include directives section: the I/O library is needed to print Hello World!

Function called `main` returning values of type `int`

Return value of the program: the integer literal `0` as return value represents a successful execution

Hello World in C++


```
#include <iostream>

int main(){
    std::cout << “Hello World!” << std::endl;
    return 0;
}
```

std is a namespace for the standard library, and :: is the scope resolution operator

Adds a new line character and flushes the buffer

Standard output stream

String literal to be printed on screen

Hello World in C++

```
#include <iostream>

int main(){
    std::cout << “Hello World!” << std::endl;
    return 0;
}
```

This function contains 2 statements, one per line

A semicolon in placed at the end of each statement

Hello World in C++

```
#include <iostream>

int main(){
    std::cout << “Hello World!” << std::endl;
    return 0;
}
```

= Equivalent!

```
#include <iostream>
int main() { std::cout << “Hello World” << std::endl; return 0; }
```

Include directives must be one per line and each statement must be succeeded by a semicolon.

The `using` directive

Since the `std` namespace has many functions implemented in itself, in order to avoid repeating it at every invocation of such functions and objects the `using` directive can come in handy:

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

int main(){
    cout << "Hello World!" << endl;
    return 0;
}
```

Every name declared in the target namespace (std) is introduced into the nearest namespace containing both std and user-declared namespace, or the global namespace

As a result of this, name collisions may arise and a compile-time error will be raised: be careful with names when declaring functions and variables in conjunction with the `using` directive.

Header and implementation files

C++ programs may be split in two separate files for organization purposes: header and implementation files.

Header files (extensions: .h, .hpp)

- ✓ New namespaces
- ✓ Function, class declarations
- ✓ Macros
- ✓ Global variables
- ✓ `include` directives
- ✓ Include guards
- ✓ Default arguments for functions

- × `using` directives
- × Non-`const` variables
- × Unnamed namespaces

Implementation files (extensions: .cpp)

- ✓ Function definitions
- ✓ `const` variables initialization
- ✓ `include` directives
- ✓ `using` directives

- × Function, class declarations
- × Macros
- × Global variables

This split allows to distribute libraries without the need of releasing the full source code, as compilers only need declarations in header files and implementations can be provided as pre-compiled objects.

Hello World over two files

example.h

```
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

#include <iostream>

int main();

#endif
```

example.cpp

```
#include “example.h”

using namespace std;

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

Hello World over two files

example.h

```
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

#include <iostream>

int main();

#endif
```

Preprocessor directives that act as include guards, preventing circular includes from causing issues during compilation

example.cpp

```
#include “example.h”

using namespace std;

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

Double quotes (“) are used for user-defined header files

Hello World over two files

example.h

```
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

#include <iostream>

int main();

#endif
```

This directive can be used to define macros: the preprocessor simply replaces all occurrences of a macro with the code that is associated with it

In this case, it is only used to check whether this file has been included before, thus avoiding double declarations

example.cpp

```
#include "example.h"

using namespace std;

int main(){
    cout << "Hello World!" << endl;
    return 0;
}
```

Hello World over two files

example.h

```
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

#include <iostream>

int main();

#endif
```

The `main` function has its signature declared in the header file, and its definition is contained in the implementation file

example.cpp

```
#include "example.h"

using namespace std;

int main(){
    cout << "Hello World!" << endl;
    return 0;
}
```

Hello World over two files

example.h

```
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

#include <iostream>

int main();

#endif
```

example.cpp

```
#include “example.h”

using namespace std;

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

Compiling a program split over two files is simple: the implementation file containing the definition of the `main()` function should be passed to the compiler, which will look for the required headers on its configured include path. However, the `main()` function implementation is usually not split across two files and is often the only function present in its file.

For a Linux OS:

```
g++ -std=c++11 example.cpp -o example_exec
```

6. Manipulating Strings

Strings

Strings are objects representing a sequence of characters, supporting multi-byte characters and variable-length sequences regardless of the encoding used.

```
#include <string>
...
std::string message = "What is your name?";
```

Strings act more than just a simple storage for character sequences, providing specialized functions too. For example:

- `message.copy()` → Copies contents of `message` elsewhere
- `message.find()` → Searches contents of `message` for the first occurrence of a string or character
- `message.substr()` → Creates a new string with a portion of the contents of `message`
- `message.compare()` → Compares `message` to another string lexicographically

Strings and standard I/O

Output

Printing a string is straightforward: simply use the `std::cout` stream and pass the string via the insertion operator.

Input

When using `std::cin` only the first token is extracted: whitespaces, tab characters, new-line characters, etc. all terminate the value being extracted, acting as delimiters.
In case of needing an input string containing such characters, the `getline()` function can be used. This function parses an input stream and stores its values in a string object, specifying the delimiter wanted (default: `'\n'`).

```
...
std::string msg = "";
std::cin >> msg;
std::cout << msg << std::endl;
...
```

Input: I'm testing strings in C++
Output: I'm

```
...
str::string msg = "";
std::getline(std::cin, msg);
std::cout << msg << std::endl;
...
```

Input: I'm testing strings in C++
Output: I'm testing strings in C++

Strings and standard I/O

A special stream explicitly operating on string objects is implemented in the `sstream` header, in the `stringstream` class.
It can be used to convert strings back and forth to other types, such as `int`s, `float`s, etc.

From string to int

```
...
int num = 0;
std::string msg = “”;
std::cin >> msg;
std::stringstream ss;
ss << msg;
ss >> num;
std::cout << num << std::endl;
```

From int to string

```
...
str::string msg = “”;
int num = 0;
std::cin >> num;
std::stringstream ss;
ss << num;
ss >> msg;
std::cout << msg << std::endl;
```

This is not the only way to convert values between these two types, instead it is a display of the flexibility of the stream paradigm in C++.

7. Exercises

Exercises

1. Write a program that prompts the user to insert an integer and a floating-point number, then performs and outputs their sum.
2. Write a program that reads a string containing a number, then converts it into an integer, adds 5 to such value, and prints the result as a string.

References

- https://en.wikipedia.org/wiki/C%2B%2B
- https://www.tiobe.com/tiobe-index/
- https://www.baeldung.com/cs/compiled-vs-interpreted-languages
- https://en.cppreference.com/w/cpp/keyword
- https://www.cplusplus.com/doc/tutorial/variables/
- https://www.programiz.com/cpp-programming/variables-literals
- https://en.cppreference.com/w/cpp/language/namespace
- https://www.learncpp.com/cpp-tutorial/class-code-and-header-files/
- https://docs.microsoft.com/en-us/cpp/cpp/header-files-cpp
- https://www.cplusplus.com/doc/tutorial/preprocessor/
- https://www.cplusplus.com/reference/string/string/
- https://www.cplusplus.com/reference/string/string/getline/
- https://www.cplusplus.com/reference/sstream/