Hello C++

Programação 2024/2025 (L.EIC009)

José Proença (FCUP) & João Bispo (FEUP) – *slides by Eduardo R. B. Marques, FCUP*

Hello C++!

A first look at (C and) C++:

- Background: a bit of history, key aspects of C/C++.
- "Hello world!" - our first C++ program compiled using GCC.

Background

History of C and C++

U.PORTO

- **1972/73**: first version of C by Dennis Ritchie, used widely in UNIX operating systems.
- **1978**: 1st edition of "The C programming language", Brian Kernighan e Dennis Ritchie.
- **Late 1970s/early 1980s**: C with Objects by Bjarne Stroustrup, later renamed to C++.
- **1985**: 1st edition of "The C++ programming language" by Bjarne Stroustrup.
- **1989/90**: C89 and C90 ANSI standards. C90 also an ISO standard.
- **1998**: C++ 98 ISO standard.

Read more: → History of C → History of C++

Old but still very much around ...

U.PORTO

TIOBE popularity index for Oct 2024

Oct 2024	Oct 2023	Change	Programming Language	Ratings	Change
1	1		Python	21.90%	+7.08%
2	3	↑	C++	11.60%	+0.93%
3	4	↑	Java	10.51%	+1.59%
4	2	↓	C	8.38%	-3.70%
5	5		C#	5.62%	-2.09%
6	6		JavaScript	3.54%	+0.64%
7	7		Visual Basic	2.35%	+0.22%
8	11	↑	Go	2.02%	+0.65%

C is an imperative programming language - programs are defined by functions containing imperative instructions that execute in sequence.

C++ is an object-oriented language - it essentially extends C with class-based object-oriented programming . C is not a strict subset of C++, although almost all of C code can be embedded in C++ without change.

There is also **Objective-C** with a different proposal for object-oriented programming. C is a strict subset of Objective-C. While the use of C and C++ is widespread, Objective-C has been historically associated to program development for Apple operating systems and related programming frameworks.

C++ and Objective-C but also other languages like **C#** and **Java** are said to be of the **“C family”** due to the similarity of of syntactic constructs for imperative programming. C# and Java are more “distant relatives” to C than C++ and Objective-C however.

C and C++ programs:

- are compiled to binary machine code (but also, more recently, to WebAssembly);
- are statically typed, meaning that declarations (for variables, functions, etc) must have a declared (or unambiguously inferred) type that is checked at compile time;
- directly access memory and must explicitly manage dynamically allocated memory;
- may have *undefined behavior*, in particular regarding memory access - memory access bugs are easy to introduce and their impact is unpredictable. → Undefined behavior - Wikipedia entry

C/C++ vs Python

C and C++ contrast with Python for instance, the language you learned during the FP course. Python programs:

- are compiled (on-the-fly) to an abstract bytecode which is then interpreted by CPython (this happens with many other languages too, e.g. C# or Java);
- need not be typed - type-related errors can occur at runtime;
- cannot access memory directly, on the other hand the programmer need not be concerned about memory access errors, moreover they rely on automatic memory management (garbage collection mechanism is embedded in the execution environment);
- have precise semantics;

Use of C and C++

U.PORTO

There are C/C++ compilers for all kinds of systems and architectures. Mature compilers (GCC, Clang, ...) can generate highly optimised machine code.

C and C++ are used for instance in the implementation of:

- virtual machines for other languages (e.g. CPython, Java Virtual Machine)
- operating systems (eg. Linux, MacOS, Windows)
- database engines (eg. MySQL, SQLite)
- embedded systems (eg. Arduino, ROS)
- parallel computing (eg. MPI, OpenMP, CUDA)
- all kinds of software applications ...

“Hello world!”

Let us edit a file named `hello.cpp` with the following contents:

```cpp
/*
  A simple program that prints "Hello world!"
*/
#include <iostream>

int main() {
  // Print the message
  std::cout << "Hello world!\n";
  return 0;
}
```

Compiling the program

The program can be compiled using GCC as follows:

```
$ g++ -Wall -Werror -std=c++17  -g hello.cpp -o hello
```

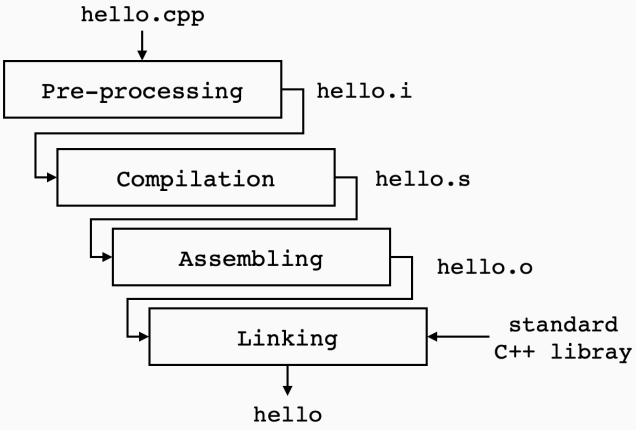
Above, we use a few compiler options that are relevant:

- `-Wall`: emit all possible warnings;
- `-Werror`: treat warnings as errors;
- `-std=c++17`: force compliance with the C++ 2017 standard (the one we will use);
- `-g`: emit code with debug information (for use with a debugger like GDB);

The generated binary file (`hello`) can then be executed:

```
$ ./hello
Hello world!
```

C++ compilation



We can get the intermediate (temporary) files using `-save-temps` with GCC.

```
$ g++ ... other options \
  ... -save-temps hello.c \
  -o hello
```

C++ compilation (cont.)

U.PORTO

1. Pre-processing

Handles pre-processing directives like `#include`. Generates C++ code without pre-processing directives (`hello.i` previously).

2. Compilation

Parses and commpiles pre-processed code onto assembly code.

3. Assembling

Translates assembly code to (binary) machine code, sometimes called "object files" (do not confuse with object-oriented concepts).

4. Linking

Links the program machine code together with the necessay libraries, yielding an executable program.

“Hello world!” dissected – Some key aspects of syntax

U.PORTO

Comments

multi-line comments started with `/*` and ended with `*/` or single-line comments started with `//`.

Pre-processing directives

started with #, e.g. `#include`.

Keywords

words with special meaning, e.g. `return` and `int`.

Expressions and operators

as in `cout << "Hello world\n"`

“White” characters

(e.g. line breaks or spaces) only separate tokens. Indentation is not semantically significant (unlike in Python) but helps reading code.

Function definitions

like `main` in the example.

Separator/grouping characters

`;`, `(`, `)`, `{`, `}`, ...

Namespaces

definitions may be grouped in namespaces.

Comments

Multi-line comments start with `/*` and end with `*/`

```
/*
  A simple program that prints "Hello world!"
*/
```

Single-line comments start with `//`

```
// Print the message
std::cout << "Hello world!\n";  // Also a comment
```

Comments (cont.)

Examples of common errors:

```
/*
  Comment section
std::cout << "Hello world!\n";
```

Comments (cont.)

Examples of common errors:

```
/*
  Comment section
std::cout << "Hello world!\n";
```

(non-terminated comment)

```
/*
  Comment section
*/ */
std::cout << "Hello world!\n";
```

Comments (cont.)

Examples of common errors:

```
/*
  Comment section
std::cout << "Hello world!\n";
```

(non-terminated comment)

```
/*
  Comment section
*/ */
std::cout << "Hello world!\n";
```

*(*/ without preceding /*)*

```
std::cout << "Hello world!\n"  // Comment
                                section
```

Comments (cont.)

Examples of common errors:

```
/*
  Comment section
std::cout << "Hello world!\n";
```

(non-terminated comment)

```
/*
  Comment section
*/ */
std::cout << "Hello world!\n";
```

*(*/ without preceding /*)*

```
std::cout << "Hello world!\n"  // Comment
                                section
```

(comment spans over more than one line)

```
/*                        A simple program that
prints        "Hello world!"*/ #include <iostream>

             int main(
){// Print the message
         std::cout << "Hello world!\n";return 0;}
```

Indentation does not alter the meaning of programs, like in most programming languages (not Python though!).

Indentation makes programs readable however! The above variant of `hello.cpp` is valid but obviously very hard to understand ...

Pre-processing directives are prefixed with #, as in

```
#include <iostream>
```

The `#include` directive includes the contents of another C++ file, `iostream` in the example. This is done for files that contain definitions that must be imported for program compilation, called **header files**.

Header files may have no extension - this is usually the case header files in the C++ library like `iostream` - but `.h` and `.hpp` extensions are also common for C/C++ header files.

Other pre-processing directives can be used (we will use some later in the course):

`#define #undef #if #ifdef #ifndef . . .` → further reference

Keywords

The source code of `hello.cpp` contains `int` and `return`. These are **keywords**. They have special meaning and cannot be used as names to identify functions, variables, types, etc.

Here are some other keywords that may be familiar from other programming languages:

```
if else switch case
for while do
int char float double
class private public protected
try catch
namespace using
...
```

→ complete list of keywords

Namespaces

"Somewhere" in `iostream` (→ here) we have:

```
namespace std {
  . . .
  ostream cout;
  . . .
}
```

C++ definitions can be defined within namespaces, which are useful to prevent name conflicts (especially in large software projects, or when a program uses several external libraries).

A symbol `x` in namespace `n` can be referred to as `n::x` or simply as `x` if the source code contains a directive of the form:

```
using namespace n;
```

(this is similar in spirit to `from n import *` in Python or `import n.*;` in Java for instance)

```
namespace std { . . .  ostream cout; . . .  }
```

`std` is the namespace used by the standard C++ library (of which the `iostream` header is part of).

In `hello.cpp` we have

```
std::cout << "Hello world!\n";
```

If we include at the beginning the `using` directive

```
using namespace std;
```

Then we can use `cout` without the namespace `std::cout`

```
cout << "Hello world!\n";
```

A **function** is defined by a declaration and a body.

```
int main() // <--- Function declaration
{ // --> Function body
  std::cout << "Hello world\n";
  return 0;
  // <--
}
```

The function declaration (also designated by prototype or signature), indicates the function's return type and arguments. A function's body, is the sequence of instructions to execute when the function is called.

Special case of `main`: `main` is always the entry point for the execution of a C++ program, i.e., `main` is invoked when the program starts.

```
int main() // <--- Function declaration
{
  ...
}
```

The declaration of `main` indicates a return type of `int` and no arguments. More generally a function declaration has the form:

```
return_type function_name(type_1 arg_1 ,
                          type_2 arg_2 ...,
                          type_n arg_n)
```

The special case of `main`: `main` can be defined alternatively as

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

if we need to process program arguments (e.g., supplied via the command line).

→ more info here

Function body

```
int main() // <--- Function declaration
{ // --> Function body
  cout << "Hello world\n";
  return 0;
  // <--
}
```

The body of a function is an instruction block grouped between { and }. Instruction containing simple instructions delimited by ; that execute in sequence.

In the example, the body of `main` contains two simple instructions: `cout << "Hello world!\n";` and `return 0;`.

A `return` instruction indicates the value to return.

The special case of `main`: the value returned by `main` is program exit code returned to the operating system. By convention, a value of `0` denotes absence of errors.

Use of `std::cout`

In C++ we can use **objects** that are instances of **classes**. During the semester we will come to understand exactly what this means. In any case the use of `std::cout` is a first example of the use of objects...

```
// In iostream header
namespace std { ...  ostream cout; ...  }
...
// Program code
#include <iostream>
...
std::cout << "Hello world\n";
```

`std::cout` is an **object** declared globally by the `iostream` header that represents the standard output stream of a program. It is an instance of **class type** `std::ostream`.

Use of `std::cout` (cont.)

What about << ?

```
std::cout << "Hello world!\n";
```

The use of << , known in the context of streams as the **stream insertion operator**, corresponds to the invocation of function that defines the appropriate behavior of the << operator.

This mechanism is known as **operator overloading**. We will see later how this works precisely.

For instance, << is defined for `ostream` in conjunction with different data types:

```
...
ostream& operator<< (int val);
ostream& operator<< (float val);
ostream& operator<< (double val);
...
```

Input using `std::cin`

In symmetry to `std::cout`, `std::cin` is a global object that can be used to read the standard input stream of a program.

The `>>` **stream extraction operator** can be used to read data:

```
int n;
std::cout << "n ? "; std::cin >> n;
```

Like `cout` and `<<`, we can chain several uses of `>>` with `cin`, e.g.

```
int a, b;
std::cin >> a >> b;
```

In the above fragment, we can input two integers separated by one or more "white" characters (space, tabs, newline, ...). **An important detail is that variables `a` and `b` are declared with the type `int` before their use.** We will discuss variables later.

Note on string constants

In

```
std::cout << "Hello world!\n";
```

`"Hello world!\n"` is a string constant.

The sequence of characters include `\n`, that stands for the line break character.

Note that `'Hello world\n'` is not a valid string constant as in Python. `'` is used to define (single) character constants, e.g., `'H'` denotes the value for letter H.