01_Hello_World

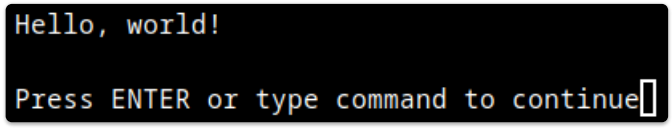
Table of contents

Hello, world!

Con la speranza di avere impostato correttamente il nostro ambiente di programmazione, usando per esempio Visual Studio Code, proviamo a creare un file a nome `helloworld.ccp` contenente le seguenti linee.

```
// g++ -Wall -Wextra helloworld.cpp -o helloworld
#include <iostream>

int main() {
        std::cout << "Hello, world!" << std::endl;
}
```



Conosciamo il codice?

Poniamoci domande (Q) banali a primo avviso, a cui proviamo a dar risposta (R):

- Q: come mai scrivere parentesi `()` nel `main`? E se mettessimo come "assegnatura" una stringa o un intero?
  R: Quì il linguaggio è stato deciso come essere il più compatibile possibile, seguendo le tecniche ormai passate nel tempo.
- Q: perché il `main` restituisce `int` se non viene restituito nulla a fine funzione?
  R: Il `main` permette di darci la libertà di evitare il *tipo di ritorno*, siccome il compilatore lo inserirà a modo automaticamente (`return 0` per un successo).
- Q: l'operatore `<<` serve per stampare output al terminale, come mai non segnala errore?
  R: perché già incluso nei tipi, siccome automaticamente viene preso in carico avendo passato come parametro un `cout` (come se stessimo aprendo una matriosca)
- Q: come mai includiamo `iostream`?
  R: `iostream` non è una libreria, è un *header file* che ci dice cosa possiamo trovare nella libreria, dove possiamo trovare le *dichiarazioni* associategli (definizione $\neq$ dichiarazione).

    - `cout` non è parte del linguaggio C++, ma piuttosto della libreria (per questo non ha syntax highlighting);

    - la libreria non fa parte del linguaggio, siccome questa raggruppa codice utente;

- usiamo `< >` per dire al *precompilatore*, di usare il file contenuto nelle cartelle base in cui vengono installate le librerie;

- al campo di visibilità *scope*, se eliminassimo la dicitura `std`, il compilatore darebbe errore in quanto il *namespace* non sarebbe specificato.

```
helloworld.cpp: In function 'int main()':
helloworld.cpp:5:9: error: 'cout' was not declared in this scope; did you mean 'std::cout'?
    5 |         cout << "Hello, world!" << std::endl;
      |         ^~~~
      |         std::cout
In file included from helloworld.cpp:2:
/usr/include/c++/12.2.1/iostream:61:18: note: 'std::cout' declared here
   61 |   extern ostream cout;          /// Linked to standard output
      |                  ^~~~

shell returned 1
```

Conosciamo la compilazione?

Cosa succede quando noi diamo il comando di compilazione?

```
g++ -Wall -Wextra helloworld.cpp -o helloworld
 ^    ^                 ^               ^
 |    |                 |               |
 |    opzioni         sorgente        oggetto
compilatore
```

Quando parliamo di *compilazione*, parliamo del senso in ampiezza:

1. **preprocessing**, guarda i file da compilare come semplici sequenze di caratteri, quindi pochi controlli sono effettuati; prende il file sorgente `helloworld.cpp` e produce un altro file sorgente a nome *unità di traduzione*, la cosa effettiva passata al compilatore per la compilazione. La verità è che sono tanti i sorgenti acquisiti.
2. **processing**, eseguito dal compilatore, che produce codice *assembler* per la nostra macchina (l'oggetto compilato non è "portabile").
3. **assembling**, il codice *assembler* viene preso e generato il *codice oggetto*.
4. **linking**, producente codice *eseguibile* o libreria.

Preprocessing

Precisare a `g++` l'opzione `-E` ci permette di vedere il file di preprocessing che verrà dato in pasto al compilatore, che se troverà errore, mi dovrà dire dove si trova.

```
-E  Stop after the preprocessing stage; do not run the compiler proper.  The
    output is in the form of preprocessed source code, which is sent to the
    standard output.

    Input files that don't require preprocessing are ignored.
```

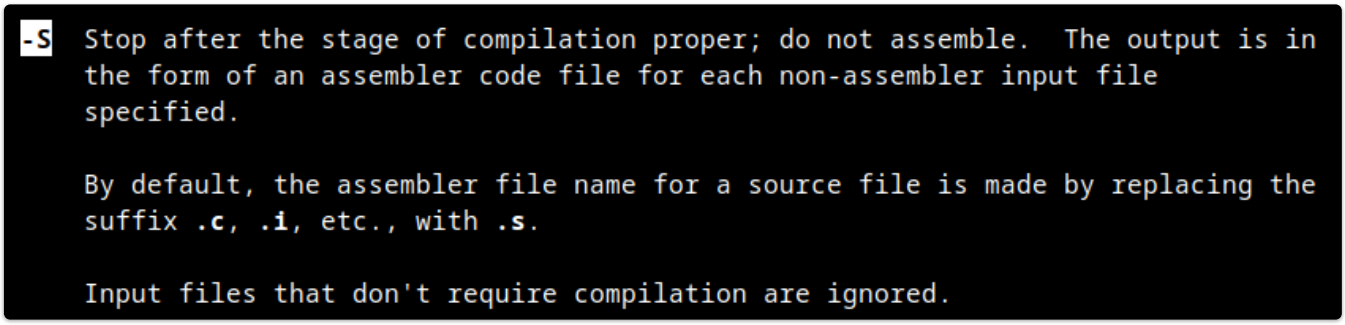
```
g++ -E helloworld.cpp -o helloworld.preproc.cpp
```

Il compilatore, compilerà tutte le linee di codice incluse in questo file (che solitamente sono migliaia).

```
File: hello.preproc.cpp
  1  # 0 "hello.cpp"
  2  # 0 "<built-in>"
  3  # 0 "<command-line>"
  4  # 1 "/usr/include/stdc-predef.h" 1 3 4
  5  # 0 "<command-line>" 2
  6  # 1 "hello.cpp"
  7  # 1 "/usr/include/c++/12.1.1/iostream" 1 3
  8  # 36 "/usr/include/c++/12.1.1/iostream" 3
  9
 10  # 37 "/usr/include/c++/12.1.1/iostream" 3
 11
 12  # 1 "/usr/include/c++/12.1.1/x86_64-pc-linux-gnu/bits/c++config.h" 1 3
 13  # 296 "/usr/include/c++/12.1.1/x86_64-pc-linux-gnu/bits/c++config.h" 3
 14
 15  # 296 "/usr/include/c++/12.1.1/x86_64-pc-linux-gnu/bits/c++config.h" 3
 16  namespace std
:
```

Assembling

Dando l'opzione `-S`, diciamo al compilatore di fermarsi al codice assembler.
Non esistono gli overloading di operatori, non esistono namespace, siccome vengono tutti codificati sul posto.



```
-S  Stop after the stage of compilation proper; do not assemble.  The output is in
    the form of an assembler code file for each non-assembler input file
    specified.

    By default, the assembler file name for a source file is made by replacing the
    suffix .c, .i, etc., with .s.

    Input files that don't require compilation are ignored.
```

```
g++ -S helloworld.cpp -o helloworld.assembler.cpp
```

Il file in assembly è decisamente più piccolo siccome c'è soltanto il necessario per l'esecuzione.

```
File: helloworld.assembler.cpp
  1      .file   "helloworld.cpp"
  2      .text
  3      .local  _ZStL8__ioinit
  4      .comm   _ZStL8__ioinit,1,1
  5      .section    .rodata
  6  .LC0:
  7      .string "Hello, world!"
  8      .text
  9      .globl  main
 10      .type   main, @function
 11  main:
 12  .LFB1761:
 13      .cfi_startproc
 14      pushq   %rbp
 15      .cfi_def_cfa_offset 16
 16      .cfi_offset 6, -16
:
```

Compilation

Dando l'opzione `-c`, che dovrebbe essere quella più comune, compilo fino al file binario che poi verrà passato al *linker*.

```
-c  Compile or assemble the source files, but do not link.  The linking stage
    simply is not done.  The ultimate output is in the form of an object file for
    each source file.

    By default, the object file name for a source file is made by replacing the
    suffix .c, .i, .s, etc., with .o.

    Unrecognized input files, not requiring compilation or assembly, are ignored.
```

```
g++ -c helloworld.cpp -o helloworld.o
```

Se andiamo a vedere, è tutto in binario.
Possiamo tradurre i simboli con il comando di *name mangling*:

```
nm -C helloworld.o
```

```
maruko@markarch ~/Documents/uni/2_anno/metodologie/hello nm -C hello.o
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
0000000000000088 t _GLOBAL__sub_I_main
0000000000000000 T main
0000000000000036 t __static_initialization_and_destruction_0(int, int)
                 U std::ostream::operator<<(std::ostream& (*)(std::ostream&))
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
                 U std::cout
                 U std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::c
har_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
0000000000000000 b std::__ioinit
                 U std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::
char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
```

BIN to HEX usando `hexdump`

In terminale, possiamo tradurre il codice binario in uno esadecimale usando il comando:

```
hexdump -C helloworld.o
```

Notiamo come la stringa "Hello, world!" sia presente, con la sola differenza che è ora in linguaggio macchina, decisamente difficile da leggere se non siamo delle macchine.

```
  14    000000d0  00 bf 01 00 00 00 e8 9b  ff ff ff 5d c3 48 65 6c  |...........].Hel|
  15    000000e0  6c 6f 2c 20 77 6f 72 6c  64 21 00 00 00 00 00 00  |lo, world!......|
```

Linking

Ultima fase, che possiamo visualizzare con il comando originale visto all'inizio.

```
g++ -Wall -Wextra helloworld.cpp -o helloworld
```

Oppure usando il file oggetto:

```
g++ -Wall -Wextra helloworld.o -o helloworld
```

Molto piccolo: non c'è dentro tutto perché di default viene usato il *collegamento dinamico*. Possiamo elencare le dipendenze (da soddisfare a tempo di esecuzione) con il comando:

```
ldd helloworld
```

```
maruko@markarch ~/Documents/uni/2_anno/metodologie/hello ldd helloworld
        linux-vdso.so.1 (0x00007fffcc7d6000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f72bfc00000)
        libm.so.6 => /usr/lib/libm.so.6 (0x00007f72bfe38000)
        libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007f72bfbe0000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f72bf9f9000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f72bff55000)
```

---

metodologie > CODE > HelloWorld > HelloWorld.cpp

21/02/2023