Hello World

A cosa serve questo programma?

Lo scopo di questo programma, in realtà, è quello di effettuare un test sulla corretta installazione dell'ambiente di sviluppo scelto per il C++. Ovvero, ci si vuole assicurare di avere un compilatore funzionante, le librerie di sistema correttamente installate, ecc.

Dal nostro punto di vista, comunque, questo semplice programma si presta bene ad evidenziare la differenza che esiste tra una conoscenza superficiale del linguaggio C++ ed una conoscenza un po' più approfondita.

Esempi di domande da esame:

- perché la funzione `main` è dichiarata per restituire un valore intero?
- perché non esiste la corrispondente istruzione di `return`?
- perché devo qualificare il nome (`cout`) del canale di output con `std`?
- che cosa indica `std`?
- qual è la differenza tra queste due varianti? `std::cout << "Hello, world!" << std::endl;` vs `std::cout << "Hello, world!\n";`
- che cosa è `iostream`?
- perché devo includere `iostream`?
- perché devo usare le parentesi angolate (e non le virgolette) quando includo `iostream`?
- a cosa si riferiscono le due occorrenze dell'operatore infisso `<<` ?
- sono invocazioni di operatori built-in o si tratta di funzioni definite dall'utente?

Esempio di helloworld:

```
#include

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

Torna all'indice

Processo di compilazione

Il compilatore `g++` è un wrapper per il compilatore `gcc` (*Gnu Compiler Collection*), il quale è in realta una collezione di compilatori per diversi linguaggi.

In senso lato, il processo di compilazione prende in input file sorgente e/o librerie e produce in output file eseguibili o librerie. Esso segue alcuni passaggi fondamentali:

1. Il preprocessore elabora il codice del file sorgente per produrre una *unità di traduzione*.
2. Il compilatore (in senso stretto) elabora l'unità e produce codice *assembler*.
3. L'assemblatore produce da questo il file oggetto.
4. Il linker si occupa infine di realizzare i collegamenti tra i vari file oggetto e le librerie al fine di ottenere l'eseguibile (o una libreria).

Comandi step-by-step

1. `g++ -E hello.cpp -o hello.preproc.cpp` L'opzione `-E` indica al compilatore di fermarsi subito dopo la fase di preprocessing. Le prime 10 righe dell'output sono le seguenti:

```
# 0 "hello.cpp"
# 0 ""
# 0 ""
```

```
 # 1 "/usr/include/stdc-predef.h" 1 3 4
 # 0 "" 2
 # 1 "hello.cpp"
 # 1 "/usr/include/c++/12.2.1/iostream" 1 3
 # 36 "/usr/include/c++/12.2.1/iostream" 3

 # 37 "/usr/include/c++/12.2.1/iostream" 3
```

2. `g++ -Wall -Wextra -S hello.cc -o hello.s` L'opzione `-S` permette di fermarsi alla fase di compilazione in senso stretto, con la generazione del codice assembler.

```
  .file "hello.cpp"
  .text
  .local _ZStL8__ioinit
  .comm _ZStL8__ioinit,1,1
  .section .rodata
.LC0:
  .string "Hello, world!"
  .text
  .globl main
  .type main, @function
main:
.LFB1761:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq %rsp, %rbp
  .cfi_def_cfa_register 6
  leaq .LC0(%rip), %rax
  movq %rax, %rsi
  leaq _ZSt4cout(%rip), %rax
  movq %rax, %rdi
```

3. `g++ -Wall -Wextra -c hello.cpp -o hello.o` L'opzione `-c` produce il codice oggetto (scritto in linguaggio macchina) fermandosi prima del collegamento.

4. `g++ -Wall -Wextra hello.cc -o hello` Quando NON si specifica nessuna delle opzioni `-E`, `-S`, `-c`, il compilatore termina dopo avere effettuato il collegamento, utilizzando il linker (ld), producendo il file eseguibile `hello` (senza suffisso, come consuetudine per l'ambiente Linux). In questo caso, avendo un solo file oggetto, il collegamento avviene tra questo file e i file che formano la libreria standard del C++ (che vengono coinvolti implicitamente, senza doverli specificare come argomenti per il compilatore). In altri casi, si può indicare il percorso degli headers da includere con l'opzione `-I`. Esempio:

Torna all'indice

Considerazioni sul contenuto dei files prodotti

Anzitutto le dimensioni sono molto variabili:

```
$ du -b hello*
16008 hello
81      hello.cpp
2736 hello.o
820162 hello.preproc.cpp
2032 hello.s
```

In particolare l'unità di traduzione risulta molto estesa: essa contiene l'espansione dell'unica direttiva di inclusione del sorgente (`#include <iostream>`). Il preprocessore include dunque il codice di numerosi altri file sorgente che in questo caso fanno parte della libreria standard del C++. I nomi di questi file (e la loro posizione nel filesystem) si possono ottenere osservando le direttive del preprocessore rimaste nell'unità di traduzione (le linee che iniziano con il carattere `#`):

```
...
# 1 "/usr/include/c++/9/iostream" 1 3
...
# 1 "/usr/include/x86_64-linux-gnu/c++/9/bits/basic_ios.h" 1 3
brav# 1 "/usr/include/features.h" 1 3 4
...
```

Queste direttive servono al compilatore per generare messaggi di errore che facciano riferimento ai nomi dei file e ai numeri di riga dei file sorgenti (e non al numero di riga dell'unità di traduzione, cosa che sarebbe alquanto scomoda per il programmatore).

Si deve notare che gli header files inclusi sono necessari in quanto contengono la dichiarazione della variabile `std::cout`(e del corrispondente tipo), dell'operatore di output `operator<<` e del modificatore `std::endl`.

```
namespace std {
 // ...
  typedef basic_ostream ostream;
 // ...
  extern ostream cout;
 // ...
  extern template ostream& endl(ostream&);
 // ...
  extern template ostream& operator<<(ostream&, const char*);
 // ...
}
```

Osserviamo che:

- `std::ostream` è un alias per la classe `basic_ostream<char>` ottenuta instanziando il template di classe `basic_ostream` con il tipo `char`.
- Il modificatore `std::endl` è una funzione templatica specializzata con il tipo `ostream`.
- La prima occorrenza di `operator<<` è una chiamata a una funzione templatica, specializzata su `ostream` e `const char*`.
- La seconda occorrenza di `operator<<` fa riferimento invece ad un altro tipo di funzione che prende come argomento due parametri di tipo `ostream`.

  Perchè non è necessario qualificare l'operatore << all'interno del proprio *namespace* (`std::<<`)? Il C++ utilizza l'**Argument-Dependent Lookup** (ADL): se viene utilizzato un argomento definito dall'utente come argomento di una funzione (o operatore) e non viene esplicitato il namespace di quest'ultima, si cerca nell'ordine

  1. nello scope del chiamante, e
  2. in tutti i namespace a cui appartengono gli argomenti, partendo dal primo a sinistra.

  Ad esempio `operator<<(ostream&, const char*)` prende in input un tipo `ostream&` dichiarato all'interno del namespace `std`; in esso si trova anche la definizione dell'operatore da utilizzare.

Esempio: ADL

```
#include
#include

namespace A {
 class Message : public std::string {
 public:
  Message() : std::string("This is a message."){};
 };

 std::ostream& operator<<(std::ostream& os1, std::string& msg) {
  std::cout << "Operator << in namespace A" << std::endl;
  return os1;
 }
} // ! NAMESPACE_A

int main() {
 A::Message msg;
 std::string str("Hello World!");
 std::cout << str << std::endl; // std::operator<<(std::ostream&, std::string&)
 std::cout << msg; // A::operator<<(std::ostream&, std::string&)
}
```

Torna all'indice

Ricapitolando

Giunti al termine di questa discussione su `"Hello, world!"` è forse il caso di ricapitolare alcuni concetti chiave.

1. Tecnicamente, quando si dice che un programma è formato da un solo file sorgente, si commette un errore; come abbiamo visto, anche un programma banale come `"Hello, world!"` necessita di oltre un centinaio di header file di sistema (il numero esatto dipende dall'implementazione specifica usata), oltre a `helloworld.cpp`.

2. I programmi "veri" sono formati da numerosi file sorgente, anche quando si escludono gli header file di sistema; è necessario comprendere i meccanismi che consentono di separare il programma in diversi file sorgente, per poi compilarli separatamente e infine collegarli in modo corretto; occorre inoltre imparare come compilare e collegare correttamente un programma che dipenda da librerie software di terze parti (ovvero, diverse dalla libreria standard del C++).

3. In generale, ogni volta che si scrive qualche linea di codice è necessario chiedersi qual è la sua funzione, evitando di dare la risposta: "Non ne ho idea, so solo che si è sempre fatto così". Questa è la risposta di un programmatore che NON sa risolvere eventuali problemi tecnici che si dovessero presentare e deve per forza chiedere consiglio a programmatori più esperti. La nostra ambizione dovrebbe essere quella di (iniziare il lungo cammino per) diventare noi stessi programmatori esperti e sapere come affrontare i problemi tecnici legati alla programmazione. Si noti che queste competenze sono ben separate ed indipendenti rispetto alla conoscenza dello specifico dominio applicativo per il quale si è deciso di sviluppare uno strumento software.

Torna all'indice