

# Imparare a programmare con Haiku

## Lezione 2

Scritto da DarkWyrn

Traduzione in italiano, per Haiku Italia, a cura di

Giuseppe Gargaro & Francesca Mora

Il nostro primo paio di programmi non era granché utile, ma man mano che procediamo, scoprirete che si può fare sempre di più con i vostri programmi. Questa volta vedremo due concetti principali: i commenti e le differenti fasi attraversate dal compilatore per generare il vostro programma, ed impareremo anche qualcosa sul debugging del codice.

I commenti sono note che inserite nel codice. Hanno molti usi, come chiarire una certa porzione di codice, fornire avvertimenti, o disabilitare temporaneamente una sezione del codice. Vedere l'esempio sotto per come possono essere utilizzati.

```
// Questo è un commento di una sola riga.  
// Anche questo lo è. Tutto ciò che è dopo i due slash è considerato parte di esso.
```

```
int main(void)  
{  
    PushTheRedButton();           // Questo codice non funziona.  
    return 1;  
}
```

Eredità dal C sono i commenti multi-linea. Iniziano con `/*` e finiscono con `*/` in un modo simile alle parentesi o alle parentesi graffe, ma con una differenza: non si può mettere un commento multi-linea all'interno di un altro.

```
/*-----  
RedButton.cpp  
Questo codice è un esempio di come il compilatore si lamenta se si utilizza  
una funzione che non riconosce.  
-----*/
```

```
// Questo è un commento di una sola riga.  
// Anche questo lo è. Tutto ciò che è dopo i due slash è considerato parte di esso
```

```
int main(void)  
{  
    PushTheRedButton();           // Questo codice non funziona.  
    return 1;  
}
```

Ora facciamo un passo indietro e guardiamo il processo che il compilatore esegue per compilare il vostro programma. E' importante capirlo perché ci sono diversi tipi di errori che possono essere fatti quando state scrivendo il codice e conoscere qualcosa del processo vi aiuterà a trovarli più facilmente.

Quando un programma è costruito a partire dal codice sorgente, ci sono quattro strumenti che agiscono su di esso prima che sia un eseguibile: il preprocessore, il compilatore, l'assemblatore e il linker.

### ***Fase 1: Il preprocessore***

Il preprocessore accetta codice sorgente come in ingresso e gli fa un piccolo massaggio prima che il codice venga inviato al compilatore. Rimuove i commenti e maneggia le direttive `#include` inserendo il contenuto degli header inclusi nel codice. Gestisce anche altre direttive che verranno discusse in seguito.

## **Fase 2: Il compilatore**

Il compilatore traduce il codice C++ in linguaggio Assembly. Assembly è molto, molto vicino alle istruzioni che il computer capisce, pur rimanendo leggibili. E' anche più difficile scrivere programmi ed è specifico per il processore per cui è scritto.

## **Fase 3: L'assemblatore**

L'assemblatore crea il codice oggetto del codice Assembly creato dal compilatore e lo pone nei file oggetto che hanno l'estensione .o . I codici oggetto sono effettivamente le istruzioni eseguibili dalla macchina e vengono usati dal computer per eseguire il programma. Però, non è ancora pronto per essere avviato. In questa fase i file oggetto utilizzati per realizzare il vostro programma sono un po' come l'insieme dei pezzi di un puzzle pronti per essere messi insieme.

## **Fase 4: Il linker**

Il linker mette insieme in un programma eseguibile i file oggetto con tutte le librerie che utilizzano.

## **Debugging**

I programmatori, essendo umani, fanno errori, e molti. La scrittura del codice e il debugging vanno di pari passo e spesso sono fatti nello stesso tempo. Ci sarà, quindi, da imparare come effettuare il debug dei programmi mentre impariamo a scriverli.

I bug sono di due tipi: sintattici e semantici. I bug sintattici sono facili da trovare perché è il compilatore che li trova per noi. Ci sono problemi come gli errori di capitalizzazione (errori nell'uso di parole con l'iniziale maiuscola), parentesi mancanti o supplementari, e errori di digitazione nei nomi delle funzioni. Gli errori semantici sono spesso difficili da trovare perché sono errori nella logica di codice perfettamente legale. Un errore semantico in Inglese sarebbe "The oxygen sensor on my car needed to be replaced" – la frase è perfettamente legale e correttamente costruita, ma la parola necessaria è sensor, non censor. Esempi di errori semantici sono 'i punti e virgola' in più in certi punti, aggiungere un ammontare sbagliato ad un numero, e fare ipotesi sul valore di ritorno di una funzione.

Diamo un'occhiata a qualche esempio semplificato di errori comuni

### **Esempio 1**

#### **Codice**

```
#include <stdio.h>
int main(void)
{
    return 1;
} }
```

#### **Errori**

foo.cpp:6: error: expected declaration before `}' token

Quello che abbiamo qui è una parentesi graffa in più. Il gcc ci ha dato un errore piuttosto criptico, come al solito, ma ci ha dato anche due indizi: il nome del file e il numero di riga. Il numero della riga data dal gcc è la posizione effettiva dell'errore non sono sempre gli stessi, ma in questo caso lo sono.

A questo punto, forse vi starete chiedendo: "Che cosa diavolo è un token, razza di genio?" Un token è un qualsiasi elemento di una lingua. Così come una qualsiasi lingua scritta ha parole e punteggiatura, lo stesso vale anche per le lingue del computer. Così come due virgole di fila sono un errore di punteggiatura, un'ulteriore parentesi graffa è un errore di punteggiatura in C++.

## Esempio 2

### Codice

```
/*-----  
RedButton.cpp  
/*Questo codice è un esempio di come il compilatore si lamenta se si utilizza  
una funzione che non riconosce.*/  
-----*/  
  
// Questo è un commento di una sola riga.  
// Anche questo lo è. Tutto ciò che è dopo i due slash è considerato parte di esso  
  
int main(void)  
{  
    PushTheRedButton();           // Questo codice non funziona.  
    return 1;  
}
```

### Errori

```
foo.cpp:7: error: expected unqualified-id before '--' token
```

Questo è un esempio di come il numero di riga per l'errore non sia il punto stesso dell'errore effettivo. L'avviso dell'errore riguarda i trattini alla fine del commento multi-linea in alto. E' causato, però dall'inserimento di un commento multi-linea all'interno di un altro. Il preprocessore rimuove tutti i commenti, così quello che vede il compilatore è questo:

```
-----*/  
int main(void)  
{  
    PushTheRedButton();  
    return 1;  
}
```

Il compilatore non sa cosa fare con la linea tratteggiata e si lamenta.

## Esempio 3

### Codice

```
int Main(void)
{
    return 1;
}
```

### Errori

```
/usr/lib/gcc/i486-linux-gnu/4.4.1/../../../../lib/crt1.o: In function `__start':
/build/buildd/eglibc-2.10.1/csu/../sysdeps/i386/elf/start.S:115: undefined
reference to `main'
/tmp/ccv39Cuo.o:(.eh_frame+0x12): undefined reference to `__gxx_personality_v0'
collect2: ld returned 1 exit status
```

Si tratta di un errore di natura diversa. Ricordate che `main()` deve essere definita in ogni programma? Noi non l'abbiamo fatto. Abbiamo definito `Main()`. Il programma, per il resto, è valido quindi compila correttamente, ma quando il linker ha tentato di mettere tutto insieme, non ha potuto trovare la funzione richiesta ed ha lanciato un hissyfit (contrazione di "Hysterical fit" crisi isterica).

Ogni volta che vedete un errore contenente 'undefined reference to', avete un errore del linker.

Risolvere gli errori `undefined reference` del linker generalmente non è difficile. Di solito significa una di queste due cose: avete dimenticato di link (collegare) una libreria che utilizzate, o un file di codice sorgente è stato accidentalmente dimenticato quando avete creato il vostro programma.