Programmazione I - Laboratorio

Introduzione alle lezioni in laboratorio

Gianluca Mezzetti[1] Paolo Milazzo[2]

1. Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/∼mezzetti
mezzetti@di.unipi.it

2. Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/∼milazzo
milazzo@di.unipi.it

Corso di Laurea in Informatica
A.A. 2012/2013

Compilare un programma C

La compilazione di un programma C si articola in tre fasi principali:

1. pre-processing
2. compilazione (vera e propria)
3. collegamento (linking) a librerie e moduli

Vediamo come funzionano queste fasi usando il compilatore `gcc`

Fase I: pre-processing

E' la fase in cui vengono gestite le direttive al compilatore:

- Espansione degli `#include`: il compilatore carica le dichiarazioni delle funzioni definite nelle librerie incluse al fine di controllare che tali funzioni siano usate correttamente nel programma
- Sostituzione delle costanti definite con `#define`: il compilatore cerca tutte le occorrenze di tali costanti nel programma e le sostituisce con i valori corrispondenti
- Gestione di altre direttive che non vedremo...

Esempio - esempio_comp.c

```
/*
  piccolo esempio per spiegare il preprocessing
*/
#include <stdio.h>

#define K 10

int main()
{
  int i=K;
  for (i=0; i<K; i++)
    printf("%d",i);
}
```

Eseguendo:

`gcc -E esempio_comp.c`

si ha come risultato l'ouput del preprocessore.

Fase II: compilazione (1)

E' la fase in cui il programma viene trasformato in codice binario (non ancora in un eseguibile).

E' composto di diverse sotto-fasi (non necessariamente sequenziali):

1. Controllo dei tipi
   - controllo che le variabili utilizzate siano state dichiarate
   - controllo che le operazioni nelle espressioni siano usate con argomenti di tipo appropriato
   - controllo che negli assegnamenti il tipo della variabile da assegnare e dell'espressione assegnata sia lo stesso (o compatibile)
   - controllo che le funzioni siano chiamate con parametri di numero e tipo opportuno
   - ....
2. Analisi e ottimizzazioni
   - Eliminazione del codice morto (non raggiungibile durante l'esecuzione)
   - Tail recursion
   - ....
3. Generazione del codice binario

Fase II: compilazione (2)

Eseguendo:
`gcc -c esempio_comp.c`
si ha come risultato l'ouput del preprocessore (detto "modulo oggetto" o "object file") che tipicamente ha estensione `.o`.

Fase III: collegamento (linking) a librerie e moduli

Si collegano insieme più moduli oggetto per creare un file eseguibile

- **file1.o file2.o ...**: diversi moduli oggetto dello stesso programma ottenuti tramite compilazione separata (vedremo tra poco)
- librerie standard e di sistema
- **libfile1.a libfile2.a ...**: diverse librerie esterne che forniscono funzioni utilizzate all'interno del programma

Il risultato di questa fase è un unico file eseguibile (ad es. `a.out`)

Eseguendo

```
gcc esempio_comp.o
```

Il modulo oggetto verrà trasformato in un file eseguibile

Moduli e compilazione separata (1)

Un programma C non deve necessariamente consistere di un unico file.

Programmi complessi sono solitamente divisi in moduli separati, ogniuno implementato in un proprio file.

Ogni modulo consisterà di un insieme di funzioni, variabili globali, costanti, ecc... che devono complessivamente non contenere duplicati.

In particolare, dovremo avere un modulo principale che contiene la funzione `main`

Moduli e compilazione separata (2)

Scopi di suddividere il codice sorgente C in più moduli

- organizzare meglio il codice
- dividersi il lavoro
- compilare separatamente le varie parti (e non ricompilare tutte le volte l'intero programma)

Moduli e compilazione separata (3)

Per incorporare un file esterno usare `#include` con le virgolette!

```c
/* main.c (modulo principale)*/
#include "modulo.c"

int main()
{
  return fibonacci(7);
}
```

```c
/* modulo.c */
int fibonacci(int n)
{
  if (n<0) return -1;
  if (n==0||n==1) return n;
  else return fibonacci(n-1)+fibonacci(n-2);
}
```

Attenzione, questo è però un modo SBAGLIATO di usare la compilazione separata: i file sono separati, ma vengono ricomposti prima della compilazione

Per convincersene basta provare a eseguire: `gcc -E main.c`

Moduli e compilazione separata (4)

Cosa è necessario fare:

- Separare le dichiarazioni delle funzioni dalle definizioni creando dei file di headers con estensione `.h`

```c
/* main.c (modulo principale)*/
#include "modulo.h" /* nota .h */

int main()
{
  return fibonacci(7);
}
```

```c
/* modulo.h */
int fibonacci(int n);
```

```c
/* modulo.c */
int fibonacci(int n)
{
  if (n<0) return -1;
  if (n==0||n==1) return n;
  else return fibonacci(n-1)+fibonacci(n-2);
}
```

Moduli e compilazione separata (5)

Il file di header contenente la dichiarazione della funzione `fibonacci` è sufficiente per compilare il codice del main

- `main.c` e `fibonacci.c` possono veramente essere compilati separatamente

Sequenza dei comandi:

1. `gcc -c main.c`
2. `gcc -c modulo.c`
3. `gcc main.o modulo.o`

Dopo la compilazione:

- Se modifico `main.c` devo ripetere i passi 1 e 3 (non 2)
- Se modifico `modulo.c` devo ripetere i passi 2 e 3 (non 1)
- Se modicifo `modulo.h` (e quindi anche `modulo.c`) devo ripetere tutti i passi

Moduli e compilazione separata (6)

Quando si inseriscono nei file `.h` dichiarazioni di tipo come, ad esempio, un costrutto `struct`, si può verificare il problema della doppia inclusione

```
/* modulo1.h */
struct prova {
  int primo;
  int secondo;
};
```

```
/* modulo2.h */
#include "modulo1.h"

int stupida(struct prova p);
```

```
/* main.c */
#include "modulo1.h"
#include "modulo2.h"
int main() {
  struct prova p;
  p.primo = 1;
  p.secondo = 2;
  return stupida(p);
}
```

In fase di compilazione di `main.c` avermo un errore (`struct prova` definita due volte)

Moduli e compilazione separata (6)

Il problema della doppia inclusione si risolve tramite alcune direttive per il preprocessore che consentono di realizzare una compilazione condizionale

```
/* modulo1.h */
#ifndef MODULO1_H
#define MODULO1_H
struct prova {
  int primo;
  int secondo;
};
#endif
```

```
/* modulo2.h */
#ifndef MODULO2_H
#define MODULO2_H
#include "modulo1.h"
int stupida(struct prova p);
#endif
```

```
/* main.c */
#include "modulo1.h"
#include "modulo2.h"
int main() {
  struct prova p;
  p.primo = 1;
  p.secondo = 2;
  return stupida(p);
}
```

La prima volta che si include un file `#ifndef` (if not defined) lascia proseguire. L'uso di `#define` però fa si che un successivo tentativo di esecuzione di `#ifndef` fallisca, non consentendo l'inclusione del file.

Parametri del compilatore

Alcuni parametri IMPORTANTI del compilatore `gcc` sono i seguenti:

- `-o` : Consente di specificare il nome del file eseguibile da generare (in alternativa a `a.out`. Utilizzo: `gcc -o prova prova.c` (l'eseguibile sarà il file `prova`)
- `-Wall` : Abilita la visualizzazione di tutti i messaggi di warning generati durante la compilazione
- `-pedantic` : controlla che il programma soddisfi esattamente le regole del C standard (ISO C). Segnala eventuali violazioni dello standard con messaggi di warning

E' sempre bene usare `gcc` come segue:

```
gcc -pedantic -Wall -o prova prova.c
```