Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2021/2022

2022

Indice

6. Organizzazione di un programma in file multipli

6.1. Organizzazione di un programma

Programmi complessi possono essere organizzati in più file. Questo consente una maggiore leggibilità e possibilità di riuso.

I file contenenti specifiche di programmi si distinguono in

- *file di intestazione* o *header* (estensione `.h`)
- *file di implementazione* (estensione `.c`).

Nei file header vengono inserite solamente le dichiarazioni delle funzioni, mentre nei file di implementazione anche il corpo che implementa la funzione. Solitamente la funzione `main` si trova in un file separato da quello in cui vengono definite altre funzioni.

Ogni file che usa funzioni definite altrove deve includere il relativo file header (non il file con l'implementazione `c`!)

Esempio:

File `f.h`

```
// dichiarazione delle funzioni radiceQuadrata e
    stampa
double radiceQuadrata(double x);
void stampa(double x);
```

File `f.c`

```
#include <stdio>
#include <math>
#include "f.h"

// implementazione della funzione radiceQuadrata
double radiceQuadrata(double x) {
    return sqrt(x);
}
// implementazione della funzione stampa
void stampa(double x) {
    printf("x = %f\n", f);
}
```

File `main.c`

```
#include "f.h"

// implementazione della funzione main
int main(int x)
{
    stampa(radiceQuadrata(25));
}
```

File multipli vengono compilati indicando tutti (e soli) i file `.c`. Esempio:

```
gcc -o test f.c main.c
```

6.1.1. Variabili globali definite `extern`

Quando si considerano file multipli, si può dichiarare `extern` una variabile globale. In questo caso per la variabile non viene allocata la memoria.

Quindi quando i programmi definiti su file diversi condividono la stessa variabile, questa viene dichiarata ed inizializzata in uno solo di essi, mentre negli altri viene dichiarata `extern`.

File a.c

```
float pi = 3.14f; // Viene allocata memoria per la
    variabile
```

File b.c

```
extern float pi; // La variabile pi e' definita in un
    altro file
```

L'uso di variabili `extern` è fortemente scoraggiato in quanto possibile causa di inconsistenze. Lo scambio di informazione tra funzioni deve essere basato quanto più possibile sull'uso dei parametri.

6.2. Il processo di compilazione

Il processo di generazione dei file eseguibili è costituito da tre fasi.

- preprocessore
- generazione codice oggetto
- collegamento (*linking*)

6.2.1. Il preprocessore

Trasforma un programma con direttive di preprocessamento denotate dal simbolo `#` in un programma C in cui le direttive sono state eliminate, dopo essere state eseguite.

- direttive di inclusione
- definizioni di macro
- compilazione condizionale
- altre direttive (si rimanda al manuale)

6.2.1.1. Direttiva `include`

```
#include <stdio.h>
```

La linea viene sostituita dal testo del file corrispondente.

```
gcc -E filespanso.c nomefile.c
```

Produce un file C risultato del preprocessore.

6.2.1.2. Direttiva `define`

Abbiamo già visto l'uso di `define` per definire valori costanti

```
#define pi 3.14f
```

A differenza della dichiarazione di una costante, l'effetto di questa definizione consiste nella sostituzione testuale (da parte del preprocessore) di tutte le occorrenze dell'identificatore `pi` con `3.14f`.

```
#define identificatore elenco-sostituzione
```

Si noti che tutti i simboli che si trovano a destra dell'identificatore (spazi esclusi) vengono presi come sostituzione.

Inoltre le definizioni restano valide fino alla fine del file.

Esempi

```
#define N = 100  /*** SBAGLIATO ***/

#define BOOL int
#define TRUE 1
#define FALSE 0

#define BEGIN {
#define END }
```

6.2.2. Macro parametriche

Tramite la direttiva define si possono definire anche macro con parametri.

define con parametri

Sintassi:

```
#define identificatore(x1, ...xn) sostituzione
```

(x1, ...xn) sono i parametri ed andranno sostituiti testualmente ai corrispondenti parametri formali delle espressioni: identificatore(y1, ...,yn)

Esempio

```
#define MAX(x,y) ((x)>(y)?(x):(y))
...
i = MAX(j+k, m-n);
```

diventa

```
i = ((j+k)>(m-n)?(j+k):(m-n));
```

Si noti che in questo caso abbiamo una definizione di macro con sostituzione di parametri per nome. Inoltre in questo caso l'intero corpo della macro viene sostituito testualmente in corrispondenza dell'uso. Si noti anche che non viene effettuato nessun controllo di tipo.

Le definizioni di macro possono utilizzare diversi altri accorgimenti sintattici (ad esempio l'uso del simbolo #) per i quali si rimanda al manuale del linguaggio.

Esistono inoltre diverse macro predefinite: _DATE_, _TIME_,...

6.2.3. Compilazione condizionale

Il compilatore C, più precisamente il preprocessore consente di utilizzare delle direttive che hanno una forma simile alle istruzioni condizionali, ma un effetto completamente diverso, in quanto consentono di selezionare quali parti di codice dovranno essere compilate. Sebbene tali direttive siano utili solo quando si devono gestire grandi quantità di codice, e quindi esulino dagli scopi di questa dispensa, si accenna brevemente al loro utilizzo.

Compilazione condizionale

Sintassi:

```
#if condizione1
  sequenza-istruzioni1
#elif condizione2
  sequenza-istruzioni2
...
#else
  sequenza-istruzioni-else
```

- `condizione` è un'espressione
- `sequenza-istruzioni` è una sequenza di istruzioni

Semantica:
Viene valutata prima la `condizione1`. Se la valutazione fornisce un valore diverso da 0, viene inclusa nel programma `sequenza-istruzioni1`. Lo stesso avviene per tutte le successive coppie condizione sequenza-istruzioni, corrispondenti alle sezioni #elif. Se nessuna delle condizioni fornisce un valore maggiore di 0, viene inclusa nel programma `sequenza-istruzioni-else`.

Esempio:

```
...
#define PALMARE 1
...
#if PALMARE
  short a,b;
#else
  int a,b;
```

L'esecuzione di questo frammento di codice produce un programma che dichiara le variabili a,b di tipo short, quando la variabile PALMARE è definita pari ad 1, altrimenti le dichiara di tipo int.

6.2.3.1. Direttive #ifdef ed #ifndef

Queste varianti della direttiva #if consentono di verificare se un identificatore è stato precedentemente definito o meno.

Esempio:

```
#ifndef MYMATH_H
#define MYMATH_H
// calcola le equazioni di secondo grado
int secondoGrado(double a,double b,double c, double*
    x1r,
    double* x1i,double* x2r,double* x2i);

// calcola la radice quadrata con il metodo di Newton
float radiceNewton(float x);
#endif
```

In questo caso la compilazione condizionale consente di evitare di includere delle definizioni se queste sono già state precedentemente incluse, evitando in questo modo di introdurre ridondanze nel codice.

6.2.4. Struttura del compilatore

Il processo di compilazione è articolato in diverse fasi:

- Analisi lessicale (sequenza di token)
- Analisi sintattica (albero sintattico)
- Analisi semantica statica (albero sintattico annotato)
- Generazione codice intermedio
- Ottimizzazione
- Generazione codice macchina

A ciascuna fase corrisponde un formato di output prodotto, in alcuni casi la fase di ottimizzazione del codice dipende dalla particolare architettura di elaborazione considerata e pertanto le ultime due fasi sono mescolate. Esistono compilatori che effettuano tutte le fasi del processo di compilazione in una sola passata, altri che effettuano più passate, ciascuna delle quali produce un output completo per la fase di compilazione successiva (es. preprocessore C).

6.2.4.1. Fasi della compilazione

La compilazione si divide anche in

- analisi/sintesi (prime 3 ed ultime 3 fasi)
- front-end/back-end (prime 4 ed ultime 2 fasi)

La prima suddivisione, che distingue l'analisi dalla sintesi, ha come risultato intermedio l'albero sintattico del programma in input corredato di informazioni relative alla semantica del programma.

La seconda suddivisione ha invece una carattere tipicamente ingegneristico, in quanto la prima parte del processo che termina con la generazione del codice intermedio è indipendente dalla particolare architettura di elaboratore per la quale viene generato il codice. Pertanto, la realizzazione del compilatore per un linguaggio di programmazione

per diverse architetture ha in comune il front-end, mentre il back-end deve essere differenziato in base alle caratteristiche dell'architettura sulla quale verrà eseguito il programma.

Per alcuni linguaggi di alto livello il C rappresenta il linguaggio intermedio. Cioè il compilatore per il linguaggio produce codice C che a sua volta viene compilato con il compilatore C per produrre il codice eseguibile

Si usa il termine *cross-compilatore* per indicare un compilatore per un'architettura diversa da quella sulla quale è implementato il compilatore stesso.

La realizzazione di compilatori sfrutta spesso una tecnica denominata *bootstrapping* che si basa sulla realizzazione del compilatore di un piccolo sottoinsieme del linguaggio, che poi viene utilizzato per scrivere il compilatore del linguaggio stesso. In questo modo, il compilatore del linguaggio viene scritto nel linguaggio stesso.

6.2.5. Compilazione incrementale

Quando il programma prevede molti componenti, per snellire il processo di generazione del codice eseguibile, i file possono essere compilati separatamente e in modo incrementale. Con riferimento all'esempio precedente, il comando:

```
gcc -c f.c
```

produce il file `f.o` che contiene il codice oggetto, cioè il codice tradotto in linguaggio macchina. Il codice oggetto, per poter essere eseguito, deve prima essere collegato agli altri moduli che ad esso fanno riferimento o a cui esso fa riferimento. Questa operazione si chiama *collegamento (linking)* ed usualmente viene fatta dal compilatore stesso. Il collegamento ha lo scopo di risolvere tutti i riferimenti ai simboli del programma. In C il collegamento viene effettuato con il comando `gcc`:

```
gcc -o test f.o main.c
```

consente di fare il collegamento del codice oggetto precedentemente compilato.

6.2.5.1. Costruzione incrementale del codice eseguibile

La sequenza completa di operazioni relativa all'esempio precedente risulta essere:

```
gcc -c f.c
gcc -c main.c
gcc -o main f.o main.o
```

I primi due comandi generano il codice oggetto corrispondente ai file `f.o` e `main.o`, rispettivamente. Il terzo comando effettua il collegamento e genera il file eseguibile `main`.

6.2.5.2. Codice rilocabile

Il codice generato dal compilatore si definisce codice *rilocabile*. Con questo termine si caratterizza il fatto che gli indirizzi di memoria sono indicati nel programma in modo relativo, facendo riferimento ad una collocazione del programma arbitraria.

Questo accorgimento consente di utilizzare in modo efficiente la memoria a disposizione del calcolatore, dato che la zona di memoria in cui il programma viene posizionato in fase di esecuzione, può essere cambiata, a seconda delle necessità. Tuttavia, ciò comporta che il programma eseguibile debba subire un'ulteriore trasformazione, che riguarda il calcolo degli indirizzi di memoria.

La *rilocazione* del codice consiste nella determinazione degli indirizzi di memoria in cui viene memorizzato il programma al momento dell'esecuzione.

6.3. Compilazione tramite `make`

Quando si costruiscono sistemi complessi, costituiti da molti moduli da compilare separatamente, diventa indispensabile utilizzare degli strumenti per gestire il processo di compilazione. Negli ambienti di programmazione come ad esempio *Geany*, il processo di costruzione del codice eseguibile viene controllato in modo semplificato con meccanismi specifici.

Lo strumento di base offerto da UNIX a questo scopo è il comando `make`.

La specifica dei passi di compilazione avviene tramite un file costituito da una sequenza di regole, chiamato `makefile`.

Esempio:

```
provamymath: provamymath.o mymath.o
  g++ -o provamymath provamymath.o mymath.o

provamymath.o: provamymath.c mymath.h
  g++ -c provamymath.c

mymath.o: mymath.c mymath.h
  g++ -c mymath.c
```

Ciascuna regola definisce un passo del processo di compilazione ed è costituita da un identificatore che corrisponde al nome del file prodotto dal passo di compilazione ad essa corrispondente, da un elenco di dipendenze, e, nella riga seguente, preceduto dal carattere "tab" il comando di compilazione.

L'esecuzione del comando `make` comporta la verifica di quali file sono stati modificati dopo l'ultima compilazione e l'esecuzione dei comandi influenzati (dipendenti) dai cambiamenti individuati, in modo da minimizzare il tempo di compilazione.

Esempio: Se è stato modificato il file `mymath.c` non occorre ricompilare il file `provamymath.o`.

Per una trattazione completa del comando `make` si rimanda ai manuali di UNIX. Vale la pena notare che questo strumento, che è indipendente dal linguaggio di programmazione utilizzato, offre numerose funzionalità, il cui uso richiede una conoscenza approfondita. Per questo motivo, sono stati sviluppati altri strumenti con l'obiettivo di semplificare l'utilizzo di `make`; tra i più diffusi `cmake`. Si noti, infine, che le funzionalità di gestione del processo di compilazione offerte dagli ambienti di programmazione, sono in genere implementate tramite il comando `make`.