Fondamenti di Informatica e Laboratorio T-AB
T-16 – Progetti su più file. Funzioni come parametro. Parametri del main

Paolo Torroni

Dipartimento di Elettronica, Informatica e Sistemistica
Università degli Studi di Bologna

Anno Accademico 2008/2009

Sommario

Librerie

- Programmi con molte funzioni richiedono una organizzazione su più file.
- Strutture dati e codice che appartengono a uno stesso contesto vengono di solito raggruppate a formare **librerie**.
- Il codice di una libreria è contenuto in uno o più file. Esempi:
  - il codice di molte funzioni "di libreria" per l'input/output (`printf`, `scanf`, ...) è contenuto in un file chiamato `stdio.c`
  - funzioni per la gestione dinamica della memoria (`malloc`, `free`, ...) → libreria → `stdlib.c`
  - funzioni matematiche (`cosh`, `sqrt`, `exp`, ...) → libreria → `math.c`
  - etc.
- Anche voi potete generare una libreria. Esempi:
  - le funzioni del tipo di dato astratto `List`
  - le funzioni e strutture dati per la simulazione di un dispositivo meccanico
  - i simboli che indicano i valori massimi/minimi di certe variabili

Dichiarazione vs. definizione

- La strutturazione su più file è una questione di manutenzione del codice, non di funzionalità.
  - Un programma su più file deve poter essere compilato come se fosse su un file solo
- La separazione tra dichiarazione e definizione aiuta.
- La **dichiarazione** (tipi, funzioni, macro, variabili globali)
  - fornisce una **specifica** di un componente
  - può essere duplicata senza problemi
  - non genera codice macchina quando viene compilata
  - *nota*: si possono dichiarare anche variabili globali (`extern`).
- La **definizione** (costanti, variabili, funzioni)
  - non può essere duplicata
  - associa un identificatore a qualcosa che ha un riscontro "fisico" (posizione in memoria di dati o codice)
  - produce codice macchina in fase di compilazione

Progetti su più file

- Bisogna sempre seguire due regole:
  1. Definire o anche solo **dichiarare** gli identificatori prima dell'utilizzo
  2. Mai duplicare definizioni di identificatori.
- I file sorgente hanno l'estensione `.c`, e contengono
  - codice (definizioni di funzioni)
  - dati (definizioni di variabili)
- La testa di un file sorgente (**header**) contiene quindi tipicamente una serie di dichiarazioni/definizioni (regola 1.)
- Il codice compilato a partire dai sorgenti viene messo assieme tramite il **linker**.
  - Notate che il codice della funzione `main()` sarà un uno solo dei file sorgenti che vogliamo collegare (regola 2.).

Progetti su più file

- Immaginiamo il seguente scenario:
  - il sorgente di una libreria di funzioni è in un file `"funz.c"`;
  - in particolare, in `funz.c` definisce una funzione `f()`;
  - voglio utilizzare `f()` nel `main()`;
  - il `main()` è definito all'interno di un altro sorgente, chiamato `"main.c"`.
- Cosa devo fare?
- Per la regola 1.,
  - se `main.c` chiama una funzione `f()` definita in `funz.c`,
    - è necessario che `f()` sia stata definita o dichiarata prima di quando viene usata (all'interno di `main.c`).
- Per la regola 2.,
  - `f()` non può essere ridefinita.
- Quindi, `main.c` deve contenere la dichiarazione di `f()`, ma non la definizione.

Header file

- La soluzione che si adotta di solito è questa:
  - Quando si implementa una libreria, si scrive anche un blocco (header) che contiene tutte le dichiarazioni di tutti gli identificatori messi a disposizione dalla libreria
    - comprese le interfacce di tutte le funzioni, le dichiarazioni di tipo, etc.
  - L'header viene poi copiato, tale e quale, in cima a tutti gli altri file sorgenti che utilizzano almeno una funzione di tale libreria.
- Per fortuna, non è necessario fare copia-incolla a mano...
- Esiste il pre-processore.
- Grazie al pre-processore, è possibile scrivere gli header in file separati (**header file**), che vengono poi inclusi in cima ai vari sorgenti `.c`, ove ce ne fosse bisogno.
- Per convenzione, gli header file hanno estensione `.h` (es., `stdio.h`, `stdlib.h`, `math.h`, `mio_header.h` ...)

Il pre-processore

- Il pre-processore interviene prima del compilatore.
- Non genera codice macchina.
- Serve a operare delle sostituzioni sintattiche.
  - Riscrive il programma sorgente in un nuovo programma C, che non contiene più istruzioni per il pre-processore (e che può essere quindi compilato).
- Le istruzioni per il pre-processore sono precedute da #
- Alcune utili istruzioni per il pre-processore:
  - `#define` *`symbol translation`* per definire MACRO
  - `#include <`*`header.h`*`>` per includere header di libreria del C (quelli che si trovano in una cartella nota all'installazione del compilatore)
  - `#include "`*`header.h`*`"` per includere i propri header (che si trovano in una cartella locale)

Funzione qsort

- Una funzione di libreria molto utile è la funzione `qsort`, dichiarata in `stdlib.h`.
- Implementa l'ordinamento degli elementi di un vettore.
- Il tipo degli elementi del vettore può essere qualsiasi!
- Come è possibile?
  - La funzione `qsort` implementa un algoritmo generale
  - All'interno di questo algoritmo, `qsort` chiama una funzione `compare` che effettua il confronto
  - Parametrizzando la funzione `compare` è possibile applicare l'algoritmo generale a casi specifici.
- Alcuni esempi:
  - vettore di interi
    - allora, x "viene prima di" y se $x < y$.
  - vettore di stringhe
    - allora, x "viene prima di" y se `strcmp(`x`,`y`)`< 0.

Funzione qsort

- La funzione `compare` è passata alla funzione `qsort` come parametro.
- L'interfaccia di `qsort` infatti è:

```
void qsort(
   void *buf, size_t num, size_t size,
   int (*compare)(const void *, const void *) );
```

- Dal manuale:
  The `qsort()` *function sorts* `buf` *(which contains* `num` *items, each of size* `size`*) using Quicksort.*
  The `compare` *function is used to compare the items in* `buf`*.* `compare` *should return*
  - *negative if the first argument is less than the second,*
  - *zero if they are equal, and*
  - *positive if the first argument is greater than the second.*

  `qsort()` *sorts* `buf` *in ascending order.*

Funzione qsort

- L'interfaccia di `qsort` è:

```
void qsort(
   void *buf, size_t num, size_t size,
   int (*compare)(const void *, const void *) );
```

- Notate come viene definito il parametro formale `compare`:

```
int (*compare)(const void *, const void *)
```

- indica che:
  - `compare` è una funzione (perché seguita da parentesi tonde)
  - che ha due parametri di tipo puntatore
  - i quali puntano a dati che non vengono modificati da `compare`
  - e restituisce un valore di tipo `int`
- in pratica, ne definisce l'**interfaccia**.

Funzione qsort

- Una possibile funzione che implementa questa interfaccia è la seguente:

```
int cmp_int( void* a, void* b ) {
   return ( *( int* )a )-( *( int* )b );
}
```

- `cmp_int` si può utilizzare come funzione di confronto per ordinare un vettore di interi.

```
main() {
   int V[100];
   ...
   qsort(V, 100, sizeof( int ), cmp_int );
}
```

- Nota: per verificare quanti confronti effettua `qsort`, è possibile definire un contatore all'interno di `cmp_int`
  - Si può ricorrere a una variabile `static`.
- Un'altra funzione con un parametro funzione è `bsearch()`.

Parametri del main

- Anche il `main()` ha dei parametri, che corrispondono a quelli inseriti nella riga di comando.
- Ad esempio, se l'eseguibile prodotto dalla compilazione è `prog.exe`, da riga di comando posso ordinare (al S.O.) l'esecuzione di `prog.exe` con dei parametri:
  - `prog.exe par1 2 -v "ultimo parametro"`
- I parametri possono essere utilizzati dall'interno del `main()`
- Vengono inseriti in un vettore, chiamato `argv`, che è un **parametro formale** del `main()`
- La dimensione logica di `argv` è un altro parametro formale del `main()`: `argc`.

Parametri del main

```
prog.exe par1 2 -v "ultimo parametro"
```

- Per utilizzare `argc` e `argv`, bisogna indicare esplicitamente `argc` e `argv` come parametri formali del `main()`:

```
main( int argc, char *argv[] ) {
   ...
   printf( "primo parametro:  %s", argv[1] );
}
```

- Da notare due cose;
  - `argv` è un vettore di stringhe.
  - Il primo elemento di `argv` è il nome stesso dell'eseguibile.
- Se `prog.exe` viene invocato come nell'esempio, si ha:
  - `argv` contiene `prog.exe`, `par1`, `2`, `-v`, `ultimo parametro`;
  - `argv[0]` vale `prog.exe`;
  - `argc` vale 5;
  - `argv[4]` vale `ultimo parametro`.