Corso di Laboratorio di Sistemi Operativi
A.A. 2016–2017

Lezione 12

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

A.A. 2016–2017 - Primo Semestre - 11/11/2016

Compilazione separata C e gestione del progetto

Distribuire il codice su diversi file

In progetti appena più grandi di un esercizio è impensabile scrivere tutto in un unico file. I programmi C possono essere scritti su più file, adottando alcuni accorgimenti:

- Uno dei file del progetto dovrà contenere la funzione `main()`.
- Una funzione definita in un file può chiamarne una definita in un file diverso, purchè sia comunque visibile la sua **dichiarazione**. Esempio:

```
int add(int, int); // Solo la dichiarazione

int add3(int x, int y, int z) {
  return add(x, add(y,z));
}
```

Distribuire il codice su diversi file

File di intestazione (header)

In ogni file è quindi necessario avere la dichiarazione delle funzioni definite altrove che verranno chiamate.
Non solo: lo stesso vale per tipi di dato (`struct`), costanti, ecc...

Per non ripetere tutte le dichiarazioni in molti posti, la pratica vuole che si riuniscano nei cosiddetti **file di intestazione**:

- Un file di intestazione (chiamato anche **header**), solitamente chiamato con l'estensione `.h`, contiene solo **dichiarazioni** di un codice
- Il file viene incluso con la direttiva `#include <file.h>` del preprocessore, che include testualmente il contenuto del file incluso nel file corrente.
- Il famoso `stdio.h` non è altro che questo.

Distribuire il codice su diversi file

Esempio

File `add.h`:

```
// Funzione che permette
// di sommare due numeri interi
int add(int x, int y);
```

File `add.c`:

```
#include "add.h"
int add(int x, int y) {
  return x + y;
}
```

File `main.c`:

```
#include <stdio.h>

#include "add.h"

int main()
{
  printf("3 + 4 = %d\n", add(3,4));

  return 0;
}
```

Si compila con:

```
$ clang main.c add.c -o add
$ ./add
3 + 4 = 7
```

Includere più volte lo stesso header

Spesso gli header saranno scritti da una persona/team, e usati da un altra. Ipotizziamo che:

- un header `A.h` ne include un altro `B.h`
- anche chi usa `A.h` include `B.h` (ignorando, com'è giusto che sia, che `A.h` già lo include)

Allora, il file `A.h` verrà incluso due volte e le dichiarazioni contenute saranno processate due volte, portando ad errori.

Serve un meccanismo per includere una volta sola sola il contenuto di un header anche quando questo viene potenzialmente incluso più volte.

Includere più volte lo stesso header

Le direttive #ifdef/#ifndef e le header guards

La direttiva del preprocessore #ifdef serve a scopi come questo:

```
#define SIMBOLO
#ifdef SIMBOLO
  codice...
#endif
```

Il codice tra #ifdef SIMBOLO e #endif verrà incluso solo se è stata prima definita la costante del preprocessore SIMBOLO.
La direttiva #ifndef funziona ugualmente, ma **esclude** il codice se la costante del preprocessore è stata definita.
Nota: È anche possibile specificare un ramo #else.

Includere più volte lo stesso header

La direttiva `#ifdef` e le header guards

Sfruttando questo meccanismo possiamo scrivere un header file che si accorga nel caso venga incluso più volte:

Esempio sul file `add.h` visto prima:

```
#ifndef ADD_H__
#define ADD_H__

// Funzione che permette di sommare due numeri interi
int add(int x, int y);

#endif
```

Il processo di compilazione

La compilazione di un programma C avviene in più fasi:

1. Preprocessing
2. Compilazione
3. Assemblaggio
4. Linking

Il processo di compilazione

Compilazione

La compilazione, in senso più stretto, riguarda la traduzione del programma da codice C a codice *assembly*. È possibile chiedere al compilatore di fermarsi dopo questa fase con l'opzione `-S`.

Ad esempio, se in un file si scrive una sola funzione:

```
int add(int x, int y) {
  return x + y;
}
```

Ne si può ottenere il codice assembly:

```
$ clang -S add.c
$ cat add.s
<codice asm...>
```

Il processo di compilazione

Assemblaggio

L'assemblaggio consiste nel trasformare il codice assembly (che è ancora testo) in codice macchina in formato binario eseguibile.
Si può chiedere al compilatore di fermarsi a questa fase con l'opzione `-c`:

```
$ clang -c add.c
$ ls
add.c add.o
```

Il file risultante, `add.o`, è detto **file oggetto**. Contiene codice eseguibile, ma non è ancora un programma completo.

Il processo di compilazione

Linking

Il linking, effettuato da un programma chiamato **linker**, prende un **insieme** di file oggetto e di **librerie dinamiche** e compone un programma eseguibile completo:

- I file oggetto vengono riuniti in un tutt'uno, e i **riferimenti** a chiamate di funzioni chiamate in un file ma non definite vengono risolti con le funzioni definite in altri file.
- Le chiamate a funzioni ancora irrisolte vengono collegate a funzioni esportate dalle librerie dinamiche elencate (es. la libreria standard).

Compilazione separata

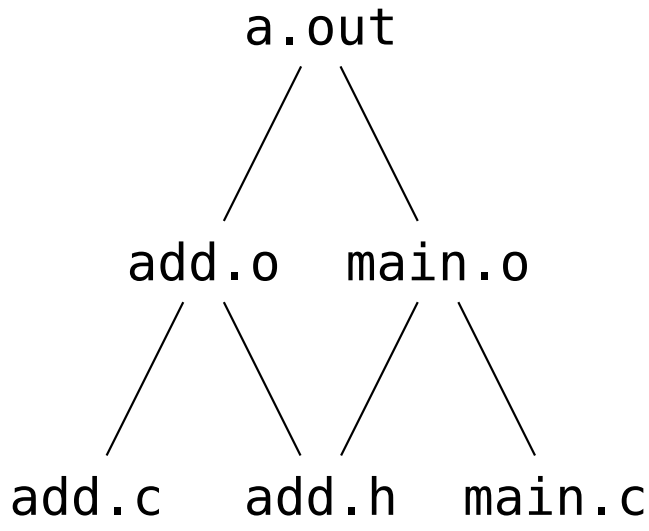
La compilazione del programma add dell'esempio precedente può essere quindi fatta anche per stadi, un file alla volta:

```
$ clang add.c -c
$ clang main.c -c
$ clang add.o main.o -o add
```

Solitamente si fa proprio in questo modo, per efficienza:

- Si possono compilare più file contemporaneamente.
- Se viene cambiato un file, basta ricompilare solo quello e poi rifare il linking.
- Va però tenuto traccia di quali file sono stati modificati dall'ultima compilazione, e in quali file vengono inclusi gli header.

Grafo delle dipendenze



Il grafo mette in luce le dipendenze fra i vari file, ad esempio da `add.h` dipende sia da `add.o` che da `main.o`.

Quindi, in caso di modifiche ad `add.h`, percorrendo il grafo verso l'alto notiamo che devono essere aggiornati sia `add.o` che `main.o`, mentre invece modificando `main.c` va ricompilato solo il file stesso.

Il comando make

Il comando make è un programma che si occupa di gestire la compilazione di programmi C (o di altri linguaggi) .

- Il grafo delle dipendenze viene codificato in un file di testo chiamato `Makefile` che risiede nella stessa directory dei file sorgente.
- Esempio di `Makefile`:

```
add: add.o main.o
    clang add.o main.o
add.o: add.h add.c
    clang -c add.c
main.o: add.h main.c
    clang -c main.c
```

- Invocando il comando make direttamente, viene interpretato il `Makefile` e compilato il programma.

Il comando make

La sintassi di base dei `Makefile` è molto semplice.

- Un `Makefile` è composto da regole specificate dalla seguente sintassi (l'ordine delle regole non è importante):

```
target : source file(s)
    command
```

- Nota: Il comando deve essere preceduto da un <Tab>.
- Il comando `make` controlla le date di ultima modifica dei file. Se un file (source) ha una data di modifica più recente di quella dei file che da esso dipendono (target), questi ultimi vengono aggiornati eseguendo i comandi specificati nelle regole del `Makefile`.

Introduzione alla programmazione di sistema

Programmazione di sistema

Il C è un linguaggio legato a doppio filo con i sistemi `UNIX`:

- Il 99% del sistema operativo è scritto in C
- L'interfaccia che il sistema operativo fornisce ai programmi utente è definita in termini di funzioni C.

Chiamate di sistema e libreria standard

Le chiamate di sistema sono il modo in cui un processo chiede al sistema operativo di fare qualcosa.

- Si attivano tramite degli interrupt della CPU
- Al programmatore C, però, vengono fornite delle **funzioni**, dichiarata in file di intestazione e implementate in una libreria, solitamente chiamata `libc`, che implementa anche le funzioni della libreria standard.
- Sono quindi due le fonti di funzioni "standard" che un programmatore C ha a disposizione:
  - Funzioni della libreria standard del linguaggio, definite dallo standard ISO C.
  - Funzioni di interfaccia per le chiamate di sistema, definite dallo standard POSIX (Portable Operating System Interface).

Programmazione di sistema in ambiente UNIX

Programmazione di sistema significa gestire ed effettuare operazioni di basso livello, messe a disposizione dal sistema operativo, per operare su:

- file
- directory
- processi
- comunicazione tra processi
- altro...

Funzionalità e operazioni solitamente disponibili all'utente tramite comandi della shell, sono implementate in C tramite chiamate di sistema apposite.

Accesso ai file

Accesso a file

Un primo esempio di accesso a servizi esposti dal sistema operativo è l'accesso ai file.

Esistono due modi complementari per accedere ai file in un programma C:

- Funzioni della libreria standard (`fopen()`, `fread()`, ecc...), dichiarate da `<stdio.h>`
- Chiamate di sistema POSIX (`open()`, `read()`, ecc...), più a basso livello ma più flessibili, dichiarate da `<unistd.h>`

Aprire un file

Prima di leggere/scrivere su un file, un programma C deve **aprire** il file tramite la funzione `fopen()`:

```
FILE *fopen(char *name, char *mode);
```

La funzione `fopen` prende come parametri:

- il nome del file, `name`
- una stringa, `mode`, che indica il **modo** di utilizzo del file:
  - `"r"` (lettura)
  - `"w"` (scrittura)
  - `"a"` (append)

e restituisce un **puntatore** ad una struttura di tipo `FILE`, detto "file pointer" che identificherà il file aperto nelle successive chiamate ad altre funzioni di manipolazione di file.

Lettura e scrittura

Una volta aperto, un file può essere letto/scritto in vario modo.
Il modo più facile sono alcune funzioni molto familiari:

```
int fprintf(FILE *fp, char *format, ...);
int fscanf(FILE *fp, char *format, ...);
int fgetc(FILE *fp);
int fputc(FILE *fp);
```

Le due funzioni operano esattamente come `printf()`, `scanf()`, `getchar()` e `putchar()`, rispettivamente, ma su un file qualsiasi invece che sullo standard input/output.

Chiudere un file

Al termine delle operazioni di lettura/scrittura di un file, è buona norma rilasciare il file pointer, utilizzando la funzione

```
int fclose(FILE *fp);
```

Si applicano le stesse avvertenze riguardo la coppia di funzioni `malloc()`/`free()`.

Accorgersi della fine del file

Alcune funzioni di lettura, come `fgetc()` ritornano la costante `EOF` quando incontrano la fine del file o quando incontrano un **errore**.

Per distinguere i due casi è comoda la funzione `feof()`:

```
int feof(FILE *fp);
```

La funzione restituisce "vero" (un numero positivo) se la posizione di lettura del file è arrivata alla fine.

Lettura e scrittura

I file predefiniti

In un programma C sono sempre presenti tre file pointer standard, già aperti e pronti all'uso:

- `stdout` è un file aperto in scrittura, puntato sullo **standard output** del processo.
- `stdin` è un file aperto in lettura, puntato sullo **standard input** del processo.
- `stderr` è un file aperto in scrittura, puntato sullo **standard error** del processo.

Lettura e scrittura

I file predefiniti (2)

In effetti, `printf()` e `scanf()` sono equivalenti ad una chiamata a `fprintf()` e `fscanf()` passando `stdout` e `stdin` come file.

```
printf("Ciao mondo!");           // Queste due righe...
fprintf(stdout, "Ciao mondo!"); // ...sono equivalenti
```

Accesso ai file

Esempio

```c
#include <stdio.h>

int main(int argc, char **argv) {
  if(argc < 2) {
    fprintf(stderr, "Fornire il nome del file\n");
    return 1;
  }

  char *filename = argv[1];

  FILE *file = fopen(filename, "r");
  if(!file) {
    fprintf(stderr, "Errore nell'apertura del file!\n");
    return 2;
  }

  int n = 0, sum = 0;
  while(fscanf(file, "%d", &n) == 1) {
    sum += n;
  }

  if(!feof(file)) {
    fprintf(stderr, "Il file non conteneva solo numeri.\n");
    return 3;
  }

  printf("Somma dei numeri contenuti: %d\n", sum);

  return 0;
}
```

Esercizi

Per venerdì 18 novembre

1. Riscrivere alcuni dei programi scritti nelle settimane passate come esercizio, dividendo il codice in più file (separando, ad esempio, la funzione `main()` dalle altre) con relativi file di intestazione e `Makefile`.
2. Riscrivere alcuni dei programmi scritti nelle settimane passate come esercizio, facendoli però operare su file passati come argomento sulla riga di comando.

Esercizi (2)

Per venerdì 18 novembre

3. Scrivere un programma C, versione semplificata del comando Unix `cat`, per l'append di uno o più file su standard output.
4. Scrivere un programma C, versione semplificata del comando Unix `cmp` per il confronto di due file, che stampa la prima linea su cui i file differiscono.
   Nota: Si può reimplementare un analogo della funzione `readline()` vista negli esercizi precedenti, che però utilizzi `fgetc()` invece di `getchar()`, oppure investigare l'utilizzo della funzione `fgets()`.