Algoritmi e Strutture di Dati

Esercitazioni in linguaggio C

Compilazione in linguaggio C

m.patrignani

Nota di copyright

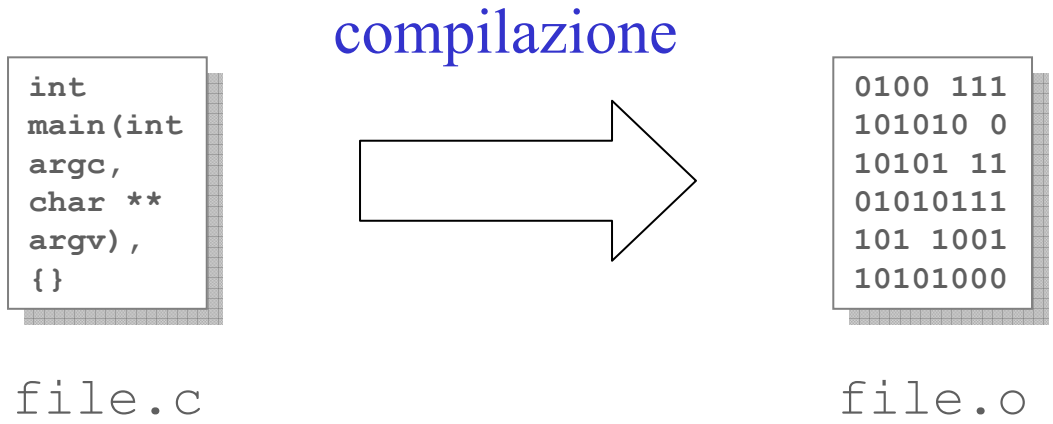
- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

Richiami di linguaggio C

- Compilazione
- Linking
- Suddivisione del codice in più file
- Dichiarazioni e definizioni
- Compilazione secondo lo standard ANSI

Compilazione in linguaggio C

- La compilazione traduce il codice sorgente nel codice oggetto (cioè compilato)
  - usualmente il codice sorgente è in uno o più file con estensione .c
  - il codice oggetto viene messo in file con estensione .o



Compilazione da linea di comando

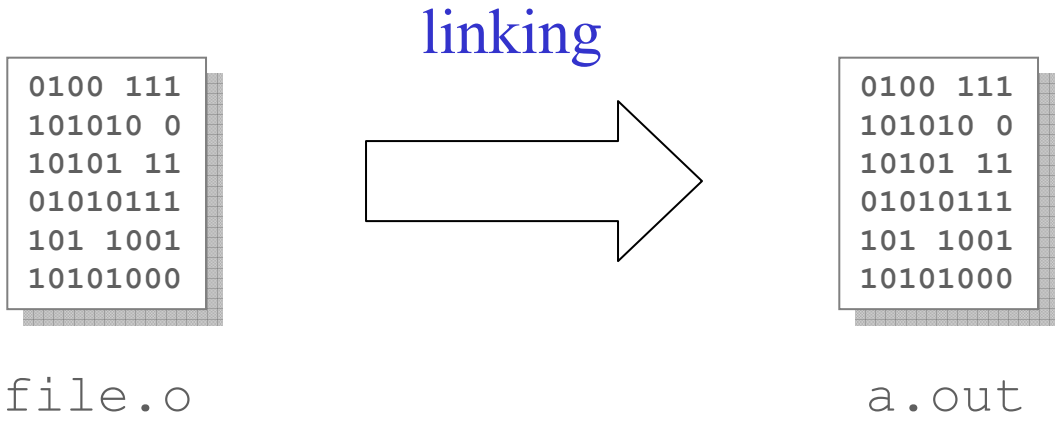
- Avviene chiamando un compilatore (cc, gcc, ecc) specificando che si vuole compilare un file sorgente

```
localhost:~$ ls
file.c
localhost:~$ cc -c file.c
localhost:~$ ls
file.c file.o
```

  - l'opzione **`-c`** significa "voglio solo la compilazione"

Linking in linguaggio C

- La fase di linking trasforma il codice oggetto in un file eseguibile
  - vengono anche utilizzate (linkate, appunto) le librerie standard del linguaggio C



Linking da linea di comando

- Avviene chiamando un compilatore (cc, gcc, ecc) e fornendo come parametro il file oggetto

```
localhost:~$ ls
file.c file.o
localhost:~$ cc file.o -o eseguimi
localhost:~$ ls
eseguimi file.c file.o
```

- Se non viene specificato il nome del file di output (**`-o eseguimi`**) allora viene generato un file eseguibile di nome **`a.out`**

Un esempio di errore di linking

- Il file linkato deve contenere la funzione **`main`**

```
localhost:~$ cc file.o -o eseguimi
/usr/lib/gcc/x86_64-linux-
gnu/4.9/../../../x86_64-linux-
gnu/crt1.o: In function `_start':
(.text+0x20): undefined reference to
`main'
collect2: error: ld returned 1 exit
status
localhost:~$
```

  - in realtà il messaggio d'errore dice che la funzione standard **`_start`** non ha trovato la funzione **`main`**

Compilazione e linking

- Si può compilare e linkare con un solo comando

```
localhost:~$ ls
prova.c
localhost:~$ cc prova.c -o prova
localhost:~$ ls
prova prova.c
```

- Il file **`prova.o`** non viene nemmeno conservato
- In un Integrated Development Environment (IDE) la compilazione e il linking avvengono automaticamente al "build" del progetto

Codice distribuito in file diversi

- Quando il codice viene distribuito in diversi file

…

… la compilazione e il linking generano un singolo file eseguibile

– uno e uno solo dei file oggetto deve contenere la funzione **`main`**

Creazione del file eseguibile

- Per generare un file eseguibile è necessario
  1. che la compilazione vada a boun fine e generi un file oggetto per ogni singolo file compilato
  2. che il linkaggio vada a buon fine
     - ogni funzione che viene utilizzata in un file oggetto deve essere contenuta in un altro file oggetto o in una libreria specificata nella fase di linking

Esempio di errore di compilazione

```
int main(int argc, char** argv) {
   float a = 0.5;
   float b = sin(a);
}
```

```
localhost:~$ cc file.c -o eseguimi
file.c: In function ‘main’:
file.c:3:13: warning: incompatible implicit
declaration of built-in function ‘sin’
   float b = sin(a);
             ^
/tmp/cctqIk3Y.o: In function `main':
file.c:(.text+0x1e): undefined reference to `sin'
collect2: error: ld returned 1 exit status
localhost:~$
```


Esempio di errore di compilazione

```c
#include <math.h> /* contiene sin */

int main(int argc, char** argv) {
  float a = 0.5;
  float b = sin(a);
}
```

```
localhost:~$ cc file.c -o eseguimi
/tmp/cc6y9GHF.o: In function `main':
file.c:(.text+0x1e): undefined reference to `sin'
collect2: error: ld returned 1 exit status
localhost:~$
```

- Anche includendo il file **`math.h`** l'eseguibile non viene generato

Esempio di errore di compilazione

```c
#include <math.h> /* contiene sin */

int main(int argc, char** argv) {
   float a = 0.5;
   float b = sin(a);
}
```

```
localhost:~$ cc -c file.c
localhost:~$ ls
file.c file.o
localhost:~$ ls
```

- La compilazione, in realtà, va a buon fine

Esempio di errore di compilazione

```c
#include <math.h> /* contiene sin */

int main(int argc, char** argv) {
   float a = 0.5;
   float b = sin(a);
}
```

```
localhost:~$ cc file.o -o eseguimi
file.o: In function `main':
file.c:(.text+0x1e): undefined reference to `sin'
collect2: error: ld returned 1 exit status
localhost:~$
```

- L'errore è nella fase di linkaggio

Correzione dell'errore

```c
#include <math.h> /* contiene sin */

int main(int argc, char** argv) {
   float a = 0.5;
   float b = sin(a);
}
```

```
localhost:~$ cc file.o -lm -o eseguimi
localhost:~$ ls
eseguimi file.c file.o
localhost:~$
```

- Il parametro **`-lm`** richiede di linkare la libreria "**`m`**" (si tratta della libreria "**`libm.a`**")

Compilazione e linkaggio corretti

```c
#include <math.h> /* contiene sin */

int main(int argc, char** argv) {
   float a = 0.5;
   float b = sin(a);
}
```

```
localhost:~$ ls
file.c
localhost:~$ cc file.c -lm -o eseguimi
localhost:~$ ls
eseguimi file.c
localhost:~$
```


Compilazione ad una sola passata

- La sintassi del linguaggio C prevede che il codice possa essere compilato con una sola passata
- tipi, variabili e funzioni, devono essere dichiarati prima di essere usati
  - attenzione: devono essere **dichiarati**, non necessariamente **definiti**
  - le funzioni che riportano un intero possono essere usate senza essere dichiarate
    - il loro prototipo viene desunto dai parametri
  - molte volte ciò può essere garantito scegliendo un opportuno ordine per le definizioni

Dichiarazione e definizione

- Dichiarazione di variabile o funzione
  - è un costrutto che annuncia al compilatore che la variabile o la funzione potrà essere utilizzata
  - non introduce codice nel file oggetto e può essere iterata
    - purché sia coerente con le precedenti
    - la compilazione di un file con sole dichiarazioni produce un file oggetto vuoto (con la sola intestazione)
- Definizione di variabile o funzione
  - è un costrutto che descrive in dettaglio come è composta la variabile o la funzione che viene definita
  - una definizione è anche implicitamente una dichiarazione
  - introduce codice nel file oggetto e non può essere iterata

Esempi di dichiarazioni

- Dichiarazione di una funzione

```
float funzione(float parametro);
```

analoga a

```
extern float funzione(float parametro);
```

- Dichiarazione di una variabile

```
extern int variabile_globale;
```

  - in questo caso la parola chiave **`extern`** è necessaria, altrimenti il compilatore equivoca la dichiarazione per una definizione

Dichiarazione e definizione di tipi

- Dichiarazioni di tipi
  - sono costrutti che annunciano al compilatore ciò che potrà essere utilizzato
  - non introducono codice nel file oggetto e possono essere iterate
    - purché siano coerenti con le precedenti
- Definizioni di tipi
  - sono costrutti che descrivono in dettaglio come sono composti i tipi che vengono definiti
  - le definizioni di tipi non introducono codice nel file oggetto e possono essere iterate
    - purché siano coerenti con le precedenti

Esempio di ordine di definizione errato

```
int main(int argc, char** argv) {
  float a = doppio(2.5);
}
float doppio(float v) {
  return v * 2.0;
}
```

funzione ancora non dichiarata

```
localhost:~$ cc -c file.c
file.c:4:7: error: conflicting types for ‘doppio’
 float doppio(float v) {
       ^
file.c:2:13: note: previous implicit declaration
of ‘doppio’ was here
   float a = doppio(2.5);
             ^
localhost:~$
```


Ordine di definizione corretto

```
float doppio(float v) {
  return v * 2.0;
}

int main(int argc, char** argv) {
  float a = doppio(2.5);
}
```

```
localhost:~$ ls
file.c
localhost:~$ cc -c file.c
localhost:~$ ls
file.c file.o
localhost:~$
```


Alternativamente: uso di una dichiarazione

```
float doppio(float v);

int main(int argc, char** argv) {
  float a = doppio(2.5);
}
float doppio(float v) {
  return v * 2.0;
}
```

dichiarazione

definizione

```
localhost:~$ ls
file.c
localhost:~$ cc -c file.c
localhost:~$ ls
file.c file.o
localhost:~$
```


Uso obbligatorio di una dichiarazione

```
float funzione1(float v) {
  if ( v > 10 ) return funzione2(v)-1;
  return v;
}

float funzione2(float v) {
  if ( v > 10 ) return 0.5*funzione1(v);
  return v;
}
```

- Non esiste un ordine di definizione delle due funzioni che metta al riparo da un errore di compilazione

Uso obbligatorio di una dichiarazione

```c
float funzione2(float v);

float funzione1(float v) {
  if ( v > 10 ) return funzione2(v)-1;
  return v;
}

float funzione2(float v) {
  if ( v > 10 ) return 0.5*funzione1(v);
  return v;
}
```

dichiarazione

Dichiarazione di un tipo

- Poiché la definizione può essere iterata è più raro trovarsi nella necessità di dichiarare un tipo
- In alcuni casi, però, ciò è indispensabile

```
typedef struct str1 struttura1;
typedef struct str2 struttura2;

typedef struct str1 {
  struttura2* puntatore;
} struttura1;

typedef struct str2 {
  struttura1* puntatore;
} struttura2;
```

forward declarations

Localizzazione delle dichiarazioni

- Generalmente, tutte le dichiarazioni vengono messe in un file .h che può essere importato da altri file .c

```
#ifndef _NOME_DEL_FILE
#define _NOME_DEL_FILE

/* dichiarazioni varie */

#endif
```

- Le variabili e le funzioni dichiarate nel file .h devono essere contenute in un file .o (quindi definite nel corrispondente file .c) perché la fase di linking vada a buon fine

Compilazione secondo lo standard ANSI

- Lo standard ANSI C originale (X3.159-1989) fu ratificato nel 1989 e pubblicato nel 1990
  - per questo motivo è spesso chiamato C89 o C90
- Se si vuole che il compilatore si attenga strettamente allo standard ANSI originale occorre compilare con l'opzione **`-pedantic-errors`**

```
localhost:~$ cc file.c -pedantic-errors -o file
```


Compilazione secondo lo standard ANSI

```
int main(int argc, char** argv) {
   int a = 10;   // commento
   int b = a;
}
```

```
localhost:~$ cc file.c -pedantic-errors -o file
file.c: In function 'main':
file.c:3:16: error: C++ style comments are not
allowed in ISO C90
   int a = 10;   // commento
                 ^
file.c:3:16: error: (this will be reported only
once per input file)
localhost:~$
```


Compilazione secondo lo standard ANSI

```
int main(int argc, char** argv) {
   int a = 10, b = 5, c;
   c = a + b;
   char ch;
}
```

```
localhost:~$ cc file.c -pedantic-errors -o file
file.c: In function ‘main’:
file.c:5:3: error: ISO C90 forbids mixed
declarations and code [-Wpedantic]
   char ch;
   ^
localhost:~$
```


Esercizi

1. Scrivi una funzione in linguaggio C che verifichi se in un array di interi ci sia almeno un intero ripetuto due volte
2. Scrivi una funzione in linguaggio C che verifichi se in un array di interi ci sia almeno un intero ripetuto tre volte

