Architettura dei Calcolatori e Sistemi Operativi
Programmazione Strutturata

Chair

Politecnico di Milano

Prof. C. Brandolese

e-mail: carlo.brandolese@polimi.it
phone: +39 02 2399 3492
web: home.dei.polimi.it/brandole

Teaching Assistant

D. Iezzi

e-mail: domenico.iezzi [at] polimi [dot] it
material: github.com/NoMore201/polimi_cr_acso_2019

Outline

- Direttive di preprocessore
  - Header file
- File ELF
- Layout di un programma C
- Librerie statiche
- Librerie shared
- Librerie dinamiche
- GNU make

Direttive di preprocessore

- **direttive di inclusione: `#include`**
  - servono a includere l'intero testo dell'header file passato come parametro alla direttiva #include.
  - `#include <header.h>` il preprocessore cerca il file header.h nelle directory predefinite dal compilatore (per sistemi GNU/Linux le directory di sistema e.g. `/usr/include`)
  - `#include "header.h"` il preprocessore da precedenza alla directory corrente, se il file non viene trovato allora cercherà tra le directory di sistema.
- **direttive di macro-definition: `#define`**
  - servono a definire macro e costanti: attenzione perché il preprocessore sostituisce nel codice il valore di queste costanti al loro simbolo (come discusso a lezione).
- **direttive condizionali: `#if #ifdef #ifndef #else #elif #endif`**
  - utilizzate principalmente per la compilazione condizionale, che permette di includere/escludere codice a seconda che determinate condizioni siano verificate o meno (ad es. il tipo di sistema operativo,versioni di libreria, hardware…)

Direttive di preprocessore

Nei progetti C di grosse dimensioni è spesso necessario articolare il codice sorgente in più file. Questo richiede l'utilizzo degli header file

- Ha estensione "`file.h`" e viene utilizzato con il comando `#include` secondo le regole viste nella slide precedente
- Contiene i prototipi delle funzioni e la dichiarazione di strutture e variabili, implementate poi in uno o più `file.c` ad esso associati
- Utili per semplificare l'utilizzo delle librerie, in quanto il programmatore non dovrà dichiarare nel proprio programma tutti i prototipi delle funzioni, le strutture e le variabili di libreria usate, ma dovrà semplicemente includere l'header file della libreria
- Forniscono una documentazione base dei componenti di un programma o libreria
- Il preprocessore C si occuperà di sostituire la direttiva `#include` con il contenuto dell'header specificato, generando un nuovo file temporaneo con estensione `file.i`

Direttive di preprocessore – header file

- Gli include guard servono ad evitare le *double inclusion*, ovvero che una stessa funzione, struttura o variabile venga inclusa più volte, generando confusione per il compilatore.

- Esempio:

my_lib1.h

```
int val = 3;
```

my_lib2.h

```
#include "my_lib1.h"

int val;
```

main.c

```
#include "my_lib1.h"
#include "my_lib2.h"


int main(int argc, char **argv)
{
    …
    return 0;
}
```

Direttive di preprocessore – header file

Output del comando `$ cpp main.c`

Definizioni multiple

main.i

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "first.h" 1
int val = 3;
# 2 "main.c" 2
# 1 "second.h" 1
# 1 "first.h" 1
int val = 3;
# 2 "second.h" 2

int val;
# 3 "main.c" 2

int main(int argc, char **argv)
{
    return 0;
}
```

Direttive di preprocessore – header file

Come evitare il problema: utilizzo delle include guards

Le include guards sfruttano una direttiva condizionale `ifndef … endif` e una `define`. Il nostro esempio diventa:

my_lib1.h

```
#ifndef MY_LIB1_H
#define MY_LIB1_H

int val = 3;

#endif
```

my_lib2.h

```
#ifndef MY_LIB2_H
#define MY_LIB2_H

#include "my_lib1.h"

int val;

#endif
```

main.i

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "first.h" 1
int val = 3;
# 2 "main.c" 2
# 1 "second.h" 1
int val;
# 3 "main.c" 2

int main(int argc, char **argv)
{
    return 0;
}
```

File ELF

- **Dalla compilazione otteniamo del codice assembly (.s).**
  - Questo tipo di file è testuale
  - Analizzeremo il codice assembly successivamente

```
$ gcc -S hello.i
$ cat hello.s
```

  - Nelle fase di assembling viene generato un file oggetto (.o)
  - L'insieme dei file in input che hanno generato un object file è chiamato **translation unit**
  - La fase di linking genera il vero e proprio file eseguibile

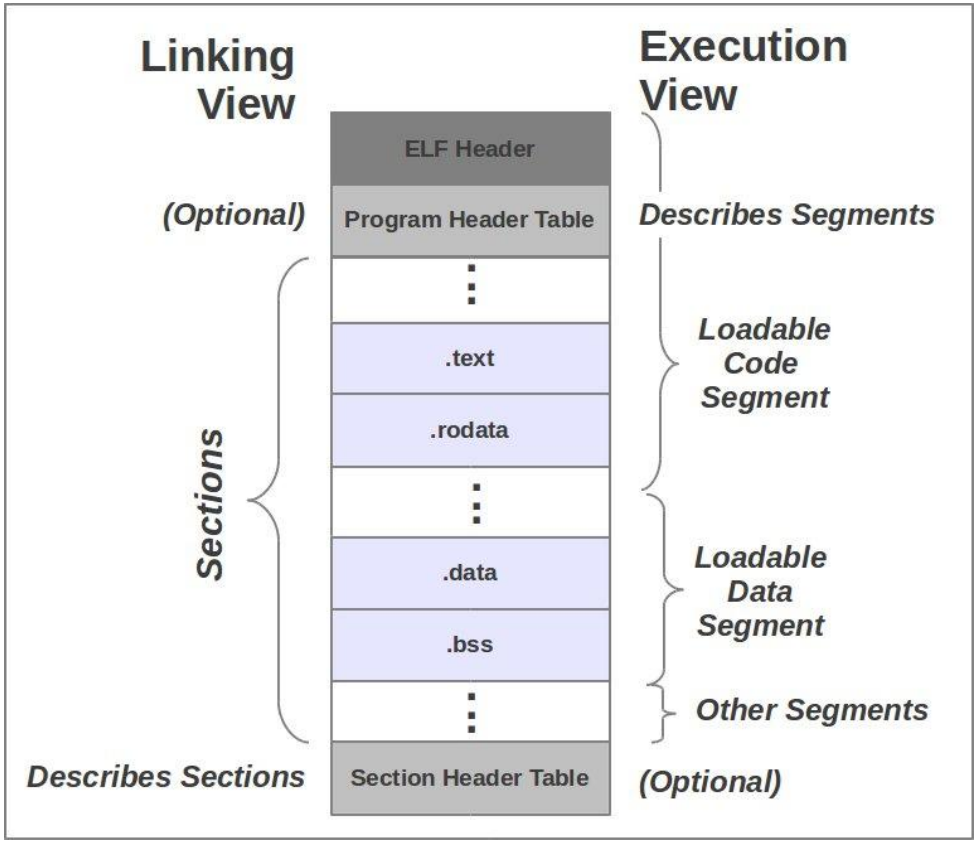
```
$ gcc -c hello.s
$ gcc -o hello hello.o file1.o
```

- **I file ELF (Executable and Linkable File) sono uno standard per immagazzinare programmi o parti di un programma in un file su disco.**
  - Eseguibili
  - Object file
  - Librerie condivise

File ELF

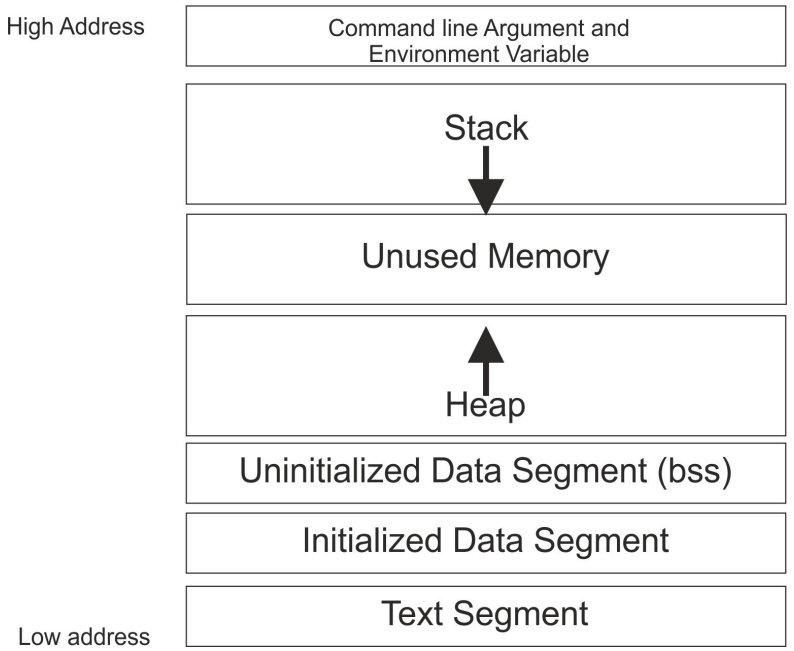
- **Un file ELF ha 3 sezioni principali:**
  - **ELF header:** descrive il file in generale e punta alle due tabelle
  - **Program headers (Segments) :** informazioni su come creare l'immagine di un processo (*runtime*)
  - **Section headers (Sections):** contiene informazioni che descrivono le sezioni del file (*linking*)

- **Ogni segmento può contenere più sezioni**
  - Il file conterrà informazioni riguardanti il mapping tra sezioni e segmenti
  - È possibile ottenere informazioni su un object file o un eseguibile in formato ELF tramite il comando `readelf`



Layout in memoria di un programma C

- Le sezioni principali di un programma sono:
  - **Sezione Text** (*.text*) – contiene le istruzioni eseguibili
  - **Sezione dati inizializzati** (*.data*) – contiene variabili globali e statiche inizializzate dal programmatore
  - **Sezione dati non inizializzati** (*.bss*) – contiene variabili globali e statiche che non sono state inizializzate
  - **Stack** – contiene variabili locali di una funzione
  - **Heap** – contiene dati o strutture allocati dinamicamente



Simboli

- **I simboli sono utilizzati dal compilatore per associare ad ogni funzione o variabile informazioni riguardo la definizione**
- **Il programma `nm` elenca tutti simboli presenti in un file oggetto. Per ogni simbolo mostra:**
  - Il valore del simbolo in esadecimale di default
  - Il tipo del simbolo. Se il tipo è minuscolo, allora è locale; se è maiuscolo, allora è globale (extern). I tipi più comuni sono:
    - B/b The symbol is in the uninitialized data section (known as BSS).
    - D/d The symbol is in the initialized data section.
    - U The symbols is undefined .
    - R/r The symbol is in a read only data section.
    - T/t The symbol is in the text (code) section.
  - Il nome del simbolo
- **È possibile anche utilizzare il comando `objdump -t file.o`**

Librerie statiche

- Insieme di routine, funzioni e variabili precompilato in un file con estensione `libsample.a` , chiamato anche *archive file*
- Durante la fase di linking, il contenuto della libreria viene copiato interamente all'interno dell'eseguibile generato
- Il codice della libreria viene quindi incluso nell'eseguibile, e non risulterà necessario fornire la libreria precompilata separatamente

Per generare un *archive file:*

```
$ gcc -c lib.c
$ ar rcs libsample.a lib.o
```

Librerie shared

- Insieme di routine, funzioni e variabili precompilato in un file con estensione `libsample.so`
- Il codice non verrà copiato nell'eseguibile, bensì verrà caricato durante lo startup del programma
- In questo caso il linker crea una nuova sezione con informazioni relative alla libreria shared che dovrà essere utilizzata a runtime

Per generare una *shared library*:

```
$ gcc -shared -o libsample.so lib.c
```

Librerie dinamiche

- Insieme di API utilizzate per caricare una libreria durante l'esecuzione di un programma
- La libreria non viene caricata durante la fase di linking, e nemmeno durante lo startup
- Dal punto di vista del file, sono delle normali librerie shared, ma possiamo caricarle dinamicamente utilizzando la funzione `dlopen(...)`

Per caricare dinamicamente una libreria:

```
#include <dlfcn.h>

int main() {
    void *handle;
    ...
    handle = dlopen("/path/to/libsample.so", RTLD_LAZY);
    ...
}
```

Makefile

GNU Make

- Determina automaticamente quali parti di un programma complesso devono essere ricompilate
- Esegue i comandi utili alla loro ricompilazione
- Utilizza dei *Makefile* nei quali sono specificate le relazioni tra i file all'interno di un progetto, e i comandi da utilizzare per compilare una certa classe di file (le cosiddette **Rules**)

Le Rules sono composte da

- *Target*: file che vogliamo generare tramite la compilazione
- *Recipe*: istruzioni per come compilare / ricompilare il *target*
- *Prerequisites*: indicano quali *target* devono essere già compilati per poter procedere

Lanciare il comando *make*

```
make [options] [target]
```

`-C dir`	Esegue il comando `cd dir` prima di iniziare
`-f file`	Specifica un makefile diverso da quelli di default
`-j [n]`	Esegue `n` job in parallelo
`-n`	Stampa i comandi richiesti per compilare il target senza eseguirli

Rules

Tipologie di rules

- *Esplicite* specificano come aggiornare un file specifico

```
main.o: main.c
    gcc –c main.c –o main.o
```

- *Implicite* specificano come aggiornare una classe di file

```
%.o: %.c
    gcc –c $< –o $@
```

Questa regola verrà automaticamente espansa da make in:

```
main.o: main.c
    gcc -c main.c -o main.o

libreria.o: libreria.c
    gcc -c libreria.c -o libreria.o
```

- **$<** è una variabile automatica che contiene il prerequisito del target corrente, in caso il target sia implicito
- **$@** contiene invece il nome del target corrente

Rules

Rules speciali

- I *target* in una rule possono non essere dei nomi di file, in tal caso vengono chiamati *phony targets*. Tipici esempi sono:

```
clean:
    rm –f *.o <program_name>
```

- Il target *clean* è utilizzato per pulire la directory da tutti i file generati dalle precedenti compilazioni

```
all: <program_name>

<program_name>: main.o libreria.o
    gcc -o $@ $^
```

- Il target *all* è utilizzato per effettuare tutti gli step necessari per la compilazione.
- È il target che viene chiamato di default quando non specificato tramite il comando `make`