Architettura dei Calcolatori e Sistemi Operativi
Programmazione C in Linux

Chair

Prof. C. Brandolese

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

Politecnico di Milano

Teaching Assistant

A. Canidio

e-mail: andrea.canidio@mail.polimi.it

Outline

- **Struttura del codice**
  - Direttive di preprocessore
    - Header file
  - Layout di un programma C
  - Librerie statiche
  - Librerie dinamiche
  - Makefile
  - Comando *nm*

Direttive di preprocessore

Il preprocessore C viene invocato dal compilatore e agisce a monte di questo per gestire tre direttive principali (ve ne sono altre minori):

- direttive di inclusione: **#include**

  → *servono a includere l'intero testo dell'header file passato come parametro alla direttiva #include. In questo modo viene semplificata l'importazione di funzioni e variabili di librerie e altre componenti sorgenti del programma.*

  → *se l'header file viene passato con doppi apici* (es. `#include "header.h"`) *il preprocessore controlla solo la directory corrente, se l'header file viene passato tra virgolette <>* (es. `#include <header.h>`) *il preprocessore controlla i path alle include directories standard del compilatore.*

- 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 and #endif**

  → *argomento un po' più avanzato: utilizzate principalmente per la compilazione condizionale, che permette di generare eseguibili diversi a seconda che determinate condizioni siano verificate o meno (ad es. il tipo di sistema opertivo,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

- Un header file ha estensione "`file.h`" e viene utilizzato con il comando #include secondo le regole viste nella slide precedente
- Un header file contiene i prototipi delle funzioni e la dichiarazione di strutture e variabili, implementate poi in uno o più `file.c` ad esso associati
- Sono estremamente 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
- Il preprocessore C si occuperà poi, una volta incluso l'header file, di sostituire a questo i prototipi delle funzioni e la dichiarazione delle strutture e delle variabili in esso dichiarate, in maniera trasparente al programmatore. Per averne un esempio digitare il comando:

  ```
  cpp file.c
  ```

  che restituisce l'output del preprocessore e da prova di quanto appena affermato.
- Il preprocessore prende in ingresso un file di estensione *.c e ne restituisce uno di estensione *.i

Direttive di preprocessore – header file

Nella scrittura di un header file occorre avere l'accortezza di utilizzare gli include guards

- 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. Vediamo un esempio di doppia inclusione e come è possibile risolverlo:

Esempio di doppia inclusione

```
// file my_lib1.h

struct pippo{
    int pluto;
    int paperino;
};
```

```
// file my_lib2.h che include my_lib1.h

#include "my_lib1.h"

int foo (struct pippo p);
```

```
// main

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

int main(int argc, char** argv){
    ...
}
```

Direttive di preprocessore – header file

- **Cosa è successo:**
  - `my_lib1.h` definisce la `struct pippo`
  - `my_lib2.h` include `my_lib1.h`
  - Il preprocessore, come abbiamo detto , "sostituisce" alla dichiarazione

    `#include "my_lib1.h"`

    tutto il testo contenuto in `my_lib1.h` (ovvero tutte le dichiarazioni di funzioni, struct, variabili, etc) e dunque, in questo caso, la dichiarazione della `struct pippo`
  - In `main.c` vengono incluse sia `my_lib1.h` che `my_lib2.h`, quindi il preprocessore "sostituirà" a tali inclusioni tutte le dichiarazioni contenute in esse, ovvero:
    - Tutte le dichiarazioni contenute in `my_lib1.h` e quindi, in questo caso la `struct pippo`
    - Tutte le dichiarazioni contenute in `my_lib2.h`. Poiché, come detto, `my_lib2.h` include a sua volta `my_lib1.h`, il preprocessore "sostituisce " a tale inclusione tutto il testo contenuto in `my_lib1.h`, quindi la `struct pippo`. Poi include le dichiarazioni contenute in my_lib2.h, ovvero la dichiarazione della funzione foo.
    - **Morale della favola**: in main.c, dopo la fase di preprocessing, la struct pippo viene definita due volte, generando confusione per il compilatore.

Output di cpp main.c

```
acanidio@lr:~/Desktop/Ese-1/header_files/Hdr-1$
cpp main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"
# 1 "my_lib1.h" 1

struct pippo{
    int pluto;
    int paperino;
};
# 2 "main.c" 2
# 1 "my_lib2.h" 1
# 1 "my_lib1.h" 1

struct pippo{
    int pluto;
    int paperino;
};
# 2 "my_lib2.h" 2

int foo(struct pippo p);
# 3 "main.c" 2

int main(){

 return 0;
}
acanidio@lr:~/Desktop/Ese-1/header_files/Hdr-1$
```

Direttive di preprocessore – header file

Come evitare il problema: utilizzo delle include guards

Di fatto le include guards sono costituite da tre direttive, all'iterno delle quali racchiudere il codice dell'header file:

```
// include guards
#ifndef NOME_LIBRERIA_H
#define NOME_LIBRERIA_H
... //codice libreria
#endf
```

Il nostro esempio dunque diventa

```
// file my_lib1.h
#ifndef MY_LIB1_H
#define MY_LIB1_H

struct pippo {
    int pluto;
    int paperino;
};

#endif
```

```
// file my_lib2.h che include my_lib1.h
#ifndef MY_LIB2_H
#define MY_LIB2_H

#include "my_lib1.h"

int foo (struct pippo p);

#endf
```

Direttive di preprocessore – header file

Cosa cambia:

- Al momento del preprocessing del file main.c:

```
// main

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

int main(int argc, char** argv){
    ...
}
```

1. Il preprocessore dapprima incontra la direttiva `#include "my_lib1.h"`
   - Leggendo tale file trova il comando `#ifndef MY_LIB1_H` , che dice qualcosa del genere: "se `MY_LIB1_H` non è ancora stato definito, includi il codice sottostante , altrimenti salta a `#endif`". Poiché non è ancora stato definito, passerà a includere il codice sottostante e quindi anche `#define MY_LIB1_H` e poi a "includere" tutte le dichiarazioni contenute in my_lib1.h, ovvero la `struct pippo`.
2. Successivamente il preprocessore incontra la direttiva `#include "my_lib2.h"`.
   - Leggendo tale file trova il comando `#ifndef MY_LIB2_H` , che dice qualcosa del genere: "se `MY_LIB2_H` non è ancora stato definito, includi il codice sottostante, altrimenti salta a `#endif`". Poiché non è ancora stato definito, passerà a definirlo con `#define MY_LIB2_H`. Passa poi a processare `#include "my_lib1.h"`, presente subito dopo. Saltando in my_lib1.h, questa volta però la condizione `#ifndef MY_LIB1_H` risulta essere falsa, in quanto `MY_LIB1_H` è già stata definita in precedenza (punto 1.). Salta dunque direttamente ad `#endif` di my_lib1.h, evitando di inserire nuovamente il testo contenuto in esso.
   - Ritorna dunque a my_lib2.h e continua il preprocessing includendo la dichiarazione di `foo`.

- **L'effetto netto è quello di evitare di "inserire" due volte le dichiarazioni di my_lib1.h in main.i**

File ELF

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

```
gcc -S hello.i
vim hello.s
```

- **Infine**
  - Nelle fase di assembling viene generato un file oggetto (.o)
  - E nella fase di linking il vero e proprio file eseguibile (.exe in Windows)

```
vim hello.o
vim hello
```

- **Entrambi sono file ELF**

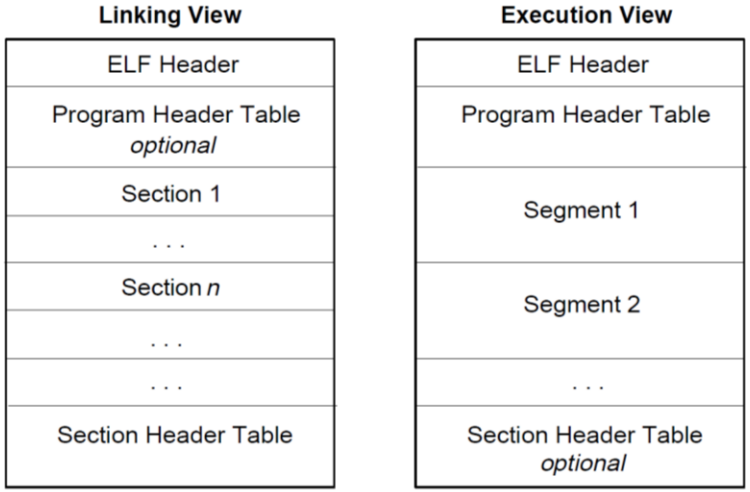
```
file hello.o
file hello
```

- **I file ELF (Executable and Linkable File) sono uno standard di file utilizzato per rappresentare:**
  - Eseguibili
  - Codice oggetto
  - Librerie condivise

File ELF

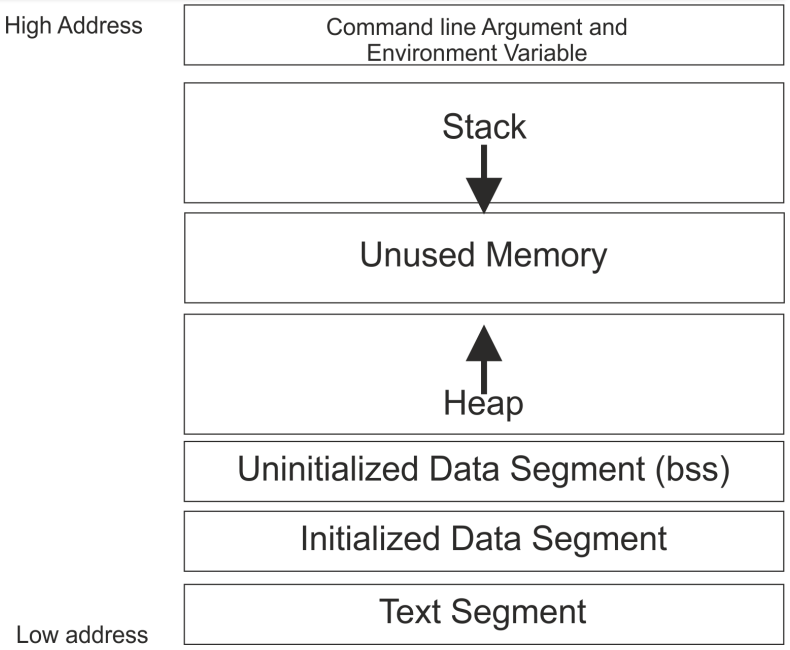
- **Un file ELF ha 3 sezioni principali:**
  - *ELF header* descrive il file in generale e punta alle due tabelle
  - *Program header table* descrive al sistema come creare un processo (*runtime*)
  - *Section header table* contiene informazioni che descrivono le sezioni del file (*linking*)
- **Leggendo l'ELF header**

```
$ readelf --header /bin/ls
ELF Header:
  Magic: 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, big endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: PowerPC
  Version: 0x1
  Entry point address: 0x10002640
  Start of program headers: 52 (bytes into file)
  Start of section headers: 87460 (bytes into file)
  Flags: 0x0 Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 8
  Size of section headers: 40 (bytes)
  Number of section headers: 29
  Section header string table index: 28
[...]
```



Layout di un programma C

- Quindi una tipica rappresentazione in memoria di un programma C è data da varie sezioni, ma le principali 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 informazioni salvate ogni volta che una funzione viene chiamata
  - **Heap** – segmento dove l'allocazione dinamica ha luogo



Makefile

GNU Make

- Determina automaticamente quali parti di un programma complesso devono essere ricompilate
- Esegue i comandi utili alla loro ricompilazione
- È lo stumento standard usato da chi sviluppa per Linux (chi sviluppa per Windows usa gli IDE)

I Makefile definiscono

- *Target*: file che vogliamo compilare / ricompilare
- *Goals*: istruzioni per come compilare / ricompilare i *target*
- *Dependencies*: indicano quali *target* devono essere ricompilati a seguito di una modifica

Lanciare il comando *make*

```
make [options] [goal...]
```

`-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, Se n è omesso, esegue quanti job possibili
`-n`	Stampa i comandi richiesti per aggiornare il goal senza eseguirli

Goals

Tipologie di goals

- *Espliciti* specificano come aggiornare un file specifico

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

- *Impliciti* specificano come aggiornare una classe di file

```
%.o: %.c
    gcc –c $<.c –o $@.o
```

Goals speciali

- *clean* regola che rimuove tutti gli object files e programmi, per cominciare con una compilazione pulita

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

- *all* regola eseguita quando viene invocato il comando make

Comando *nm*

- **GNU nm elenca tutti simboli presenti nel file oggetto objfile… Se non c'è nessun file come argomento, nm userà a.out di default. Per ogni simbolo nm 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.
    - P The symbols is in a stack unwind section.
    - 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