

Architettura dei Calcolatori e Sistemi Operativi

Esercitazione 1

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

A. Canidio

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

Outline

- **Programmazione C in Linux**
 - Comando *man*
 - Compilatore *gcc*
 - Parametri *argv* e *envp*
 - Classi di memorizzazione
 - Puntatori
 - *Struct* e *union*

Comando *man*

- Man è il manuale di sistema per i sistemi Linux.

```
man [<section>] <page>
```

- Ogni argomento *page* è il nome di un programma, una utility o una funzione disponibile nel sistema. Eventualmente aggiungendo l'argomento *section* si può specificare la sezione del manuale che si vuole accedere:
 - Executable programs or shell commands
 - System calls (functions provided by the kernel)
 - Library calls (functions within program libraries)
 - Special files (usually found in /dev)
 - File formats and conventions eg /etc/passwd
 - Games
 - Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
 - System administration commands (usually only for root)
 - Kernel routines [Non standard]
- **Attenzione! *man* è vostro amico, sfruttatelo.**

Compilatore *gcc*

- Compilatore in grado di trasformare il codice sorgente C in codice macchina

```
gcc [options] <filename>
```

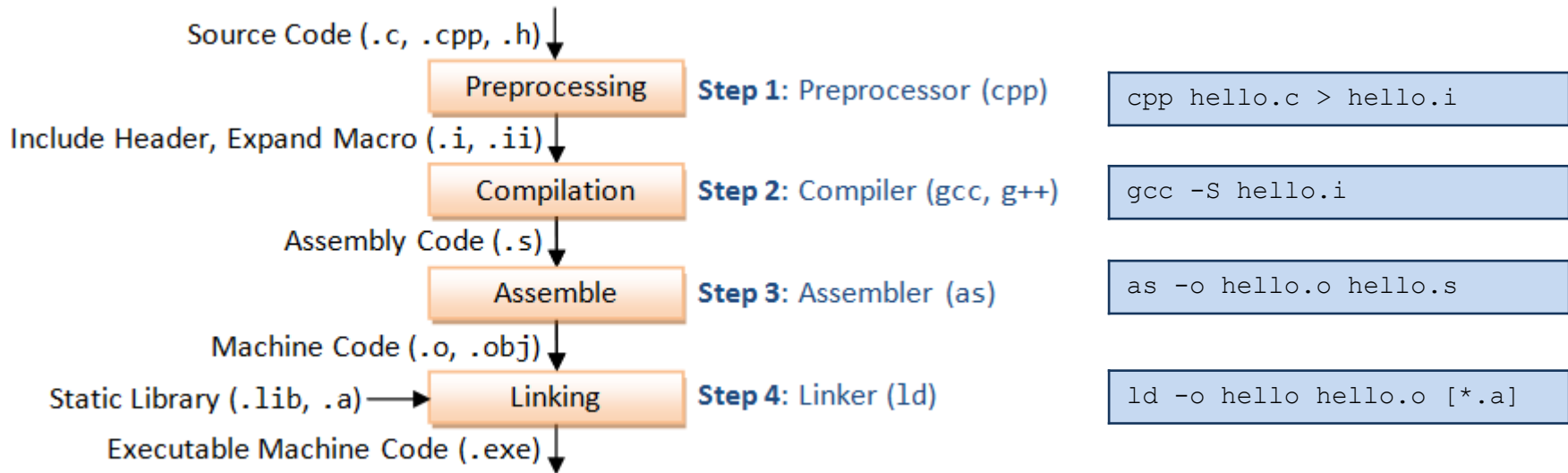
- Tra le opzioni più importanti troviamo:
 - `-o outputfile` specifica il nome del file di output
 - `-Wall` attiva tutti i warning
 - `-g` genera simboli aggiuntivi per *gdb*
 - `-v` attiva la modalità *verbose*
 - `-lm` linking libreria *math.h*
 - `-S` genera i file *assembly*
 - `-D name` Definisce *name* come macro, con definizione 1.

Fasi della compilazione *gcc*

Come si passa da un codice sorgente C ad un programma eseguibile?

```
gcc hello.c -o hello
```

- Il compilatore GCC compie questo processo in 4 passi successivi



- L'alternativa è chiedere al GCC di salvare i file intermedi prodotti durante la compilazione

```
gcc -save-temps hello.c -o hello
```

Preprocessore e compilazione condizionale

- Il preprocessore legge un sorgente C e produce in output un altro sorgente C, dopo avere espanso in linea le macro, incluso i file e valutato le compilazioni condizionali o eseguito altre direttive.
- Il preprocessore agisce principalmente sulle keyword
 - #include
 - #define
 - ...
- Esistono direttive del preprocessore che consentono la compilazione condizionata, vale a dire la compilazione di parte del codice sorgente solo sotto certe condizioni. Questo è possibile attraverso le keyword
 - #if, #ifdef, #ifndef
 - #else, #elif
 - #endif

Parametri *argv* e *envp*

Dichiarazioni possibili

- `int main(int argc, char *argv[])`
- `int main(int argc, char *argv[], char *envp[])`
- `int main(void)`

Significato dei parametri

- `argc` numero degli argomenti
- `argv` vettore di puntatori a char che contiene la lista dei parametri passati al main
 - `argv[0]` restituisce sempre il nome del programma;
- `envp` restituisce le variabili d'ambiente

Ritorno dal main

- Il main ritorna di default con `return 0`, se non viene specificato altro dal programmatore.
- Tradizionalmente lo standard C prevede solo due possibili stati di uscita dal main:
 - `return 0;` `EXIT_SUCCESS` - indica che il programma ha avuto successo
 - `return x;` **con $x \neq 0$** `EXIT_FAILURE` - in pratica , il significato dei valori di ritorno diversi da zero può essere gestito dal programmatore

Classi di memorizzazione

- Definiscono le regole di visibilità delle variabili e delle funzioni quando il programma è diviso su più file.
- Variabili e funzioni hanno un attributo che specifica una tra 4 classi di memorizzazione possibili.
- Le classi di memorizzazione in C possono essere:
 - *auto* solo per variabili
 - *static* allocazione di memoria e visibilità
 - *extern* per variabili e funzioni
 - *register* solo per variabili

Classe *auto*

- E' quella usuale per le variabili locali. Lo spazio per variabili automatiche viene riservato all'interno del record di attivazione della funzione e rilasciato quando questa termina. La parola riservata che specifica tale attributo è `auto`
- Di default tutte le variabili locali sono automatiche

```
void f(void)
```

```
{
```

```
    int tmp; equivale a
```

```
    ...
```

```
}
```

```
void f(void)
```

```
{
```

```
    auto int tmp;
```

```
    ...
```

```
}
```

Classe *static* - memoria

- Una variabile locale statica è una variabile di una funzione che vede associato uno spazio per tutto il tempo che il programma è in esecuzione. Una variabile statica conserva il proprio valore (anche se inaccessibile) tra una chiamata e l'altra della funzione in cui è definita. La parola riservata che specifica tale attributo è `static`

- **Esempio:** questa funzione stampa il numero di volte che è stata chiamata

```
void f(void)
{
    static int count = 0;
    ...
    printf("%d", ++count);
}
```

Classe *static* - visibilità

- Un secondo uso della parola riservata `static` riguarda la possibilità di limitare la visibilità di variabili globali o funzioni. Una variabile globale o una funzione con attributo di memorizzazione `static` sono visibili esclusivamente nel file d'appartenenza a partire dal punto in cui sono dichiarate.

Esempio: file1.c

```
void f(void)
{
    ... /* qui s non è disponibile */
}
static int s; /* variabile globale statica */
void g(void)
{
    ... /* qui s è disponibile */
}
```

file 2.c

```
extern int s; /* errore: s non è disponibile */
void g(void)
{
    s = 2; /* errore */
}
```

Classe *extern*

- L'uso dell'attributo esterno riferito a variabili locali rappresenta il modo che una funzione adotta per accedere a variabili globali definite in altri file. Una variabile locale esterna non è quindi memorizzata nel record di attivazione della funzione. La parola riservata che specifica tale attributo è `extern`
- L'attributo `extern` utilizzato nella definizione di un prototipo di funzione rappresenta un'indicazione data al compilatore che la definizione completa della funzione si trova in un altro file.

Classe *register*

- Una variabile locale con classe di memorizzazione registro è una variabile che viene direttamente associata a un registro del processore. Se ciò non è possibile (numero limitato di registri, tipo non compatibile) il compilatore tratta la variabile come automatica. La parola riservata che specifica tale attributo è `register`
- **Esempio:** un uso tipico di questa classe è per gli indici di ciclo

```
void f(void)
{
    register int i;
    for (i=0; i<SIZE;i++)
    { ... }
}
```

Puntatori

Operatore di referenziazione “Reference” (&) e dereferenziazione (*)

- Utilizzando l’operatore di referenziazione, ovvero &, seguito immediatamente dal nome di una variabile è possibile estrarne l’indirizzo, ovvero referenziarla.
- Utilizzando l’operatore di “dereferenziazione”, ovvero *, seguito immediatamente dall’indirizzo di una variabile è possibile estrarne il valore, ovvero dereferenziarla.

Puntatore

- Un puntatore è una speciale variabile, in grado di contenere l’indirizzo di un’altra variabile
- La dichiarazione di un puntatore avviene antepoendo al nome della variabile l’operatore di dereferenziazione*:

```
int pluto=10;          /* variabile */  
int *pippo=&pluto;     /* puntatore pippo alla variabile pluto */
```

- Il valore NULL viene utilizzato per inizializzare puntatori di qualunque tipo specificando che essi non puntano a nessuna zona di memoria esistente

Fin qui tutto OK...

Puntatori

Puntatori di puntatori

```
int    pippo=1;
int*   pluto=&pippo;
int**  paperino=&pluto;
```

Come accedere al valore di pippo per mezzo di paperino?

→ I puntatori di puntatori presentano multipli stati di reindirizione: in questo caso pluto contiene l'indirizzo di pippo e paperino l'indirizzo di pluto, pertanto per leggere il valore di pippo dovrò dereferenziare due volte paperino.

Puntatori a void

- Possono essere utilizzati per puntare a dati di qualunque tipo
- Sono indeterminati in potere di dereferenziazione e lunghezza del blocco dei dati punti
- A patto che...venga effettuato il cast

```
int foo(void* data){
    int *prova; /* se già sappiamo di voler recuperare un intero*/
    prova=(int*) data;
}
```

Type casting

- È buona norma effettuare il cast da un sottotipo a un supertipo (es. da int a float, pericoloso e da usare con estrema cautela il viceversa), per evitare perdita di informazione non voluta.

Puntatori

Puntatori a funzioni

- Si dichiarano come i prototipi delle funzioni, con l'accortezza di includere il nome della funzione tra parentesi e far precedere allo stesso l'operatore di dereferenziazione

Esempio

```
#include <stdio.h>
void pippo(int i){
    printf("Valore: %d\n",i);
}

void (*foo) (int);

int main(){
    int i=10,j=5;
    pippo(i);
    foo=pippo;
    foo(j);
    (*foo)(j);
}
```

Quale delle due chiamate a foo è quella corretta? Cosa stampa foo?

→ Entrambe, una volta assegnato al puntatore di funzione l'indirizzo della funzione che si vuole chiamare non fa differenza il fatto di dereferenziare o meno il puntatore a funzione al momento della chiamata. `foo(j)` stampa 5.

Struct, union e typedef

- **Struct** Le *struct* del C sostanzialmente permettono l'aggregazione di più variabili, in modo simile a quella degli array, ma a differenza di questi non ordinata e non omogenea (una struttura può contenere variabili di tipo diverso).

```
struct <name> {  
    field1;  
    field2;  
    ...  
}
```

- **Union** Il tipo di dato *union* serve per memorizzare (in istanti diversi) oggetti di differenti dimensioni e tipo, con, in comune, il ruolo all'interno del programma

```
union {  
    field1;  
    field2;  
    ...  
}
```

- **Typedef** Per definire nuovi tipi di dato viene utilizzata la funzione typedef

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 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

- 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'interno delle quali racchiudere il codice dell'header file:

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

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);

#endif
```

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`**

Makefile

GNU Make

- Determina automaticamente quali parti di un programma complesso devono essere ricompilate
- Esegue i comandi utili alla loro ricompilazione
- È lo strumento 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...]
```

- | | |
|----------------------|---|
| <code>-C dir</code> | Esegue il comando <code>cd dir</code> prima di iniziare |
| <code>-f file</code> | Specifica un makefile diverso da quelli di default |
| <code>-j [n]</code> | Esegue <code>n</code> job in parallelo, Se <code>n</code> è omesso, esegue quanti job possibili |
| <code>-n</code> | 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