

Architettura dei Calcolatori e Sistemi Operativi

Programmazione C

Chair

Prof. C. Brandoles

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

Politecnico di Milano

Teaching Assistant

D. Iezzi

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

Outline

■ Programmazione C

- Comando *man*
- Compilatore *gcc*
- Main e parametri *argv, 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:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. 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:

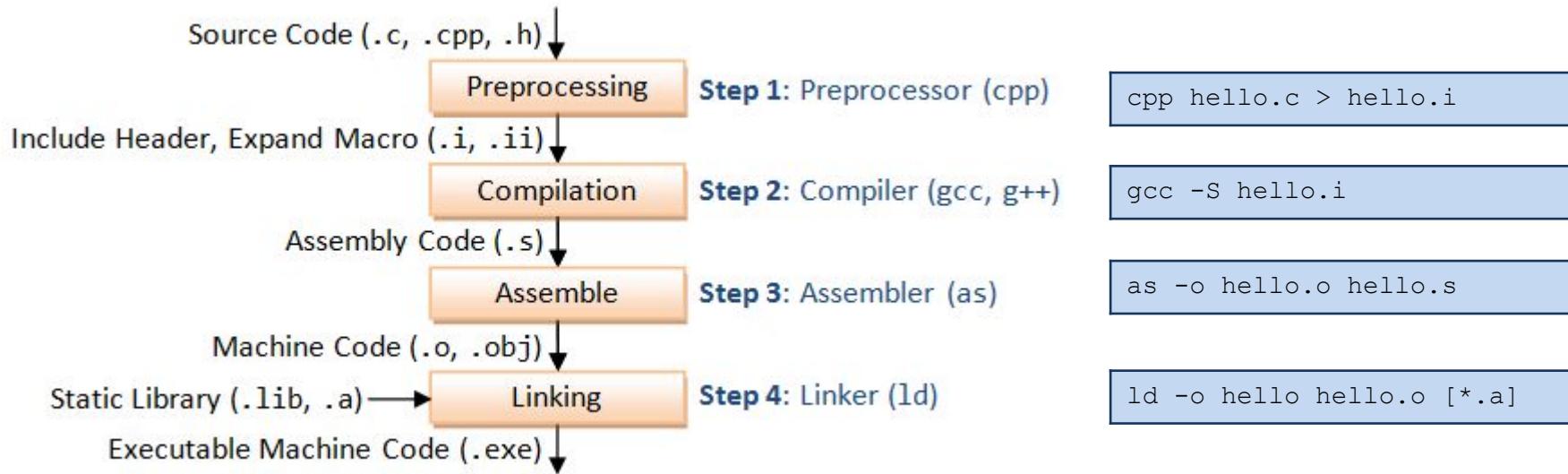
- `-o outputfile` specifica il nome del file di output
- `-Wall` attiva tutti i warning
- `-g` genera simboli aggiuntivi per *gdb*
- `-v` attiva la modalità *verbose*
- `-lm` linking libreria *math.h*
- `-S` genera i file *assembly*
- `-D name` definisce *name* come macro, con definizione 1

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 -fverbose-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

Funzione *main*

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 - il significato del valore può essere gestito dal programmatore

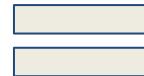
Classi di memorizzazione

- 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;
    ...
}
```



```
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\n", ++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 di appartenenza a partire dal punto in cui sono dichiarate.

file1.c

```
void f(void)
{
    ... /* qui s non è disponibile */
}

/* variabile globale statica */
static int s;

void g(void)
{
    ... /* qui s è disponibile */
}
```

file2.c

```
/* errore: s non è disponibile */
extern int s;

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 anteponendo 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 reindirezione: in questo caso pluto contiene l'indirizzo di pippo e paperino l'indirizzo di pluto, pertanto per leggere il valore di pippo dovrà 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* permette di accedere ad una stessa area di memoria con differenti rappresentazioni

```
union {
    type1 field1;
    type2 field2;
    ...
}
```

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