Corso di Sistemi Operativi
A.A. 2008-2009
-
INTRODUZIONE AL C

Fabio Buttussi

Il linguaggio C

- Il **C** è un linguaggio **imperativo** legato a Unix, adatto all'implementazione di compilatori e sistemi operativi.
- È stato progettato da D. Ritchie per il PDP-11 (all'inizio degli anni '70). Nel 1983 l'**ANSI** ne ha definito una versione standard **portabile** (ANSI C).
- A differenza dei linguaggi da cui ha tratto le idee fondamentali, ovvero, BCPL (M. Richards) e B (K. Thompson), è un linguaggio **tipato**.
- Il C è **compilato**; la compilazione è preceduta da una fase di *preprocessing* (sostituzione di macro, inclusione di file sorgenti ausiliari e compilazione condizionale).
- Il C è considerato un linguaggio ad *alto livello, ma non "troppo"* in quanto fornisce le primitive per manipolare numeri, caratteri ed indirizzi, ma non oggetti composti come liste, stringhe, vettori ecc.
- Il C è un linguaggio "piccolo": non fornisce direttamente nemmeno delle primitive di input/output. Per effettuare queste operazioni si deve ricorrere alla **Libreria Standard**. Si può pensare al C come al nucleo imperativo di Java più i **puntatori** e la gestione a **basso livello** di numeri, caratteri e indirizzi.

Struttura di un programma C

Consideriamo il programma C che stampa a video la stringa `ciao, mondo!` seguita da un avanzamento del cursore all'inizio della linea successiva:

```
#include <stdio.h>

main()
{
    printf("ciao, mondo!\n");
}
```

- Ogni programma C è composto da variabili e funzioni (contenenti comandi); fra queste ne esite una particolare, chiamata `main`, da cui **inizia l'esecuzione** e che quindi deve essere presente in ogni programma. Le due parantesi `()` vuote dopo il `main` significano che quest'ultimo non prende alcun parametro in input.

- La prima riga è una **direttiva al preprocessore** che dice di includere le funzioni per l'input/output della libreria standard prima di compilare il programma.

- La funzione `printf` della libreria standard stampa a video (standard output) la stringa fornita come parametro. All'interno di quest'ultima la **sequenza di escape** `\n` specifica il carattere speciale di "avanzamento all'inizio della linea successiva" o newline.

I tipi base del C

`int`	interi
`float`	floating-point a precisione singola
`char`	caratteri (un singolo byte)
`short` (`short int`)	intero corto
`long` (`long int`)	intero lungo
`double`	floating-point a precisione doppia

In C esistono due tipi di **conversioni di tipo**:

1. **Promozioni**: conversioni automatiche

   `char` $\rightsquigarrow$ `short` $\rightsquigarrow$ `int` $\rightsquigarrow$ `long` $\rightsquigarrow$ `float` $\rightsquigarrow$ `double`

2. **Cast**: conversione esplicita (nel verso opposto); per esempio: `x=(int)5.0;`

Le funzioni in C

I programmi C sono costituiti da **definizioni di variabili** e **funzioni**.

Una definizione di funzione ha il seguente formato:

```
tipo-ritornato nome-funzione(lista-parametri)
{
   dichiarazioni
   istruzioni
}
```

Le definizioni di funzioni possono comparire in qualsiasi ordine all'interno di uno o più **file sorgente**, ma non possono essere spezzate in più file.

Argomenti: chiamata per valore e per riferimento

In C tutti gli argomenti delle funzioni, che **non** sono **vettori** nè **puntatori**, sono passati **per valore**. Cioè le funzioni lavorano su copie dei parametri e non modificano i parametri passati dal chiamante.

Al contrario, argomenti **vettore** o **puntatore** sono passati **per riferimento**, cioè viene passato l'**indirizzo** dei parametri e la funzione lavora sui parametri originali.

I puntatori

- Un **puntatore** è una variabile che contiene l'**indirizzo** di un'altra variabile.

- L'operatore `&` fornisce l'indirizzo di un oggetto:
  `p = &c;` assegna a `p` l'indirizzo di `c`, i.e., `p` "punta a" `c`.

- L'operatore `*` (indirezione/deriferimento) si applica ad un puntatore e fornisce come risultato l'oggetto puntato da quest'ultimo.

- Esempi di dichiarazione ed uso di puntatori:

```
int x=1, y=2, z[10];
int *ip;         /* ip è un puntatore a interi */
ip = &x;         /* ip punta a x */
y = *ip;         /* y vale 1 */
ip = &z[0];      /* ip punta a z[0] */
*ip = *ip + 10; /* incrementa di 10 il valore di z[0] */
```

Le strutture

Una **struttura** C è una collezione di variabili di uno o più tipi, raggruppate sotto un nome comune.

- **Dichiarazione** di una struttura:

```
struct point {
        int x;
        int y;
};
```

  La dichiarazione di una struttura definisce un tipo.

- **Dichiarazione** di una **variabile** di tipo `struct point`:

```
struct point punto1;
```

- **Dichiarazione ed inizializzazione** di una **variabile** di tipo `struct point`:

```
struct point punto2 = { 15, 7 };
```

Accesso alle componenti, puntatori a strutture

- Alle **componenti** (o **membri**) della struttura si accede con l'operatore `.`:

```
punto1.x = 3;
punto1.y = 5;
```

- **Dichiarazione** di un **puntatore a struttura**:

```
struct point *pp;
```

- L'**accesso** alle componenti della struttura puntata da `pp` avviene mediante l'operatore `->`:

```
pp->x = 3;
```

Argomenti sulla linea di comando

Analogamente a quanto succede con i comandi Unix, è possibile scrivere dei programmi C che accettano argomenti sulla linea di comando:

```
#include <stdio.h>

main(int argc, char *argv[]) /* argc: argument count (n. argomenti+1) */
                             /* argv: argument vector (puntatore ad   */
                             /* un vettore di stringhe che contiene   */
                             /* gli argomenti).                       */
{
    int i;

    for(i=1; i<argc; i++)
        printf("%s\n",argv[i]);

    printf("\n");
    return 0;
}
```

Il programma precedente emula il comando Unix `echo`, nel senso che accetta un numero arbitrario di comandi e li stampa sullo standard output.

N.B.: `argv[0]` è il nome del programma C compilato, mentre `argv[argc]` è la costante `0` (i.e., un **null pointer**).

Le funzioni `printf` e `getchar`

La funzione

```
printf(stringa,arg2,arg3,...);
```

prende come primo argomento una stringa di caratteri da stampare (es., `"%3.0f %6.1`
in cui ogni occorrenza del simbolo % indica il punto in cui devono essere sostituiti, nell'ordine, il 2^o, 3^o, ... argomento.
I caratteri successivi ad ogni % indicano il formato in cui deve essere stampato l'argomento.

La funzione

```
int c = getchar();
```

attende, invece, che l'utente del programma digiti un carattere sullo standard input e lo salva nella variabile intera c, eseguendo un'operazione di casting da carattere ad intero. Se lo standard input non contiene un carattere di fine file (*EOF*) o se si verifica un errore, c prende il valore della costante EOF=-1.

Per ottenere più informazioni sulle diverse funzioni si può usare `man`.

Input formattato: `scanf` e `sscanf` (I)

- La funzione di libreria
  `int scanf(char *format,...);`
  legge i caratteri dallo standard input interpretandoli secondo il formato specificato dal primo argomento e memorizzando i risultati nei rimanenti argomenti, che **devono** essere dei **puntatori**.
  Ad esempio `scanf("%f %d", &x, &i);` legge dallo standard input un numero a virgola mobile ed un intero e li assegna, rispettivamente, alle variabili `x` e `i` (si noti l'uso dell'operatore `&` per ottenere gli indirizzi delle variabili argomento).

- L'esecuzione di `scanf` termina quando esaurisce l'argomento `format`, quando l'input non soddisfa le specifiche od in caso di end-of-file.

- Il valore restituito da `scanf` rappresenta il numero di elementi in input che sono stati memorizzati con successo negli argomenti corrispondenti. In caso di end-of-file viene invece restituito il valore `EOF`.

- La funzione
  `int sscanf(char *string, char *format,...);`
  si comporta esattamente come `scanf` tranne per il fatto di leggere i caratteri dalla stringa puntata da `string` invece che dallo standard input.

Accesso a file

Un programma C può leggere l'input da file passati sulla linea di comando.

Prima di leggere/scrivere su un file, un programma C deve **aprire** il file tramite la funzione di libreria

```
FILE *fopen(char *name, char *mode);
```

La funzione `fopen` prende come parametro il nome del file, `name`, e una stringa, `mode`, che indica il **modo** di utilizzo del file, `r` (lettura), `w` (scrittura), `a` (append), e restituisce un **puntatore** (**file pointer**) da utilizzare per la lettura/scrittura del file. Il file pointer punta ad una struttura che contiene informazioni sul file (indirizzo di un buffer, posizione corrente nel buffer, etc.)

Dichiarazione di un file pointer e chiamata a `fopen`:

```
FILE *fp;
fp = fopen(name, mode);
```

Lettura, scrittura e chiusura di file

Una volta aperto, un file può essere letto/scritto mediante le funzioni:

- `int getc(FILE *fp)` che ritorna il prossimo carattere del file `fp`, `EOF`, in caso di errore o fine file;
- `int putc(int c, FILE *fp)` che scrive il carattere `c` sul file `fp` e ritorna il carattere scritto, oppure `EOF` in caso di errore;
- `char *fgets(char *line, int maxline, FILE *fp)` che legge dal file `fp` un numero di caratteri pari al minimo fra `maxline-1` e quelli compresi tra la posizione corrente ed il prossimo carattere di newline, memorizzandoli nell'array di caratteri puntato da `line` (in caso di errore o di end-of-file restituisce `NULL`, altrimenti `line`);
- `int fputs(char *line, FILE *fp)` che scrive nel file `fp` la stringa puntata da `line` (in caso di errore restituisce `EOF`, altrimenti `0`).

Al termine delle operazioni di lettura/scrittura di un file, è buona norma rilasciare il file pointer, utilizzando la funzione

`int fclose(FILE *fp)`

Compilazione di programmi C

I programmi C si memorizzano in file con estensione `.c` in Unix.

Supponendo quindi di aver salvato il programma precedente nel file `ciao_mondo.c`, per compilarlo usiamo il comando `cc` (C compiler) o `gcc` in Linux (GNU C compiler) nel modo seguente:

```
> gcc ciao_mondo.c
```

Il risultato del precedente comando è un file **binario** `a.out` che contiene l'immagine dell'**eseguibile** da caricare in memoria. Quindi digitando

```
> ./a.out
```

verrà stampata sullo standard output la stringa `ciao, mondo!` seguita da un newline.

Per ottenere un file eseguibile con un nome più significativo di `a.out`, è sufficiente specificarlo con l'opzione `-o`:

```
> gcc -o ciao_mondo ciao_mondo.c
> ./ciao_mondo
```

Il comando `make`

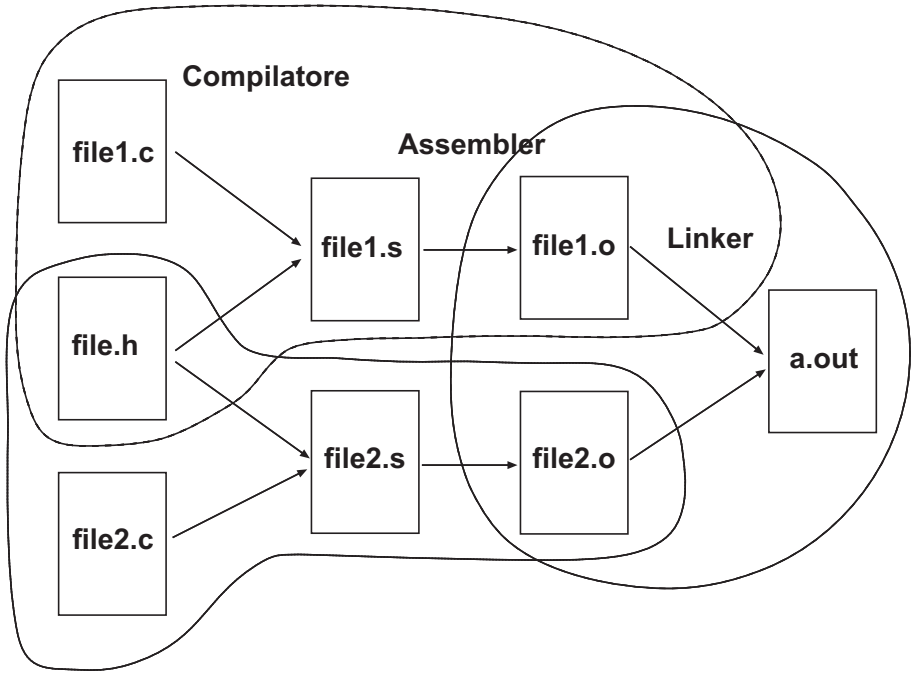
- Sviluppando programmi complessi, si è spesso portati a suddividere il codice sorgente in diversi file. La fase di compilazione quindi richiede maggior tempo, anche se le modifiche apportate riguardano soltanto una piccola parte dell'intero progetto.
- Il comando `make` è uno strumento che aiuta il programmatore nella fase di sviluppo, tenendo traccia delle modifiche apportate e delle dipendenze fra i vari file, ricompilando soltanto quanto necessario.
- Per produrre un eseguibile da un programma C sono necessari tre passi compiuti dai seguenti moduli:

  1. Compilatore (fase preceduta dall'invocazione del preprocessore): converte il codice sorgente in linguaggio Assembly (linguaggio di basso livello).

  2. Assembler: converte il codice in linguaggio Assembly in linguaggio macchina (direttamente eseguibile dal processore).

  3. Linker: "collega" il codice macchina prodotto dalla fase precedente a quello delle funzioni di libreria utilizzate nel programma (e.g., `printf`).

Compilazione con più file sorgente

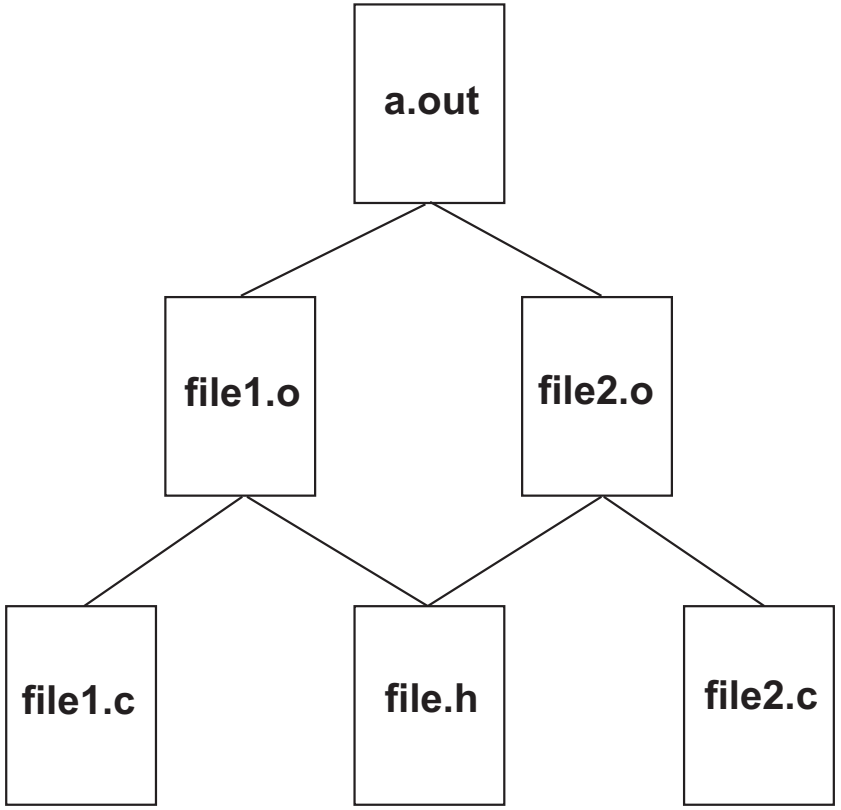


Si può pensare all'intero processo come al risultato dell'esecuzione dei comandi seguenti:

```
> gcc -c file1.c
> gcc -c file2.c
> gcc file1.o file2.o
```

N.B.: eseguendo il compilatore con l'opzione `-c` il processo si ferma dopo l'esecuzione dell'Assembler, producendo i file con estensione `.o` invece dell'eseguibile `a.out`.

Grafo delle dipendenze



Il grafo mette in luce le dipendenze fra i vari file (e.g., `file2.o` dipende sia da `file.h` che da `file2.c`. Quindi, in caso di modifiche a quest'ultimo, percorrendo il grafo verso l'alto notiamo che devono essere aggiornati anche `file2.o` e `a.out`, mentre i rimanenti file vengono lasciati inalterati.

Il `Makefile`

- Il grafo delle dipendenze viene codificato in un file di testo chiamato `Makefile` che risiede nella stessa directory dei file sorgente.

- Esempio di `Makefile`:

```
a.out: file1.o file2.o
    cc file1.o file2.o
file1.o: file.h file1.c
    cc -c file1.c
file2.o: file.h file2.c
    cc -c file2.c
```

- Un `Makefile` è composto da regole specificate dalla seguente sintassi (l'ordine delle regole non è importante):

```
target : source file(s)
    command
# Importante: command deve essere preceduto da un <Tab>
```

- Il comando `make` controlla le date di ultima modifica dei file. Ogni volta che un file (source) ha una data di modifica più recente di quella dei file che da esso dipendono (target), questi ultimi vengono aggiornati eseguendo i comandi specificati nelle regole del `Makefile`.

Utilizzo di `make`

Se le regole sono memorizzate in un file chiamato `Makefile` o `makefile` è sufficiente digitare il comando `make` seguito dal target che si vuole aggiornare (altrimenti si deve usare l'opzione `-f` per specificare il file corretto).
Se non si specifica alcun target, viene eseguito automaticamente il primo (per questo solitamente la prima regola è quella che permette di ottenere il risultato finale).
Esempio:

```
a.out: file1.o file2.o
    cc file1.o file2.o
file1.o: file.h file1.c
    cc -c file1.c
file2.o: file.h file2.c
    cc -c file2.c
clean:
    rm -f *.o a.out
```

La prima esecuzione di `make` lancia i seguenti comandi:

```
> cc -c file1.c
> cc -c file2.c
> cc file1.o file2.o
```

Il comando `make clean` provoca **sempre** l'esecuzione di `rm -f *.o a.out` in quanto `clean` è un target senza dipendenze.

Lo script `configure`

A volte il makefile è specifico per una piattaforma (es. a causa dell'utilizzo di particolari librerie che diverse distribuzioni installano in percorsi diversi).

In questi casi deve poi essere adattato prima di lanciare make.

Poiché è difficile modificarlo manualmente, la maggior parte dei software fornisce uno script chiamato `configure` per generare i makefile.

In questo caso, per compilare un programma è necessario digitare in sequenza:

```
> ./configure
> make
> make install
```

dove `make install`, da eseguire come amministratore, copia i compilati nelle apposite directory di sistema.

Per automatizzare la creazione degli script configure, esistono appositi tool, come Automake e CMake.