Appunti sul linguaggio C

25 novembre 2015

Indice

Parte I

Linguaggio

Capitolo 1

Introduzione al C

1.1 Caratteristiche

Il linguaggio C fu implementato all'inizio degli anni 70 presso i Bell Labs, per permettere la semplificazione del porting del kernel `UNIX`, precedentemente scritto in Assembly, su differenti architetture.
Nel 1988 del linguaggio è stato dato uno standard ANSI; i seguenti due standard sono quelli del 1999 e del 2011.

1.1.1 Caratteristiche del Linguaggio

Le caratteristiche principali del C sono:

- *Economia di linguaggio*: sintassi sintetica, pochi modi di fare le cose
- *Efficienza*: gestione a fondo della memoria, strumento per creare programmi efficienti;
- *Linguaggio di alto livello*: è il linguaggio di più basso livello, tra quelli di alto livello;
- Utilizzo frequente di *chiamate a funzioni*;
- *Largo uso di puntatori*
- *Non vi è un controllo stretto sui tipi di dato*, nelle dichiarazioni e nelle valutazioni: se può essere fonte di bug, può permette di risolvere velocemente i problemi senza dover sottostare a strutture rigide.

1.1.2 Caratteristiche di ogni programma

Un esempio di programma minimale in C è

```
1 /* Esempio di primo programma*/
2 #include <stdio.h>
3
4 int main()
5 {
```

```
6   printf("Hello, World!\n");
7   return 0;
8 }
```

Ogni programma:

- deve contenere una e una sola funzione **main**, che rappresenta il programma principale, ed è il punto di inizio dell'esecuzione del programma.
- fa uso di **librerie** ovvero di codice/funzionalità già disponibili per l'utilizzo.
  La libreria standard C (quella che fornisce funzioni standard di input, output, matematica di base) è la più importante poichè molte altre si appoggiano su di essa per l'esecuzione dei compiti di base.
  Non tutte le librerie sono disponibili su tutte le architetture di calcolatori. Tuttavia il ristretto insieme di funzioni definito da ANSI viene reso disponibile dalla maggior parte dei compilatori.
  Molti compilatori mettono a disposizione librerie proprietarie, che se da un lato facilitano la programmazione all'utente, dall'altro rendono difficile il **porting** del programma, ossia il trasferimento dello stesso su architetture differenti.
  Le librerie vengono materialmente utilizzate nella programmazione mediante direttive di **inclusione degli header** ad opera del preprocessore.
- fa largo uso di **funzioni**, che costituiscono l'unità di programmazione del linguaggio
- è documentato dai **commenti**, che possono esser posti ovunque facendo uso di `/**/`; tutto quello che è compreso tra i due asterischi viene considerato commento.

1.2 Fasi di sviluppo di un programma

Lo sviluppo di un programma in C avviene attraverso le seguenti fasi:

1. **scrittura del file sorgente**, un file di testo con estensione `.c`.
2. **compilazione** dei sorgenti. Essa segue le seguenti fasi, seguite rispettivamente da preprocessore, compilatore, assemblatore e linker:
   (a) *preprocessore*: il codice sorgente viene controllato dal preprocessore. Il risultato sarà il codice sorgente "ripulito". Più precisamente, il preprocessore:
   - interpreta speciali direttive per il preprocessore che iniziano con `#`, come `#include` o `#define` o roba della compilazione condizionale (`#ifndef`, `#else`, `#endif`).
   - rimuove eventuali commenti presenti nel sorgente;
   - controlla la presenza di eventuali errori nel codice.

   (b) *compilazione* vera e propria: il compilatore prende in input il codice ottenuto dal preprocessore e crea il codice oggetto, in cui ancora esistono dei riferimenti non risolti.

Eventualmente può essere creato codice assembly, mappabile 1 a 1 con il codice macchina, ma scritto in una forma umanamente comprensibile e utile per ottimizzazione o debugging.

(c) *linking*: il linker prende in input i vari file di codice oggetto, sia quelli del programma che quelli delle librerie, per creare il file eseguibile collegando tutti i riferimenti a variabili e funzioni da un file all'altro.

3. **esecuzione** del programma compilato;
4. **debugging** del programma per la rimozione di bug.

Compilazione con `gcc` Una compilazione con numerosi check e nell'ipotesi di programmazione secondo standard ANSI avviene attraverso

```
gcc -Wall -Wextra -ansi -pedantic file.c
```

1.3 Setup del sistema in Debian

Tool per lo sviluppo Installare i seguenti tool

```
apt-get install build-essential gdb valgrind git indent
```

Documentazione La documentazione principale rilevante può essere installata mediante

```
apt-get install manpages-dev manpages-posix-dev glibc-doc-reference
apt-get install glibc-doc gcc-doc gdb-doc
```

Brevemente:

- `manpages-dev`: nella sezione 2 della documentazione le system call di Linux, nella sezione 3 le pagine di manuale delle funzioni della libreria standard del C (es `man 3 strtok`) e degli header (es `man 3 stdio`);
- `manpages-posix-dev`: funzioni ed header di estensione dello standard POSIX;
- `glibc-doc-reference` contiene il manuale della libreria del C in formato html o info (`info libc`);
- `glibc-doc`: contiene le pagine di manuale per le funzioni di libpthread;
- `gcc-doc`: documentazione sul compilatore gcc (in formato html ed info, `info gcc`).
- `gdb-doc`: documentazione sul debugger gdb (in formato html ed info, `info gdb`).

Capitolo 2

Tipi di dati, operatori, espressioni

Costanti e **variabili** sono le principali categorie di dati utilizzate da un programma. Esse sono caratterizzate dall'appartenere ad un determinato tipo di dato, che determina la gamma di valori assumibili, la memoria occupata (che dipende anche dal sistema operativo) e le operazioni ammesse sul dato.
Le **espressioni** combinano variabili e/o costanti per produrre nuovi valori.

2.1 Tipi di dati

2.1.1 Tipi fondamentali

Il C prevede un numero ristretto di tipi di dati fondamentali: tra gli **interi** abbiamo `char` e `int`, tra i **reali** `float` e `double`.
Più in dettaglio:

- `char`: un byte, in grado di contenere un carattere o un numero di piccole dimensioni. In ogni assegnamento ad una variabile di tipo carattere possiamo assegnare o il codice numerico (derivato dalla tabella ASCII) o il carattere rappresentato da quel codice numerico, posto tra apici singoli.
- `int`: intero, numero più grande del `char`
- `float`: numero a virgola mobile precisione singola
- `double`: numero a virgola mobile precisione doppia

2.1.2 Qualificatori

È possibile specificare ulteriormente alcuni di questi tipi mediante i qualificatori di dimensione `short` e `long`, o mediante i qualificatori di **segno** `signed` e `unsigned`.
Entrambi servono per specificare il range dei valori rappresentabili con ogni tipo di dato; i **qualificatori ammessi**, per tipo di dato sono esposti in tabella 2.1.

	short	long	signed	unsigned
char			•	•
int	•	•	•	•
float				
double		•		

Tabella 2.1: Tipi e qualificatori

Gli header `<limits.h>` e `<float.h>`[1] specificano il range dei valori assumibili; ad esempio nel seguito una porzione di codice di `<limits.h>` che specifica i bit di un `char` e i valori assumibili se esso è `signed` o `unsigned`.

```
#  define CHAR_BIT      8
/* Minimum and maximum values a `signed char' can hold.  */
#  define SCHAR_MIN     (-128)
#  define SCHAR_MAX     127
/* Maximum value an `unsigned char' can hold.  (Minimum is 0.)*/
#  define UCHAR_MAX     255
```

Qualificatori di dimensione I qualificatori `short` e `long` servono per specificare una minore o maggiore quantità di bit destinata per ogni tipo di dato. Per verificare la quantità di bit (che è architettura-dipendente) può tornare utile l'operatore `sizeof` (che può restituire la dimensione di un tipo di dato)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6     printf("sizeof(char) = %u\n", (unsigned int) sizeof(char));
7     printf("sizeof(short int) = %u\n", (unsigned int) sizeof(short int))
8     printf("sizeof(int) = %u\n", (unsigned int) sizeof(int));
9     printf("sizeof(long int) = %u\n", (unsigned int) sizeof(long int));
10    printf("sizeof(float) = %u\n", (unsigned int) sizeof(float));
11    printf("sizeof(double) = %u\n", (unsigned int) sizeof(double));
12    printf("sizeof(long double) = %u\n", (unsigned int) sizeof(long doub
13    return 0;
14
15 }
```

che eseguito sulla mia macchina (sistema UNIX) restituisce:

```
sizeof(char) = 1
sizeof(short int) = 2
sizeof(int) = 4
sizeof(long int) = 8
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 16
```

[1] In Debian non si trova in `/usr/include`; utilizzare `locate` per trovarlo

Qualificatori di segno I qualificatori `signed` (con segno) o `unsigned` (senza segno) posson esser applicato a qualunque tipo intero (`int` o `char`).
Nel caso un intero sia `signed`, il primo bit viene destinato al segno (numero positivo se `0`, negativo se `1`).
Perdendo il primo bit per il segno, il valore assoluto assumibile dal dato diminuisce. Tuttavia specificando signed è possibile utilizzare e memorizzare numeri negativi.

2.1.3 Altri tipi di dati

Tipo void Il tipo `void` rappresenta un tipo di dato indefinito, e svolge due funzioni:

1. indica che una funzione non riceve come parametro o non restituisce come output nessun valore;
2. serve per definire un puntatore che punta ad un dato generico.

Tipo booleano Nol C **non esiste** un tipo booleano; come variabili di tipo boolano si debbono utilizzare variabili intere, signed o meno. Quando valutate dal punto di vista booleano il valore di una espressione numerica è:

- *falso* se da come risultato **0**;
- *vero* se da come risultato un **qualsiasi numero diverso 0**;

2.2 Costanti

Le *costanti*

- sono dati che non possono essere modificate durante l'esecuzione del programma, a contrario delle variabili;
- sono dati per le quali, sempre a contrario delle variaibli, il compilatore non riserva nessuno spazio in memoria nel quale è possibile salvare qualcosa; pertanto assegnare un valore ad una costante in una istruzione è un errore di sintassi.

2.2.1 Costanti senza identificatore

2.2.1.1 Costanti numeriche intere

Sono espresse da numeri interi, eventualmente preceduti da segno.
Le costanti intere possono essere scritte in **notazione** differente a seconda del *prefisso* al numero:

- decimale, senza prefisso;
- ottale, ponendo come prefisso uno `0`;
- in notazione esadecimale, apponendo `0x` o `0X`[2].

[2]Su macchine che hanno Gcc è possibile specificare costanti binarie mediante `0b` o `0B`.

Per indicare il **tipo di dato** con cui rappresentare la costante, si utilizzano dei caratteri *postfissi*; nello specifico[3]:

- di default le costanti numeriche sono considerate `int`; se poi la costante è troppo grande viene considerata `long int`;
- se si vuole specificare una costante come `long int` deve essere seguita dal carattere `L` o `l` (es: `49761234L`);
- una costante `unsigned` deve essere seguita dal carattere `U` o `u`;
- una costante `unsigned long` deve essere seguita dai caratteri `UL` o `ul`.

2.2.1.2 Costanti numeriche in virgola mobile

Possono esser scritte in **notazione** decimale o esponenziale:

- in notazione decimale, ad esempio

```
-10.4          7.          32.235f          .001
```

- in notazione esponenziale, ad esempio

```
-1.04e+1       0.37235e+2L       0.7e+1       1.e-2
```

Il **tipo di dato** utilizzato per la costante dipende dall'eventuale *suffisso* e sarà:

- `double` se non viene specificato un suffisso alla costante
- `float` se viene aggiunto un suffisso `f` o `F` (es `37.235f`)
- `long double` se viene aggiunto un suffisso `l` o `L` (es. `0.37235e+2L`)

2.2.1.3 Costanti di tipo carattere

La *costante di tipo carattere* possono essere rappresentate da un carattere compreso tra apici singoli, come `'a'`, o alternativamente attraverso il numero (codifica ASCII) corrispondente alla lettera desiderata.
La tabella ASCII è uno standard per le prime 128 cifre, dopo varia a seconda del computer.

```
    0 1 2 3 4 5 6 7 8 9
 30     ! `` # $ % & '
 40 ( ) * + , - . / 0 1
 50 2 3 4 5 6 7 8 9 : ;
 60 < = > ? @ A B C D E
 70 F G H I J K L M N O
 80 P Q R S T U V W X Y
 90 Z [ \ ] ^ _ ` a b c
100 d e f g h i j k l m
110 n o p q r s t u v w
120 x y z { | } ~
```

[3] Esempi con numeri decimali

Pertanto, ad esempio i seguenti statement assegneranno ad `a` e `b` il medesimo valore:

```
char a = 'A';
char b = 65;
```

Alcuni caratteri particolari si esprimono mediante le **sequenze di escape**, alcune delle quali sono listate in seguito:

```
\a      bell
\b      backspace
\n      carattere newline (vai a capo)
\r      ritorno del carrello
\t      tab orizzontale
\v      tab verticale
\\      backslash
\?      punto interrogativo
\'      apice (delimitatore di carattere)
\"      virgolette (delimitatore di stringa)
\ooo    numero in notazione ottale
\xhh    numero in notazione esadecimale
```

2.2.1.4 Stringhe

Una stringa viene scritta come una sequenza di caratteri racchiusa tra 2 virgolette, le quali non fanno parte della stringa.

```
"Questa e' una stringa"
```

In C le stringhe **non esistono come tipo di dato primitivo**: per rappresentarle internamente si usa un vettore di $n + 1$ `char`, dove n è la lunghezza della stringa e il $+1$ è dovuto al carattere in ultima posizione, il nullo $\backslash 0$, che rappresenta il delimitatore finale della stringa (ossia l'indicazione che essa è finita).

Bisogna prestare attenzione a distinguere una costante di tipo carattere e una stringa con un solo carattere: `'x'` non è uguale a `"x"`. Il primo è un intero, e denota il valore numerico della lettera `x` nel set di caratteri della macchina. Il secondo è un vettore di caratteri contenente un carattere e un $\backslash 0$.

Le costanti stringa possono essere **concatenate** in fase di compilazione. Ciò si effettua ponendo una costante stringa a fianco dell'altra, non separate da token del linguaggio se non da spazi. Questo torna utile per spezzare le stringhe lunghe e distribuirle su più righe di codice:

```
printf("The licenses for most software "
       "and other practical works are designed "
       "to take away your freedom to share and "
       "change the works.")
```

2.2.2 Costanti con identificatore

A volte si desidera associare un identificatore ad una costante; questo perchè:

- si può dover utilizzare la costante in diversi punti (es un dato importante) del programma e non la si vuole definire una volta sola e riutilizzare in maniera sicura
- le variabili non sono sempre facili da rintracciare e il cambiamento del loro valore è meno immediato.

Le costanti con identificatore possono essere create mediante

- l'impiego del comando `define` del preprocessore;
- la definizione di enumerazioni

Solitamente gli **identificatori** di queste sono *maiuscoli*, per distinguerli da quelli delle variabili.

2.2.2.1 L'uso di `define` per creare costanti

La direttiva al preprocessore `define` permette di associare un identificatore ad una costante, come negli esempi seguenti:

```
1 #define SIZE 10
2 #define TRUE 1
```

In tal modo potremo utilizzare, all'occorrenza il simbolo `TRUE` per ottenere 1: ciò che viene effettuato in realtà è una sostituzione ad opera del preprocessore nella fase prima della compilazione vera e propria.
Si noti che alla fine dello statement *non c'è il punto e virgola.* Se ci fosse, ad esempio, al posto di `SIZE` verrebbe sostituito `10;`, non `10`, creando non pochi macelli.

2.2.2.2 Le enumerazioni

Le enumerazioni forniscono una comoda tecnica alternativa a `#define` per associare nomi a costanti. Al primo identificatore il cui valore non viene specificato viene associato il valore 0, al secondo non specificato 1 e così via.

```
enum boolean {FALSE , TRUE};
```

dove `boolean` è opzionale e costituisce un *tag* dell'enumerazione, mentre tra parentesi si ha la *lista degli alias.* Per specifica i valori assegnati:

```
enum escapes { BELL='\a', TAB='\t', NEWLINE='\n' };
```

Qualora rimangano valori non specficati, essi continuano la progressione a partire dall' ultimo valore specificato:

```
enum months { JAN=1 , FEB, MAR, APR, MAY, JUN
              JUL   , AUG, SEP, OCT, NOV, DEC } ;
/* FEB=2, MAR=3 , ... , DEC=12 */
```

I vantaggi principali dell'enumerazione, rispetto ad una serie di `#define`, consistono nell'economia di linguaggio e nella generazione automatica dei valori (diminuendo la possibilità di errori).

2.3 Operatori

Si distinguono gli operatori:

- aritmetici
- di incremento-decremento
- relazionali
- logici
- di assegnamento
- per la manipolazione dei bit: si vedranno in apposita sezione.

cui si aggiungono l'operatore `sizeof`, l'operatore ternario delle *espressioni condizionali* e l'operatore di *cast*.
Detti operatori possono distinguersi in *unari* e *binari* a seconda che si applichino rispettivamente ad uno o due operandi.

2.3.1 Operatori aritmetici

Sono generalmente operatori *binari*:

```
+   addizione (binario)
-   sottrazione (binario)
-   cambio segno (unario)
*   moltiplicazione (binario)
/   divisione (binario)
%   resto della divisione tra interi (binario)
```

La divisione da come output:

- un valore in virgola mobile se almeno uno dei due tra dividendo e divisore è in virgola mobile floating;
- un valore intero se tra due interi; in tal caso vengono troncati gli eventuali decimali. La direzione di troncamento di `/` e il segno del risultato di `%` dipendono dalla macchina.

2.3.2 Operatori di incremento-decremento

Operatori unari, `++` e `-` aggiungono o tolgono rispettivamente un'unità alla variabile cui si applicano.
Equivalgono ad assegnare alla stessa variabile il proprio valore incrementato di uno:

```
x++     x = x+1
x--     x = x-1
```

ma lavorano più velocemente perchè sfruttano meglio i registri.
Nel caso l'operando sia un puntatore, lo incrementano della quantità necessaria a farlo puntare all'inizio dell'elemento che segue (si vedrà nell'aritmetica dei puntatori).

Qualora l'incremento sia parte di un'istruzione più complessa è utile distinguere i due usi di questi operatori, quello come *prefisso* e quello come *suffisso*.

```
x++;      prima usa x nell'istruzione, poi lo incrementa
++x;      prima incrementa x, poi lo usa nell'istruzione
```

Da isolate non fanno differenza, ma in istruzioni meno semplici cambia.

```
int n, m=0;
n = m++;         /* n vale 0, m vale 1 */
n = ++m;         /* n vale 2, m vale 2 */
```

2.3.3 Operatori relazionali

```
>    maggiore
>=   maggiore o uguale
<    minore
<=   minore o uguale
==   uguale
!=   non uguale
```

Uno stratagemma che a volta si adotta con l'operatore == all'interno di strutture decisionali dove si confronta una variabile con una costante, è di porre a sinistra la costante e a destra la variabile, come segue

```
8 == var
```

invece

```
var == 8
```

in quanto cancellazioni accidentali di un = daranno origini ad errori di compilazione che segnaleranno l'accaduto.

2.3.4 Opereratori logici

```
&&   and
||   or
!    not
```

La valutazione, per ciò che riguarda && e || avviene da sinistra verso destra e si interrompe appena il risultato si rivela vero (nel caso di ||) o falso (in quello di &&).
Questo modo di valutare le espressioni velocizza le operazioni perché non esegue operazioni inutili.
Pertanto, per efficienza, nelle espressioni che utilizzano && mettere prima la condizione che ha più probabilità di esser falsa; nelle espressioni con || prima la condizione con probabilità maggiore di esser vera.

2.3.5 Operatori di assegnamento

Istruzioni C del seguente tipo

```
exp1 op= exp2
```

dove `op` è un'operatore un'operatore del tipo

```
+ - * / % << >> & ^ |
```

equivalgono all'istruzione

```
exp1 = exp1 op exp2
```

Ad esempio

```
Questo...      ...equivale a...
x += y             x = x+y
```

Questi operatori composti servono a *velocizzare il codice*, perchè evitano di valutare ripetutamente una stessa espressione (`exp1`), e mantenendo i dati nei registri *minimizzano l'accesso alla memoria.*

2.3.6 Operatore ternario per le espressioni condizionali

Vi sono situazioni in cui una determinata valutazione deve produrre due valori distinti a seconda di una determinata condizione. Per generare questo meccanismo si utilizza l'operatore di *espressione condizionale*; svolge pertanto una funzione similare a quella di `if`, all'interno delle espressioni e non nel flusso di comando.

Operatore ternario (i tre operandi sono *condizione* da verifica, espressione prodotta se questa è *vera*, o se *falsa*)

```
condizione ? se.vera : se.falsa
```

Un'applicazione[4]

```
int a=1;
int b=2;
int Max = (a>b) ? a : b ;
printf("max(1,2)=%d\n", Max);
```

Che come risultato da:

```
l@a6k:~$ ./a.out
max(1,2)=2
```

Si consiglia di mettere `condizione` tra parentesi più che altro per la leggibilità del codice.

[4] Interessanti applicazioni a pag 50 del K&R.

2.3.7 L'operatore `sizeof()`

Il C mette a disposizione, fra gli altri, l'operatore unario `sizeof()` che restituisce la dimensione della variabile o del tipo. La sintassi è duplice:

```
sizeof (tipo di dato)
sizeof (oggetto)
```

Pertanto prende in input:

- come *tipo di dato* un tipo fondamentale (es. `char`, `int` ...) o uno derivato (es. struttura o altro tipo definito dall'utente);
- come *oggetto* una variabile, un vettore o una struttura.

Restituisce un intero senza segno (tipo `size_t`) con la dimensione in byte dell'input. Un esempio consta nella determinazione del numero di elementi di un vettore `int`):

```
#define ITEMSOF(arr) (sizeof(arr) / sizeof(arr[0]))
int a[20];
printf("# of elements: %d\n", ITEMSOF(a) );
```

restituisce come risultato 20.
Questo operatore è molto importante per la **portabilità** del codice, in quanto risulta utile per determinare le dimensioni di alcuni tipi di dati che potrebbero cambiare al variare dell'architettura su cui il programma deve girare (es cambiano le dimensioni degli interi).

2.4 Valutazione delle espressioni

La valutazione delle espressioni è quella fase in cui il sistema valuta un insieme di costanti variabili ed operatori al fine di produrre un nuovo valore.
Temporalmente la valutazione delle espressioni avviene:

- in sede di compilazione del programma per le espressioni formate esclusivamente da costanti ed operatori, dette **espressioni costanti**
- in sede di esecuzione (runtime) in tutti gli altri casi (es le espressioni che includono variabili)

Per analizzare la valutazione delle espressioni nel linguaggio C occorre precedentemente introdurre le conversioni del tipo di dato (o *type casting*).

2.4.1 Conversione di tipo (*type casting*)

Per lo svolgimento dei programmi può essere necessario effettuare la modifica di un tipo di dato. Tale modifica può essere automatica (**conversione automatica o implicita** di tipo) o svolta dall'utente (**conversione esplicita**).

2.4.1.1 Conversione automatica o implicita

La conversione di tipo automatica avviene:

- nella valutazione delle espressioni;
- negli assegnamenti;
- nei passaggi di argomenti a funzioni.

Nella **valutazione di espressioni**, il C effettua automaticamente delle conversioni implicite di tipo, specialmente quando effettua operazioni matematiche tra operandi di tipo diverso.
In tal caso l'operando più piccolo (con un minor numero di byte) viene convertito (o *promosso*) nel tipo più grande, prima che vengano effettuate le operazioni; ciò non implica che venga cambiato il tipo della variabile originaria. Nello specifico:

- se entrambi i fattori coinvolti in un calcolo sono interi, l'intero con meno bit sarà convertito nell'intero con più bit. Il risultato sarà un intero, del tipo del fattore con più bit: dunque risultato di una divisione tra interi sarà intero, senza decimali;
- quando i due operandi sono rispettivamente a virgola mobile e intero, quest'ultimo viene convertito in virgola mobile prima che l'operazione venga eseguita. Il risultato sarà in virgola mobile del tipo del fattore con più bit;
- se entrambi sono in virgola mobile, si convertirà al tipo in virgola mobile con un numero maggiore di bit. Il risultato sarà un numero in virgola mobile, del tipo del fattore con più bit.

Solo dopo questa conversione l'operazione viene effettuata, secondo le regole proprie del nuovo tipo di dato, ottenendo come risultato un valore coerente con il nuovo tipo.

Anche nell'**assegnamento** si effettuano conversioni di tipo automatiche, qualora l'espressione a destra abbia tipo differente dalla variabile a sinistra. In particolare il valore del lato destro verrà convertito nel tipo del lato sinistro, perdendo informazioni nel caso si passi da un tipo ad un altro rappresentato da un numero inferiore di bit.

Allo stesso modo avviene type casting automatico nel **passaggio di argomenti a funzioni**, essendo assimilabili ad un assegnamento.

2.4.1.2 Conversione esplicita

Il type casting può anche esser forzata dal programmatore mediante un opportuno operatore unario (detto *cast*):

```
(type) expr
```

dove `type` è un tipo di dato primitivo o definito dall'utente ed `expr` può esser una variabile, una costante o una espressione complessa.
`expr` viene risolta fino ad arrivare ad un risultato (di un suo certo tipo), poi il risultato viene convertito nel tipo `type`. Un esempio con costanti:

	Operatori	Associatività
1	`() [] -> .`	`sx -> dx`
2	`! ++ - + - * & (cast) sizeof ~`	`sx <- dx`
3	`* / %`	`sx -> dx`
4	`+ -`	`sx -> dx`
5	`« »`	`sx -> dx`
6	`< <= > >=`	`sx -> dx`
7	`== !=`	`sx -> dx`
8	`&`	`sx -> dx`
9	`^`	`sx -> dx`
10	`|`	`sx -> dx`
11	`&&`	`sx -> dx`
12	`||`	`sx -> dx`
13	`?:`	`sx <- dx`
14	`= += -= *= /= %= &= ^= |= «= »=`	`sx <- dx`
15	`,`	`sx -> dx`

Tabella 2.2: Associatività e diritto di precedenza fra operatori.

```
5 / 2;            /* due costanti intere, il risultato valutato è 2*/

(double) 5 / 2;   /* prima di effettuare la frazione, 5 viene convertito a
                     5.0, quindi abbiamo un rapporto tra un numero in
                     virgola mobile; 2 sarà convertito a 2.0 e il risultato
                     sarà 2.5 */

(double) (5 / 2);  /* per la precedenza imposta, il risultato sarà 2.0 */
```

2.4.2 Ordine operatori e valutazione delle espressioni

Gli operatori si pongono fra loro in un determinato ordine di priorità, che sancisce in quale ordine vengono effettuate le operazioni.
In tabella 2.2 muovendosi **verso il basso la priorità decresce**; a pari livello di priorità si valuta l'espressione da sinistra verso destra salvo i gruppi di operatori al numero 2, 13 e 14 . In generale è comunque meglio utilizzare parentesi tonde e spaziature in abbondanza: si commettono meno errori di programmazione guidando la valutazione delle espressioni tramite parentesi tonde ad hoc, piuttosto che affidandosi esclusivamente alla precedenza degli operatori.

Lo standard ANSI C specifica solamente l'ordine di valutazione tra operatori:

- **non specifica l'ordine di valutazione tra operandi di un medesimo operatore**, fatta eccezione per gli operatori logici `||` e `&&`, l'operatore ternario `?:` e l'operatore virgola.
  Ad esempio lo standard non specifica quale tra `f()` e `g()` verrà valutata per prima in questa somma:

  ```
  x = f() + g()
  ```

Se è importante l'ordine di valutazione o da esso dipendono i risultati perchè `f` modifica `g` o viceversa, salvare risultati parziali (posti nell'ordine desiderato) in altrettante variabili strumentali. Ad esempio se vogliamo prima `g` e poi `f`:

```
a = g();
b = f();\
x = a + b;
```

- **non specifica l'ordine di valutazione degli argomenti di una funzione**; alcuni compilatori si comporteranno in un modo, altri in un altro. Pertanto bisogna fare sì che l'ordine di valutazione non sia determinante per il corretto funzionamento del programma.

Capitolo 3

Variabili

Le variabili sono contenitori per dati: servono per memorizzare le informazioni e permetterne l'utilizzo all'interno del programma.

3.1 Identificatori

L'identificatore è il nome assunto alternativamente da una variabile, da una funzione o da una costante considerata.
In generale, un identificatore può esser costituito da uno o più caratteri; deve iniziare con una lettera o underscore. I caratteri successivi al primo possono esser numeri, lettere o underscore. Non sono ammessi caratteri di punteggiatura o altro, che hanno significati speciali.
Il C è **case sensitive** quindi `Foo` e `foo` si riferiranno a variabili (o funzioni, o costanti) differenti.
In C per convenzione si adoperano le minuscole per gli identificatori delle variabili e le maiuscole per gli identificatori delle costanti.

Qualsiasi identificatore deve esser differente dalle **parole riservate** al linguaggio.

```
1    break case char const continue default do double
2    else enum extern float for goto if int long register
3    return short signed sizeof static struct switch
4    typedef union unsigned void volatile while
```

3.2 Dichiarazione e assegnamento di variabili

La **dichiarazione** crea una nuova variabile, specificandone l'identificatore e rendendone esplicito il tipo; l'**assegnamento** memorizza delle informazioni nella variabile creata.
Dichiarazione ed assegnamento possono essere effettuati mediante istruzioni differenti, o nella medesima istruzione.

3.2.1 . . . in statement separati

Nel primo caso lo statement della **dichiarazione** ha la seguente forma:

```
type identificatore;
type identificatore1, identificatore2, identificatore3;
```

Con la prima forma si dichiara solamente una variabile di tipo `type`, avente nome `identificatore`. Nella seconda linea si sono dichiarate contemporaneamente più variabili aventi lo stesso tipo (separate da virgole)[1].

L'**assegnamento** successivo avrà la forma:

```
identificatore = espressione;
```

dove `espressione` può esser una semplice costante o una combinazioni di variabili, operatori, funzioni e costanti.

In generale, un **valore** assegnato **è riutilizzabile**. Esso può esser riutilizzato ad esempio:

- come espressione, ad esempio come condizione in un `if`, eventualmente tenendolo dentro parentesi per evitare confusioni:

  ```
  if (i = 10) func(9);
  ```

  `i = 10` vale `10 != 0`, e quindi la condizione è vera;

- per un ulteriore assegnamento, tenendolo alla destra di un uguale.

  ```
  int i, j, k ;
  k = j = i = 10;
  ```

  Nell'esempio abbiamo assegnato alla variabile a sinistra di un qualsiasi uguale il valore assegnato alla variabile alla destra dello stesso. Il vantaggio è un'esecuzione più veloce delle istruzioni di assegnamento separate, perchè il dato da assegnare è già caricato sui registri.

3.2.2 ... nel medesimo statement

Nel caso in cui *dichiarazione e inizializzazione siano contemporanee*, lo statement è:

```
tipo identificatore = espressione;
```

Ad esempio:

```
int       i = 0;          /* i inizializzato a 0 */
double    f = 13.7;       /* f inizializzato a 13.7 */
int       j=2, k, m=3;    /* j inizializzato a 2,
                             k non inizializzato,
                             m a 3 */
```

In generale è preferibile gia inizializzare le variabili in sede di dichiarazione per evitare eventuali erronei accessi a **valori spazzatura** che possono risiedere nelle variabili solamente dichiarate.
Infatti variabili `static` ed `extern` sono inizializzate a zero per default; ma le altre hanno valori iniziali spazzatura.

[1] È meglio in generale effettuare una dichiarazione singola per ogni variabile, che meglio si presta a successive modifiche e all'apposizione di un commento che chiarisca il contenuto della variabile per ogni riga (variabile).

3.3 Tipologie di variabili

A seconda della posizione in cui avviene la dichiarazione si distinguono tre tipi di variabili:

1. variabili locali
2. variabili globali
3. variabili parametro di funzione

3.3.1 Variabili locali

Sono le variabili **dichiarate all'interno di un blocco di istruzioni**, ovvero una qualsiasi sequenza di istruzioni racchiuse tra due parentesi graffe (es all'interno del corpo di una funzione, di un ciclo `for`). Le variabili locali devono esser *poste necessariamente all'inizio del blocco*, cioè mai dopo che nel blocco sia stata scritta un'istruzione diversa da una dichiarazione.
Alle variabili locali è possibile **l'accesso** solo dall'interno del blocco stesso: non sono visibili/utilizzabili dal di fuori del blocco. Dall'interno di un blocco è possibile vedere variabili di blocchi esterni, se non sono state sovrascritte.

```c
#include <stdio.h>
int main (void)
{
   int a = 2;
   int b = 1;

   {
     /* andiamo sopra alla variabile */
     int b = 3;
     /* variable nel blocco includente sono presenti
        se non sovrascritte nel blocco considerato */
     printf("in, a is %d\n", a);
     printf("in, b is %d\n",b);
     /* modifico variabile dell'environment
        includente */
     a = 9;
     printf("in2, a is %d\n", a);
   }

   printf("out, b is %d\n",b);
   return 0;
}

/* in, a is 2 */
/* in, b is 3 */
/* in2, a is 9 */
/* out, b is 1 */
```

Le variabili locali avranno un **ciclo di vita** che inizierà (vengono create) nel momento in cui il controllo entra nel blocco (momento in cui vengono caricate nello stack) e termineranno nel momento in cui il controllo ne esce dal blocco.

3.3.2 Variabili globali

Le variabili globali sono **dichiarate fuori da tutte le funzioni** (anche da `main`), in una posizione qualsiasi di un file sorgente[2].
L'**accesso** ad esse è possibile da qualsiasi funzione di qualsiasi modulo/file sorgente a patto che, alternativamente:

- la **dichiarazione** della variabile globale oppure una sua **definizione** mediante la sintassi

```
1 extern tipo nome_variabile;
```

  si trovi **prima della definizione della funzione** cui vogliamo garantire l'accesso della variabile globale.
  La definizione con `extern` è una cosa diversa da una dichiarazione:

  - la dichiarazione crea una variabile e alloca lo spazio necessario nella memoria
  - la definizione con `extern`, al contrario, dice al compilatore che nel file in cui la definizione è presente la variabile prescelta non esiste, ma esiste qualche altro file, e che il modulo con la dichiarazione `extern` è autorizzato ad usare la variabile.
    Pertanto il compilatore non si deve preoccupare se non la trova in questo file; sarà il linker a cercare in tutti i moduli fino a trovare il modulo in cui esiste la dichiarazione senza `extern` per la variabile in questione.

- si abbia una **definizione** con `extern` all'interno della funzione cui vogliamo concedere l'accesso, posta nella sezione delle dichiarazioni; in tal caso la variabile sarà accessibile fino all'uscita dal blocco.
  Questo si fa tipicamente se si vuol concedere accesso alla variabile a poche/determinate funzioni.

Le variabili globali hanno **ciclo di vita** pari alla durata in esecuzione del programma.

3.3.3 Variabili parametro di funzione

Sono le variabili che racchiudono i parametri passati come argomento alla funzione:

- la loro dichiarazione si trova nella dichiarazione della funzione;
- vengono assegnate in sede di chiamata della funzione stessa.

Analogamente alle variabili locali, l'**accesso** è garantito solamente all'interno del blocco (funzione) e hanno un **ciclo di vita** che inizia nel momento in cui il controllo entra nella funzione e termina nel momento in cui il controllo esce dal blocco.

[2] Le variabili globali sono utili come strumento di comunicazione tra sezioni di codice che non hanno una interazione diretta, come ad esempio nel caso due funzioni debbano condividere dati ma nessuna di esse chiama l'altra.
L'uso va comunque limitato allo strettamente necessario perchè complica il debugging del programma.

3.4 static e const

3.4.1 Lo specificatore static

3.4.1.1 L'uso di static con variabili locali

Se applico il qualificatore `static` alla dichiarazione di una variabile locale come in

```
1 static int foo;
```

questa viene definita e inizializzata a 0 la prima volta che si entra nel blocco; essa non viene posizionata nello stack bensì in una porzione di memoria permanente (esiste per tutta la durata del programma).
Quando il controllo uscirà dal blocco la variabile `static` mantiene il proprio valore: se si rientrerà una seconda volta si erediterà il vecchio valore.
La variabile locale `static` sarà visibile solo all'interno del blocco in cui è stata dichiarata (o al suo esterno se ne viene fornito l'indirizzo come valore di ritorno del blocco[3]).

3.4.1.2 L'uso di static con variabili globali

Se si applica `static` alla dichiarazione di una variabile globale collocata in un certo file faccio sì che detta **variabile globale sia accedibile solo in quel file**, e in nessun altro.
Tuttavia è possibile per una funzione del modulo passare un indirizzo della variabile globale ad un'altra funzione di un'altro modulo (rendendo possibile lettura e modifica).

3.4.1.3 L'uso di static con funzioni

La dichiarazione `static` può anche esser applicata alle funzioni, con effetto simile a quanto avviene per le variabili globali.
Di norma, i nomi delle funzioni sono globali, cioè visibili a qualunque brano del programma. Nel momento in cui si dichiara una funzione static (apponendo il qualificatore prima del tipo restituito nella definizione e nel prototipo) il nome della funzione diventa invisibile al di fuori del file in cui è dichiarata.

3.4.2 Lo specificatore const

`const` può essere applicato alla dichiarazione di qualsiasi variabile, con l'effetto che il suo valore non potrà esser modificato. È utile soprattutto nelle chiamate di funzioni per riferimento, per impedire la modifica dei valori passati.

[3] Vedi k&r pg. 111

Capitolo 4

Flusso del controllo

Le istruzioni per il flusso del controllo (*control flow*) stabiliscono il percorso seguito dalla computazione nel file sorgente e si possono suddividere in:

- **esecuzione condizionale**: `if` e `switch`
- **cicli**: `for`, `while` `do-while`

Alcune di queste possono essere accompagnate dalle istruzioni `break` e `continue`, che ne modificano il comportamento standard.
Prima di presentare dette istruzioni si presenta la definizione di *istruzione* di *blocco di istruzioni*.

4.1 Istruzioni e blocchi

Una qualsiasi **espressione** (come `3*2`, una assegnazione `x = 0`, o una chiamata di funzione) diviene una **istruzione** (o *statement*) quando è terminata da un punto e virgola. Questo comunica al compilatore che l'espressione è terminata ed occorre effettuare la sua valutazione.
L'istruzione consistente solo nel `;` è detta **istruzione nulla**. È dal punto di vista formale corretta e non da errori, ma non produce nulla.

Un **blocco di istruzioni** è una sequenza di istruzioni racchiuse tra parentesi graffe[1]. Alle parentesi graffe di chiusura di un blocco **non** segue un punto e virgola; se viene messo viene considerato un'altra istruzione (l'istruzione nulla). Dal punto di vista del linguaggio un blocco di istruzioni viene considerato come un'unica istruzione; per di più, dove risiede un'istruzione singola può risiedere anche un blocco e viceversa.

[1] Esempi di blocchi sono costituiti da il corpo delle funzioni, o i corpi di istruzioni multiple dopo `for`, `while` ecc.

4.2 Esecuzione condizionale

4.2.1 `if`

L'esecuzione/flusso condizionale si realizza come segue:

```
1      if (expr1)
2           istruz
3      else if (expr2)      /* else if non obbligatorio */
4           istruz
5      else                 /* else non obbligatorio */
6           istruz
```

Le espressioni vengono valutate nell'ordine in cui si presentano; viene eseguita l'istruzione in corrispondenza della prima espressione vera, dopodichè il ciclo viene interrotto. Per **efficienza**, quindi è opportuno porre come prime le espressioni che più probabilmente risulteranno vere in sede di esecuzione.
`if` e `else if` specificano condizioni determinate, mentre `else` funge da default per gli altri casi non specificati. `else if` ed `else` non sono obbligatorie.

Essendo `else` facoltativa, si crea **ambiguità** quando un `else` è omesso da una *sequenza if annidata*. Per convenzione il compilatore abbina `else` al più vicino `if` precedente che ne sia privo:

```
if (n > 0)
     if (a > b)
          z = a;
     else
          z = b;
```

`else` viene accoppiato dal compilatore con l'istruzione `if` più interna, come segnalato anche dalla rientranza. Se si vuole un risultato diverso è necessario ricorrere alle parentesi graffe per imporre il giusto abbinamento:

```
if (n > 0) {
     if (a > b)
         z = a;
} else
   z = b;
```

In sostanza, per **robustezza** di programmazione, è buona pratica **usare blocchi** anche al posto di singole istruzioni nel caso di `if` annidati.

4.2.2 `switch`

La sintassi è:

```
1     switch (myexpr)
2     {
3
4          case expr1:
5               istruz1
6               break;     /* break non obbligatorio */
7          case expr2:
8               istruz2
9               break;     /* break non obbligatorio */
10         ...
11
12         default:        /* default non obbligatorio */
13              istruz
14    }
```

dove:

- `exprX` sono espressioni costanti (formate solo da costanti e/o operatori, non da variabili o funzioni);
- `istruzX` una o più istruzioni;
- `myexpr` è una espressione che a contrario di `exprX` non deve essere un'espressione costante

`myexpr` viene confrontato in sequenza con le espressioni costanti `exprX`; nel caso vi sia uguaglianza vengono eseguite le istruzioni presenti al di sotto del caso desiderato.
Pertanto se entriamo nel ciclo al caso 4 e ci sono 6 casi (oltre al default) eseguiremo le istruzioni associate ai casi 4, 5, 6 e default.
`break` si usa tipicamente per evitare l'esecuzione a cascata implementata con questo meccanismo (che a volte può tornare utile) e fare sì che vengano eseguite soltanto le istruzioni tra l'etichetta che ha matchato e il primo `break` (che fa uscire dallo `switch`).
Le istruzioni associate al `case` di `default` vengono eseguite soltanto se nessuna delle altre è soddisfatta: come `else` è facoltativa.

4.3 Cicli

4.3.1 `while`

Il ciclo `while` presenta la sintassi

```
1     while (expr)
2        istruz
```

`expr` viene valutata; qualora sia vera, l'istruzione `istruz` viene eseguita, dopodichè `expr` viene rivalutata. Il ciclo prosegue finchè l'espressione `expr` risulta falsa (pertanto occorre prevedere istruzioni all'interno dell'istruzione/blocco, che agiscano sulla `expr` conducendola a falsità); solo allora il flusso lascierà il ciclo.

4.3.2 `for`

La sintassi è

```
1     for (expr1 ; expr2 ; expr3) /* exprX facoltative */
2        istruzione
```

Le tre espressioni (tutte opzionali) del ciclo `for` sono quelle che ne determinano il funzionamento. Possiamo rispettivamente definirle

- espressione di **inizializzazione** (`expr1`). Viene eseguita una sola volta, nel momento in cui il controllo passa a `for`; tipicamente usata per inizializzare variabili contatore;

- condizione di **prosecuzione** (`expr2`). Viene valutata ogni volta prima di eseguire le istruzioni del loop: se è vera si esegue l'istruzione (o il blocco), alternativamente si esce dal ciclo `for`.
  Nel caso in cui non sia presente, il compilatore valuta vero e continua ad eseguire le istruzioni dentro al loop (dal quale bisognerà uscire mediante l'istruzione `break`).

- espressione d'**incremento** (`expr3`). Viene eseguita una volta terminata l'esecuzione dell'istruzione (o del blocco) si passa all'esecuzione dell'incremento (o decremento) della variabile contatore. In seguito la condizione di prosecuzione viene rivalutata e così eventualmente il ciclo riparte.

4.3.3 `do-while`

La sintassi è:

```
1     do
2           istruz
3     while (expr);
```

Si esegue dapprima l'istruzione `istr` (o il blocco equivalente) e si valuta poi l'espressione `expr`: se questa è vera, si torna a eseguire `istr`, e così via. Il ciclo termina quando `expr` diventa falsa.
Pertanto a differenza di `while` e `for`, l'istruzione/blocco è sempre eseguito almeno una volta.

4.4 Interruzione del flusso

4.4.1 `break`

L'istruzione `break` serve ad imporre l'uscita da un costrutto diloop (`for`, `while`, `do-while`) o da `switch`, e a far riprendere il controllo di flusso immediatamente dopo la fine del costrutto.
Nel caso di costrutti annidati, `break` fa uscire solo dal costrutto piu interno entro il quale detta istruzione si trova.

4.4.2 `continue`

`continue` si può applicare solamente ai cicli; essa interrompe l'esecuzione del ciclo ma anzichè uscirvi porta alla successiva iterazione, ovvero:

- nel `while` o `do-while`, un `continue` nel corpo del ciclo fa saltare all'espressione di controllo (che verrà rivalutata e da li si procederà o meno con il loop);
- nel `for`, un continue fa eseguire l'espressione che effettua l'incremento, poi si passa alla valutazione della condizione, e di li all'esecuzione o meno del loop

Come per `break`, in caso di cicli annidati, si porta alla successiva iterazione solo il ciclo più interno, dove è avvenuta la chiamata alla continue.

Capitolo 5

Funzioni e struttura dei programmi

I programmi di C sono composti da una funzione `main` che chiama a turno altre funzioni (delle librerie o definite dall'utente). Le funzioni permettono di racchiudere una porzione di codice dedicato allo svolgimento di un compito specifico in modo tale che quella funzionalità sia riutilizzabile in più punti del programma.
Sono tre le fasi salienti di una funzione:

1. dichiarazione
2. definizione
3. chiamata

5.1 Dichiarazione e scope

Nel C tutti gli identificatori debbono essere dichiarati prima di poter essere utilizzati: questo vale sia per le variabili che per le funzioni.
Il campo di visibilità (**scope**) di un nome/identificatore corrisponde alla parte di programma in cui quel nome può essere usato. Abbiamo già visto lo scope per le variabili; lo **scope di una funzione** si estende dal punto in cui questa è stata dichiarata fino alla fine del file.

Nel caso delle funzioni, la dichiarazione specifica al compilatore l'**interfaccia**, ovvero:

- identificatore/nome della funzione;
- numero e tipo di parametri presi in input[1];
- tipo restituito in output

La dichiarazione può essere effettuata mediante il **prototipo**

[1] I nomi dei parametri non sono obbligatori, ma è meglio porli per motivi di documentazione

```
1 int power(int base, int n);
```

Il prototipo permette al compilatore di svolgere una **funzione di controllo** sulle chiamate alla funzione: è dal prototipo che il compilatore conosce il tipo di dato restituito dalla funzione, il numero di parametri, il loro tipo e il rispettivo ordine.

In **assenza del prototipo** la definizione della funzione viene supplita automaticamente dal compilatore; le informazioni sulla funzione vengono però prese dalla sua prima occorrenza, sia essa la definizione o una sua chiamata.
Di default il compilatore assumerà che la funzione restituisca un `int` e non presumerà nulla riguardo agli argomenti (numero, tipo, ordine dei tipi). Pertanto se gli argomenti passati non sono corretti, gli errori non saranno individuati dal compilatore.

In merito al **posizionamento del prototipo**, a contrario della definizione di funzione (che non può risiedere all'interno di un'altra definizione), un prototipo può risiedere **all'esterno** (come normalmente è) **o all'interno di una definizione** di funzione:

- un prototipo esterno a qualsiasi funzione si applicherà a tutte le relative invocazioni che appariranno nel file dopo il prototipo;

- se il prototipo della funzione `a` viene posto all'interno della definizione di un'altra funzione (`b`), sarà possibile chiamare la prima solo dall'interno della seconda.
  O meglio, le altre funzioni che invocheranno `a` non disporranno di un prototipo da analizzare, che verrà pertanto definito in automatico dal compilatore e questo potrà produrre errori rilevati in sede di compilazione.
  Questo può essere utile se abbiamo una funzione strumentale ad un'altra: servendo soltanto a questa non è necessario che sia visibile alle altre funzioni.

5.2 Definizione

La definizione di funzione è la sezione del codice dove risiede l'**implementazione** della funzione stessa.
Le definizioni di funzioni possono apparire in qualsiasi ordine e all'interno di uno o più sorgenti, benchè nessuna possa esser spezzata e suddivisa in file sorgente diversi, ne una funzione possa essere definita all'interno di un'altra.
La **sintassi** della definizione di funzione è:

```
tipo nome( lista parametri )
{
  dichiarazioni
  istruzioni
}
```

Dove a parte le `dichiarazioni` interne al blocco di variabili necessarie per il funzionamento e le `istruzioni` che specificano cosa la funzione fa e come lo faccia, abbiamo:

- `nome` è un qualsiasi identificatore valido utilizzato per indicare la funzione[2];
- `tipo` si riferisce al dato restituito come output al chiamante (al sistema operativo nel caso di `main`) [3], a `main` o ad un'altra funzione).
  Se non viene ritornato nulla al chiamante il `tipo` dell'intestazione deve essere `void`; il valore prodotto dalla valutazione della funzione potrà esser un valore di scarto (quindi non bisogna effettuare assegnazioni).
  Se invece il `tipo` non è specificato (cosa sconsigliata) il compilatore assumerà `int`;
- `lista parametri` è un insieme di coppie `tipo-identificatore` separati da virgole che specificano i dati di input della funzione; qualora la funzione non riceva alcun input, la lista di parametri sarà `void`.
  Distinguiamo **parametro** ed **argomento**: per parametro si intende una variabile dell'elenco nell'intestazione della funzione. Si dirà invece argomento il valore usato nella chiamata di funzione per quel dato parametro. Possono costituire argomento delle chiamate costanti, variabili o espressioni.

La prima linea della definizione viene chiamata l'**intestazione** della funzione; ad esempio una funzione che calcoli una potenza data base ed esponente avrà come intestazione

```
int power( int base, int n)
```

I nomi utilizzati per i parametri (`base` ed `n` in questo caso) sono circoscritti al corpo della funzione (variabili locali) e invisibili ad ogni altra funzione; possono esser riutilizzati anche all'esterno di `power` senza generare conflitti. Questo vale anche per le variabili dichiarate all'interno del corpo della funzione (anche esse sono locali).
L'istruzione **ritorna il controllo alla chiamata** quando:

- si giunge alla fine del blocco (ovvero alla parentesi graffa chiusa);
- si incontra l'istruzione `return`: essa serve per ritornare il controllo e restituisce l'espressione che segue al chiamante[4].
  Nel caso in cui la funzione non debba ritornare i calcoli dell'elaborazione, può essere di utilità fornire info sull'esecuzione del programma: tipicamente la restituzione del valore zero indica la normale termnazione della funzione, mentre valori diversi indicano errori o anormalità.

5.3 Chiamata

La funzione viene utilizzata nel programma attraverso una chiamata; la sintassi della chiamata è

```
nome( expr1, expr2, ...)
```

[2]Una volta che la definizione è avvenuta all'identificatore della funzione sarà associato l'indirizzo di memoria dove inizia il codice della funzione.

[3]Per i sistemi Unix, una volta che il programma è stato eseguito è possibile stampare il valore ritornato mediante `echo $?`

[4] Generalmente, non necessariamente, `return` è posta come ultima istruzione del corpo della funzione.

dove:

- `nome` è il nome della funzione desiderata;
- `expr*` sono le espressioni (o variabili o costanti) fornite come argomento della funzione.

Detta chiamata poi può costituire istruzione a sè (se si aggiunge alla sintassi della chiamata un punto e virgola) o fare parte di un'altra espressione.
I tipi degli argomenti `expr*` saranno coercizzati ai tipi specificati dal prototipo: come di consueto, se il numero di bit è inferiore si avrà una promozione, altrimenti una perdita di bit.
Il passaggio di valori alle funzioni verrà affrontata nel capitolo sui puntatori.

5.4 Altri argomenti

5.4.1 Elenchi di argomenti di lunghezza variabile

Vogliamo creare una funzione per calcolare la media di un numero di variabili non conosciuto a priori: per realizzare una funzione del genere occorre implementare funzioni con liste di argomenti di lunghezza variabile.
Ciò viene ottenuto mediante l'**ellissi** `...` posto tra gli argomenti. Essa:

- indica che la funzione riceve un *numero variabile di argomenti di qualsiasi tipo*;
- deve essere posta come ultimo parametro tra quelli specificati tra parentesi;
- deve essere preceduta da almeno un argomento con nome (l'argomento normale di funzione che abbiamo visto sino ad adesso).

Il **prototipo** sarà

```
double mean(int nofelem, ...);
```

Per creare questo tipo di funzioni occorre includere `<stdarg.h>`, che contiene i tipi di dato e le macro necessarie, ovvero:

- il tipo `va_list` serve a dichiarare una variabile che, di volta in volta, si riferisce a ciascun argomento. Chiamiamo questa variabile `ap` (argument pointer);
- la macro `va_start` inizializza la variabile `ap`, facendo si che punti al primo argomento;
- ogni invocazione della macro `va_arg` restituisce un argomento e modifica `ap` affinchè punti all'argomento successivo. Come secondo argomento di `va_arg` bisogna specificare un tipo, per decidere il tipo con cui la macro restituisce l'argomento in questione;
- `va_end`, che deve essere obbligatoriamente chiamata prima che la funzione termini, compie le operazioni di pulizia necessarie.

L'**ultimo argomento con nome** prima di `...` serve a `va_start` come punto di riferimento per individuare gli argomenti anonimi.

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  double mean(int nofelem, ...);
5
6  int main (void) {
7
8    printf("%.2f\n", mean(3, 1. , 2. , 3. ) );
9    printf("%.2f\n", mean(3, 1. , 5. , 15.) );
10
11   return 0;
12 }
13
14 double mean(int nofelem, ...){
15
16   double sum = 0.;
17   int i;
18
19   va_list ap;
20   va_start(ap, nofelem);
21
22   for(i=1; i<=nofelem ; i++) {
23     sum += va_arg(ap, double);
24   }
25
26   va_end(ap);
27
28   return sum/nofelem ;
29
30 }
```

Queando non vengono forniti elementi destinati all'ellissi, le macro di `<stdarg.h>` fanno comunque funzionare il tutto.

5.4.2 Chiusura dei programmi con `exit` e `atexit`

La libreria `stdlib.h` fornisce alcuni metodi per terminare l'esecuzione del programma in maniera alternativa al ritorno della funzione `main`.

exit La funzion `exit` forza la chusura del programma; è spesso usata per terminare in presenza di errori.
Exit riceve un argoment, normalmente le costanti simboliche :

- `EXIT_SUCCESS`: sarà restituito all'ambiente chiamante il valore definito dall'implementazione per la chiusura con successo
- `EXIT_FAILURE`: sarà restituito il valore definito dall'implementazione per la chiusura con fallimento

Nel momento in cui verrà chiamata `exit`:

- le funzioni registrate con `atexit` saranno invocate
- saranno svuotati e chiusi tutti gli stream associati al programma
- il controllo ritornerà all'ambiente ospite

atexit La funzione atexit registra una funzione (fino ad un massimo di 32) da invocare subito dopo la chiusura con successo del programma (per il ritorno di `main` o l'effetto di `exit`).
Essa riceve come argomento un puntatore a una funzione (il nome della stessa): la funzione richiamata alla chiusura non può avere argomenti ne restituire un valore.
Se vengono registrate più funzioni queste verranno eseguite in uscita nell'ordine inverso a quello della registrazione.

Capitolo 6

Array e puntatori

Le **strutture di dati statiche** sono sia strutture di dati complesse che una volta create mantengono le proprie dimensioni durante l'esecuzione del programma. Il C mette a disposizione, le seguenti[1]:

- **array**: è un insieme di locazioni di memoria atte a contenere **informazioni dello stesso tipo base**, cui è possibile accedere mediante un **nome comune** ed uno o più **indici**. Un *vettore* è un array ad una dimensione: basterà pertanto un solo indice per selezionare gli elementi al suo interno. Una *matrice* è un array a due dimensioni (servono due indici). Array con più di due dimensioni sono meno frequenti ma possibili;
- **strutture** (in senso stretto, o `struct`): sono insiemi di informazioni accessibili mediante un **nome comune**; a differenza degli array **non necessariamente** le singole **componenti** debbono essere dello **stesso tipo**.

Qualora si rendano necessarie strutture di dati la cui dimensione (capacità di immagazzinamento) vari a seconda delle esigenze bisogna ricorrere alle **strutture di dati dinamiche**, le quali si basano grandemente sull'utilizzo dei **puntatori** e `struct`, oggetto del prossimo capitolo.

6.1 Array

Qui ci si soffermerà sui vettori: data l'omogeneità con gli array a 2 o più dimensioni ci si limiterà in seguito ad evidenziarne le differenze.

6.1.1 Dichiarazione e inizializzazione

La **dichiarazione** di un vettore segue la forma:

```
tipo nome_vettore[ dimensione ];
```

dove:

- `tipo` è il tipo base degli elementi costituenti il vettore;

[1] Tra le strutture di dati complesse statiche complessi si annoverano anche le *unioni*, che tralasciamo per ora.

- `dimensione` è una costante numerica intera che indica quanti elementi l'array deve contenere.
  Spesso dimensione è specificata mediante una costante creata mediante `define` in maniera tale da render facilmente scalabile la dimensione del vettore a seconda delle esigenze.

Ad esempio:

```
int vettore[10];        /* un vettore di 10 interi */

char stringa[100];      /* un vettore di 100 caratteri:
                           potenzialmente una stringa di 99 caratteri
                           piu' il carattere terminatore '\0' */
```

La dichiarazione serve al compilatore per riservare in memoria spazio sufficiente al vettore. Lo spazio occupato dal vettore in *byte* sarà:

$$dim = \texttt{sizeof(tipo)} \cdot \texttt{dimensione}$$

Fino a che il vettore non è inizializzato, contiene **valori spazzatura**; l'utilizzo di un vettore in tali condizioni può provocare errori al programma.

L'**inizializzazione** può essere svolta insieme alla dichiarazione, o in una fase successiva (mediante la semplice scrittura degli elementi del vettore). Nel primo caso la dichiarazione presenta la forma:

```
int vet[3] = {1,2,3};
```

ovvero si fa seguire alla dichiarazione un segno di uguale e delle parentesi graffe contenenti una **lista di inizializzatori**, separati da virgole.
Nel caso ci fossero **meno inizializzatori degli elementi del vettore**, quelli rimanenti sarebbero inizializzati a 0. Ad esempio per inizializzare un vettore a zero basta la seguente istruzione:

```
int n[10] = {0};
```

Infine qualora in una dichiarazione con lista di inizializzazione sia **omessa la dimensione** del vettore il numero di elementi sarà determinato da quello degli inizializzatori specificati. Ad esempio

```
int n[] = {1,2,3}
```

crea un vettore di tre elementi.

A volte si vedono nei blocchi delle funzioni **vettori di grandi dimensioni dichiarati mediante** `static`:

```
static int foo[100000]
```

Questo serve per fare in modo che tale vettore non sia creato e inizializzato ogni volta che la funzione verrà invocata. Ciò riduce il temo di esecuzione del programma, soprattutto per quei programmi che prevedono frequenti invocazioni di funzioni che dichiarano vettori di dimensioni ragguardevoli.
Il vettore `static` verrà *inizializzato* una volta sola durante la compilazione: se il programmatore non specifica altrimenti nel sorgente sarà inizializzato a 0.

6.1.2 Accesso

Il C indicizza gli elementi facenti parte del vettore, ai fini dell'accesso in lettura o scrittura, partendo da 0 (primo elemento), sino a (`dim` − 1) (ultimo elemento del vettore).

```
+-----------------------------+
| 0 | 1 |    ...    | dim - 1 |
+-----------------------------+
```

Pertanto è importante distinguere il terzo elemento del vettore dall'elemento con l'indice 3.
L'accesso in lettura o scrittura all'array, si effettua mendiante il nome dello stesso, seguito dall'indice racchiuso tra parentesi quadre: un **indice** può esser costituito da costante intera o espressione che produce un intero.
Alcuni esempi di sintassi di accesso ad un elemento del vettore:

```
vettore[3]     /* accede al _quarto_ elemento di "vettore" */
stringa[2-2]   /* accede al _primo_ carattere di "stringa" */
```

Pertanto:

- per **scrivere un elemento** del vettore basterà effettuare un'assegnazione utilizzando la sintassi di accesso come *lvalue*;

- per **leggere un elemento** contenuto in un vettore basterà valutarne la sintassi d'accesso (non a sinistra di un'assegnazione).

La lettura o scrittura di un vettore nel suo complesso si effettua mediante un ciclo (`for` tipicamente) che a turno effettua l'elaborazione (lettura, scrittura) per tutti i suoi elementi.

6.1.3 Sconfinamento

Bisogna assicurarsi che quando si scorre un vettore in lettura o scrittura il suo indice non scenda mai al di sotto dello 0 e sia sempre inferiore (di una unità) al numero degli elementi che compongono vettore, altrimenti si verificherà uno **sconfinamento**.
Il C non effettua controlli per verificare ed evitare che non si acceda a posizioni fuori dal vettore; accedere ad una locazione di memoria fuori dal vettore è un errore tanto grave quanto comune. Le conseguenze:

- se si accede in **lettura** può accadere, a seconda del sistema operativo, un errore logico o una violazione d'accesso (`segmentation fault`);

- se invece si accede in **scrittura** ci saranno danni maggiori, perché oltre ai due problemi già citati si sovrascriverà una locazione di memoria che potrebbe contenere dati essenziali al programma (si modifica il valore delle variabili dichiarate immediatamente sopra o sotto al vettore) o peggio al sistema operativo (nel caso peggiore si può arrivare al blocco del sistema).

6.1.4 Stringhe - Vettori di caratteri

Presentano alcune peculiarità. Possono esser (dichiarate e) inizializzate in due modi: utilizzando una costante stringa o con una lista di inizializzatori di costanti letterali:

```
char string1[] = "first";
char string1[] = {'f','i','r','s','t','\0'};
```

La dimensione del vettore `string1` sarà in questi casi determinata dal compilatore, in base alla lunghezza della stringa (5 caratteri + 1 di terminazione = 6 caratteri). Tutte le stringhe del C terminano con il carattere nullo.

Il vettore di caratteri può anche essere solo dichiarato:

```
char string2[10]
```

In tal caso la dimensione dovrà esser sufficiente a contenere il numero di caratteri della stringa che è destinato a contenere più un carattere per la terminazione. Se la lunghezza del vettore sarà insufficiente si verificherà un problema di sconfinamento.

6.1.5 Matrici ed array multidimensionali

Gli array a 2 (o più) dimensioni sono vettori, in cui elementi sono vettori(i cui elementi sono vettori ... fino ad esaurire le dimensioni desiderate). Si faranno esempi con le matrici, il più semplice esempio di array.

La **dichiarazione** di una matrice di interi avviene, se `n.row` è il numero di righe ed `n.col` quello delle colonne, come segue:

```
tipo nome_matrice[ n.row ][ n.col ];
```

Nella dichiarazione le dimensioni delle matrici devono essere delle costanti.

In sede di dichiarazione è possibile **inizializzare** le matrici come segue:

```
int matrice[2][3] =
     {
          {1,2,3} ,
          {4,5,6}
     };
```

I valori sono raggruppati per riga all'interno di parentesi graffe; i valori della prima lista di inizializzazione serviranno per la riga 0, quelli della seconda per la riga 1. Nell'eventualità che per una data riga non sia stato fornito un numero sufficiente di inizializzatori, gli elementi rimanenti saranno inizializzati a 0, come nel caso dei vettori.

Le matrici in C sono locazioni di memoria contigue che contengono i dati memorizzati per righe (come in pascal, mentre il fortran memorizza per colonne): l'indice di destra (quello della colonna) è quello che cambia più velocemente. Come detto, anche se viene pensata come un oggetto a due dimensioni,

```
+-----------+
| 1 | 2 | 3 |
+-----------+
| 4 | 5 | 6 |
+-----------+
```

è in realtà un oggetto disposto linearmente in memoria, riga dopo riga, come un vettore

```
||-----------||-----------||
|| 1 | 2 | 3 || 4 | 5 | 6 ||
||-----------||-----------||
```

Indicizzando righe e colonne a partire da 0, se `NR` è il numero di righe, ed `NC` è il numero di colonne, allora l'elemento in posizione `(r,c)` della matrice sta fisicamente nella posizione

$$r \cdot NC + c$$

del vettore.

L'**accesso** al dato in posizione `(r,c)` viene effettuato mediante `[r][c]`. Ad esempio:

```
matrice[1][3] = 7;
```

scrive 7 nell'elemento alla seconda riga quarta colonna. Analogamente ai vettori, il nome della matrice rappresenta il puntatore al primo elemento della matrice, in questo caso alla prima riga.
Spesso nell'elaborazione delle matrici si utilizzeranno doppi cicli `for` (uno per passare in rassegna le righe, l'altro per le colonne).

6.2 Puntatori

La memoria (RAM) di una macchina è rappresentabile mediante una collezione di celle o locazioni di memoria numerate consecutivamente da *indirizzi*.
La stampa di indirizzi di memoria avviene solitamente in notazione esadecimale per compattezza, essendo, su una macchina a 32 bit le celle indirizzabili 2^{32}.

```
+----------------------+
| 0x1 | 0x2 | 0x3 | ...|
+----------------------+
```

All'atto della dichiarazione di una variabile tot celle (in base ai byte necessari per il tipo di dato) vengono allocate; queste verranno in seguito riempite, al momento dell'inizializzazione della variabile stessa.
Pertanto in C **ogni variabile** si caratterizza per avere:

1. un **indirizzo** di memoria;
2. un **valore** contenuto in quella locazione di memoria.

Un **puntatore** è una *variabile che contiene un indirizzo di memoria*; precisamente l'intero che rappresenta il numero progressivo della cella di RAM del primo byte della variabile.
Attraverso il deferimento della variabile puntatore è possibile utilizzare (accedere o modificare) il valore di quest'ultima.
Il puntatore è caratterizzato da un **tipo** che serve più che altro per assicurare operazioni corrette nell'aritmetica dei puntatori, come si vedrà.

6.2.1 Dichiarazione, inizializzazione e deferimento

Dichiarazione L'istruzione:

```
tipo *p;
```

un puntatore `p` ad una variabile di tipo `tipo`. Ad esempio se `cPtr` dovrà essere un puntatore all'`int c`:

```
int c=0;    /*dichiarazioni staccate */
int *cPtr;

int *cPtr, c=0 ;   /* dichiarazione congiunta */
```

Le istruzioni di sopra dichiareranno la variabile cPtr di tipo `int *` (ovvero un puntatore a valore intero). Anche la variabile `c` sarà un intero, ma non un puntatore (l'asterisco della dichiarazione sarà applicato solo a `cPtr`, ogni puntatore deve avere il proprio asterisco davanti).

La dichiarazione di un puntatore specifica il tipo dell'oggetto a cui il puntatore punta, affinché il compilatore sappia le dimensioni di tale variabile (questo è necessario per l'aritmetica dei puntatori).
Ogni puntatore deve esser dichiarato per puntare ad oggetti di un determinato genere, ad uno specifico tipo di dati. L'eccezione è il puntatore a `void` (che si vedrà in seguito), il quale si usa per contenere qualunque tipo di puntatore ma non consente il deferimento.
Quando un puntatore viene dichiarato non punta a nulla, o meglio poiché il contenuto di una cella di memoria è casuale fino a che non viene inizializzata ad un valore noto, il puntatore punta ad una locazione di memoria casuale, che potrebbe non essere accedibile dal processo. Prima di scrivere (soprattutto, più che leggere) occorre inizializzare il puntatore.

Inizializzazione Può avvenire in sede di dichiarazione o dopo mediante un normale assegnamento. Un puntatore può contenere:

- la costante `NULL`
- con un indirizzo;

Nel primo caso non si farà riferimento a nessun dato (puntatore vuoto): `NULL` è un puntatore `void` a 0, ovvero la prima cella di memoria della ram, dove il computer non alloca mai niente. Non è possibile dereferenziare un puntatore a `NULL` (viene dato errore a runtime).
Per inizializzare con un indirizzo di qualcosa residente in memoria occorre un operatore che applicato a quel qualcosa restituisca il suo indirizzo. Questo è l'operatore unario `&` ; pertanto l'istruzione

```
p = &c;
```

assegna l'indirizzo di `c` alla variabile `p`. Si dice che "`p` punta a `c`".
L'operatore `&` su applica solo ad oggetti *riposti in memoria*: variabili ed elementi di vettori. Non può applicarsi a espressioni, costanti o variabili di classe register.
Per vedere quale indirizzo ha la variabile `c`, possiamo sfruttare la specifica `%p`[2] di `printf`:

[2] Da warning perché sarebbe dedicata ad un puntatore di tipo `void`, ma l'eseguibile funziona lo stesso

```
printf("%p= %p =%d\n", &c, p, c);
l@a6k:~$ ./a.out
0xbfc1b58c=0xbfc1b58c=0
```

Per evitare di lasciare il puntatore inizializzato è opportuno inizializzarlo a `NULL` in sede di definizione.

Deferimento L'operatore unario `*` è detto di deferimento; se applicato a un puntatore, permette l'accesso all'oggetto indicato per lettura o scrittura. Ciò significa che se `ip` punta all'intero `x`, allora `*ip` può comparire dovunque lo possa fare `x` (sono perfettamente sostituibili)

```
int c=0, *cPtr;
int d;
cPtr = &c;

*cPtr += 2;                  /* Aumentiamo c di 2 */
++(*cPtr);                   /* Aumentiamo c di 1 */
d = *cPtr;                   /* Assegnamo a d c (3) */
printf("%d", *cPtr)          /* Stampiamo il valore di c */
```

Le parentesi nella penultima istruzione potevano essere omesse perché a parità di precedenza degli operatori $++$ e * si procede da dx a sx (poste per leggibilità); non sarebbe vero il contrario, nel caso `*cPtr++` (che aumenterebbe `cPtr` per poi deferire il nuovo indirizzo).
Infine i puntatori, in quanto variabili, possono essere utilizzati senza ricorrere al deferimento. Ad esempio

```
cPtr = dPtr
```

avrà l'effetto che (posto che `cPtr` punti a `c` e `dPtr` a `d`) `cPtr` punti a `d`.

6.2.2 Operazione sui puntatori

Le operazioni valide con i puntatori sono:

- assegnamenti fra puntatori dello stesso tipo (o assegnamenti a puntatori di tipo `void`);
- somme o sottrazioni di un puntatore e un intero;
- sottrazioni o confronti di due puntatori a elementi dello stesso vettore.

6.2.2.1 Aritmetica dei puntatori

È possibile addizionare o sottrarre interi ai puntatori (utilizzando gli operatori `+`, `-`, `++` e `-`).
L'effetto è di far aumentare (o diminuire) il puntatore, e farlo puntare ad una locazione di memoria diversa. Nello specifico, il *risultato* numerico di una operazione su un puntatore dipende dalle dimensioni del tipo di dato cui esso si riferisce, specificato nella dichiarazione del puntatore: in particolare sommare una unità ad un puntatore significa spostare in avanti il puntatore nella memoria di un numero di byte corrispondenti alle dimensioni del dato puntato.

Ad esempio se il puntatore punta ad un `char`, poichè esso ha dimensione 1, l'istruzione p++ aumenta di un byte l'indirizzo contenuto nel puntatore. Se è uno `short int` di 2, e così via. Nel caso di puntatori a void (es. `void *p`) il comportamento è uguale al caso di `char`; si aumenta o diminuisce a passi di un byte.

In generale se p è un puntatore ad un certo tipo (`tipo *p;`) e contiene un certo valore `addr` l'espressione `p[k]` accede all'area di memoria che parte dal byte `addr+k*sizeof(tipo)` ed ha dimensione `sizeof(tipo)`. Nel caso di un puntatore a `void`, viene considerata 1 la dimensione del dato puntato, cioè il puntatore punta ad un byte.
Anche per i vettori l'accesso ai dati avviene secondo queste modalità `vet[k]`, perché in C, il nome di un array è trattato dal compilatore come un puntatore costante alla prima locazione di memoria dell'array stesso.

L'aritmetica dei puntatori è più che altro utile per avanzare in strutture di dati che sono composte da celle di memoria contigue e riempite con dati della medesima dimensione (stesso tipo): in altre parole, gli array. Non serve nel caso si applichi a variabili: anzi può essere controproducente, portandoci in zone della memoria di cui non si conosce il contenuto. `gcc` infatti segnala errore in presenza di codice del genere:

```
int c=0, *cPtr;
cPtr = &c;
printf("%d",  *(c++));

error: invalid type argument of ‘unary *’ (have ‘int’)
```

Se invece sappiamo che un vettore di una data lunghezza composto da elementi di un determinato tipo (e medesimo numero di byte per le singole componenti), sappiamo che se abbiamo l'indirizzo del primo byte del primo elemento del vettore memorizzato in un puntatore, e aggiungiamo una unità a tale valore, l'indirizzo di memoria che otterremo sarà quello del primo byte del secondo valore. Ovvero il puntatore modificato punterà al secondo elemento del vettore.

6.2.2.2 Confronto e sottrazione tra puntatori

È possibile *confrontare* tra loro puntatori a diverse cose per determinare quale si trovi prima o dopo all'interno della memoria (può essere utile nel caso degli array per determinare quale elemento viene prima dell'altro considerato).

La *sottrazione tra due puntatori* facenti parte di un medesimo array serve per determinare il numero di elementi esistenti fra i due:

```
float a[4]= {1.1, 2.2, 3.3, 0} ;
printf("%d\n", &a[3] -  &a[0] );

l@a6k:~$ ./a.out
3
```

6.2.3 Valutazione, valori e indirizzi

Come detto gli oggetti sono caratterizzati da due cose, un indirizzo di memoria dove è memorizzato o il valore contenuto. Il compilatore però interpreta in

maniera diversa diversi oggetti; nel senso che la valutazione di alcuni produrrà il valore, mentre quella di altri il loro indirizzo.
In particolare, producono il **valore** contenuto:

- una variabile;
- l'elemento di un vettore

per ottenere l'indirizzo di questi oggetti occorre applicare l'operatore `&`.
Al contrario restituiscono di default un **indirizzo**:

- il nome (da solo) di un vettore: restituisce l'indirizzo al suo primo elemento;
- le costanti stringa; sono vettori di `char` che restituiscono l'indirizzo al primo elemento;
- il nome delle funzioni.

Ad esempio:

```
int a[10] ;
printf("%p = %p\n",  a , &a[0]);
printf("%p\n",  printf );

l@a6k:~$ ./a.out
0xbfeacecc = 0xbfeacecc    /* l'indirizzo del vettore*/
0x8048300                  /* l'indirizzo della funzione */
```

Pertanto il nome di un vettore funziona similmente a un puntatore: la differenza principale è che i puntatori sono variabili che contengono un indirizzo, e questo contenuto può essere cambiato per puntare ad un'area di memoria diversa.
Invece il nome dei vettori è trattato come un *puntatore costante* all'inizio del vettore stesso (ovvero non posso assegnare qualcosa al nome del vettore ma solo alle sue posizioni); pertanto questo indirizzo non può essere modificato, è costante (ad es: il compilatore da errore se `int vet[100]; vet++; vet=10;`)

La relazione tra vettori e puntatori è tale che ogni operazione ottenibile tramite l'accesso a un vettore per mezzo di un indice può essere realizzata anche con i puntatori. L'impiego di questi garantisce generalmente *maggiore velocità d'esecuzione*.

Quindi un riferimento ad `a[i]` si può scrivere come `*(a+i)`; nel valutare `a[i]`, nella forma di *vettore e indice*, il C la trasforma in `*(a+i)`, detta in forma di *puntatore e offset*. La stessa cosa la si può fare, nell'ipotesi che `aPtr` sia puntatore alla prima cella di `a` mediante l'espressione `*(aPtr+i)` o anche attraverso `aPtr[i]`.

6.3 Funzioni, puntatori e array

6.3.1 Chiamate per valore o per riferimento

Il *passaggio di valori* come argomento alle funzioni può avvenire in due maniere:

- per **valore**: se si passa un qualcosa per valore verrà creata una copia dell'oggetto che sarà utilizzata nelle elaborazioni (lettura o modifica) all'interno della funzione. La variabile originale non sarà toccata pertanto una volta usciti dalla funzione il suo valore non sarà stato modificato da eventuali scritture;
- per **riferimento**: il chiamante da la possibilità alla funzione chiamata di modificare la variabile originale, indicandone l'indirizzo).

Le chiamate per valore dovrebbero esser utilizzate ogniqualvolta la funzione chiamata non abbia la necessità di modificare il valore della variabile originale definita dal chiamante; ciò preverrà effetti collaterali di accidentali modifiche all'interno della funzione chiamata.
Nel caso in cui sia necessario modificare un oggetto (ad esempio ordinare un vettore) oppure nel caso di oggetti di dimensioni elevate (la cui copia impegnerebbe notevoli risorse) si può effettuare una chiamata per riferimento. Per richiamare una funzione con degli argomenti che devvano esser modificati si passeranno i loro indirizzi.

Come affermato nel capitolo delle funzioni è possibile effettuare chiamate per riferimento. Vediamone alcuni utilizzi con variabili e vettori.
Una funzione che cambi di valore a due *variabili* passate dalla chiamante avrà prototipo:

```
void swap(int *xPtr, int *yPtr);
```

Si noti che è stato inserito l'asterisco per specificare ciò che verrà inizializzato in ingresso sono degli indirizzi alle variabili. All'interno della funzione, come al solito, si utilizzerà sempre `*xPtr` per deferire. Nella chiamata occorrerà passare un puntatore alla variabile o valutarne l'indirizzo mediante `&x`.
Se invece la funzione deve operare sul vettore (esempio determinazione lunghezza di una stringa) è possibile specificare prototipo e intestazione in due maniere alternative. Nella notazione del vettore

```
int strlen( char s[] )
```

oppure in quella di puntatore

```
int strlen( char * s )
```

La preferenza va a questo secondo perché è più coerente con ciò che effettivamente viene passato in input nella chiamata alla funzione: ovvero il nome del vettore, un indirizzo.

Come nota di margine, è anche possibile passare alle funzioni solo una parte di vettore, semplicemente si specifichi il relativo indirizzo:

```
strlen( str+2 )      /* Si parte dal terzo elemento ... */
```

6.3.2 L'utilizzo di `const` con puntatori e funzioni

Il passaggio per riferimento è efficiente poiché in memoria non si devono creare copie; tuttavia espone alla possibilità di modifiche non desiderate. Per evitare questo è possibile utilizzare opportunamente il qualificatore `const` nell'intestazione e nel prototipo della funzione. Ad esempio, è possibile implementare:

- **puntatore variabile a dati variabili**: ad esempio per modificare una stringa dobbiamo lavorare con l'aritmetica dei puntatori (il puntatore non deve essere `const`) e poter modificare il vettore (che non deve essere `const`. Intestazione esempio:

  ```
  void funct(char * ptr)
  ```

- **puntatore variabile a dati costanti**: questa configurazione basta (principio del minimo privilegio) per visualizzare una stringa (che non deve essere modificata). Intestazione esempio:

  ```
  void funct(const char * ptr)
  ```

- **puntatore costante a dati variabili**: non si può modificare il puntatore, bensì la variabile cui punta. Dichiarazione esempio:

  ```
  void funct(int * const ptr);
  ```

- **puntatore costante a dati costanti**: minime facoltà di accesso (solamente lettura, sia per puntatore che per dati). Dichiarazione esempio:

  ```
  void funct(const int * const ptr);
  ```

A fare da spartiacque per l'interpretazione dell'argomento della funzione è il simbolo `*`: ciò che viene prima riguarda il dato, ciò che segue il puntatore

6.3.3 Puntatori a funzioni

Come si è detto, il nome di una funzione è in realtà un puntatore alla zona di memoria nella quale inizia il codice che eseguirà il compito della funzione.
I puntatori di funzione, ovvero variabili che contengono gli indirizzi delle funzioni servono per creare codice che si comporta diversamente a seconda delle evenienze. Una strategia alternativa consiste nel definire diverse funzioni che:

- prendono in input lo stesso tipo di parametri;
- restituiscono gli stessi valori

Al momento della chiamata si deciderà quale funzione scegliere mediante un puntatore alla stessa.

Essendo un puntatore a funzione una variabile come le altre, andrà altrettanto *definita* :

```
int (*functPtr)(float, char) = NULL;
```

Si è definito e inizializzato (a `NULL`) un puntatore ad una funzione che riceve due argomenti (un `float` e un `char`), e restituisce un `int`.

Una volta definite le funzioni (accomunate da numero e tipi di ingresso e da tipo di uscita, coerenti con la definizione del puntatore a funzione) la *memorizzazione di un indirizzo* della funzione nel puntatore può avvenire in due maniere:

```
functPtr =  funzione1
functPtr = &funzione2    /* Più portabile */
```

Sarà in seguito possibile utilizzare gli operatori di comparazione come visto per i puntatori ad altri tipi di dato.

Una volta che l'indirizzo è memorizzato nel puntatore, possiamo *utilizzare la funzione puntata* in due modi:

```
asd = ( * functPtr) (foo, bar)
asd = functPtr(foo, bar)
```

dove `foo` è il parametro di tipo `float` e `bar` di tipo `char`[3]; in `asd` (di tipo `int`) viene memorizzato il risultato restituito.

Questo è l'utilizzo "base" dei puntatori a funzioni; altre cose che si vedono/-possono tornare utili sono:

- passaggio come argomenti a e/o restituzione da funzione;
- array di puntatori a funzioni.

Il *passaggio* ad una funzione si effettua impostando una intestazione del tipo

```
void PassPtr (int (*functPtr) (char, float))
```

Passandolo in una funzione, il puntatore (funzione puntata) potrà essere eseguita dall'interno di questa.
Per *ritornare un puntatore a funzione* la notazione è un po' più complessa; la cosa più semplice è definire in anticipo mediante `typedef` il tipo ritornato, per poi restituirlo nella funzione. La `typedef` sarà

```
typedef int (*i_functPtr_cf) (char, float);
```

ad esempio per rendere sinonimo `f_functPtr_cf` di un puntatore a funzione che prende in input un char e un float e restituisce un `int`. Dopo `typedef`, l'intestazione della funzione (nonché il prototipo) potranno aver la forma:

```
i_functPtr_cf nome( argomenti )
```

Infine tornano utili gli array di puntatori a funzioni; una volta utilizzato `typedef` come avvenuto in precedenza, è immediato dichiarare ed inizializzare un vettore del genere:

```
i_functPtr_cf vettore[10];
vettore[0] = &fnz1;
vettore[2] = &fnz2;
...
```

Un classico esempio di utilizzo dei puntatori a funzione è l'algoritmo di sort in grado di ordinare sulla base della scelta dell'utente (crescente vs. decrescente, legato a due funzioni distinte): vedi k&r pag. 117 e *the function pointer tutorial*, sezione 3.

[3] Non si sono posti i nomi dei parametri per non appesantire la riga.

6.4 Altri argomenti

6.4.1 Puntatori a caratteri

Una costante stringa, come detto, quando viene valutata restituisce il puntatore alla prima occorrenza.

```
char *pmessage
pmessage = "now is the time"
```

Come regola è opportuno utilizzare `pmessage` solo ed esclusivamente per la lettura della stringa cui punta, non per la scrittura (es modifica di elementi). Vi è una differenza tra

```
char amessage[] = "now is the time";
char * pmessage = "now is the time";
```

`amessage` è un vettore di grandezza sufficiente per contenere la stringa con tanto di carattere di terminazione. Singoli caratteri del vettore possono esser cambiati, ma `amessage` farà riferimento alla stessa zona di memoria.

```
          +---+   +-----------------+
pmessage: | o-|-->| now is the time |
          +---+   +-----------------+
          +-----------------+
amessage: | now is the time |
          +-----------------+
```

`pmessage` è un puntatore inizializzato per puntare a una costante stringa; esso potrà esser modificato per denotare qualcos'altro (ad esempio si memorizza un diverso messaggio a seconda del flusso nella medesima variabile), ma il risultato è indefinito se si tenta di cambiare i contenuti della stringa. La lettura da suddette stringhe è ok: così funziona tra l'altro printf. Tuttavia per la scrittura su stringa è opportuno memorizzare detta stringa in un vettore. Infatti se si cerca di cambiare i contenuti di stringa di cui si possiede solo il puntatore all'elemento iniziale, il risultato è indefinito. Ad esempio su `gcc`, il seguente codice produce `Segfault`

```
char *foo = "hello";
*(foo+2) = 'd';
```

È pertanto opportuno utilizzare opportunamente `const` con questo tipo di dati, per dare errore in sede di compilazione.

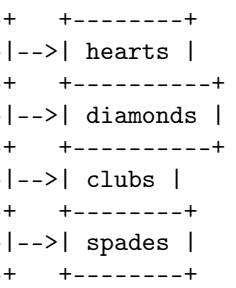
```
   const char *foo = "hello";
   *(foo+2) = 'd';
prova.c: In function ‘main’:
prova.c:7: error: assignment of read-only location ‘*(foo + 2u)’
```

6.4.2 Vettori di puntatori vs. array bidimensionali

I puntatori possono esser memorizzati in vettori, dal momento che sono essi stessi delle variabili. Un utilizzo tipico è la formazione di un vettore di stringhe (di sola lettura). Una dichiarazione esempio:

```
const char * suit[4] = {"hearts", "diamonds", "clubs", "spades"};
```

`suit[4]` dichiara un vettore di 4 elementi; `char *` dice che ogni elemento di `suit` è un puntatore a `char`.



Una forma di utilizzo:

```
for(i=0; i<4; i++)
   printf("%s\n", suit[i]);
```

L'utilizzo di vettori di puntatori permette di effettuare operazioni complesse che penalizzerebbero le prestazioni nel caso si utilizzassero array bidimensionali. Si confrontino

```
int a[10][20];
int *b[10];
```

`a[3][4]` e `b[3][4]` sono entrambi riferimenti ad un intero sintatticamente corretti.
Tuttavia nel primo caso vengono riservate 200 locazioni di memoria (ciascuna atta a contenere un intero) e l'elemento `a[r][c]` si trova nella posizione $20 \cdot r + c$. Nel secondo caso si stanzia spazio solo per 10 puntatori, non ancora inizializzati. L'inizializzazione deve avvenire esplicitamente. Ad esempio[4]:

```
   const int * numbers[2];
   int a[3] = {1, 3, 5};
   int b[2] = {7, 9};
   numbers[0] = a;
   numbers[1] = b;
   printf("numbers[1][1]=%d\n", numbers[1][1]);
   printf("numbers[0][2]=%d\n", *( *(numbers+0) +2) );
numbers[1][1]=9
numbers[0][2]=5
```

In questo caso si è utilizzato un vettore di puntatori a interi. La maggior parte delle volte i vettori di puntatori servono più che altro per le stringhe.

I principali vantaggi dei vettori di puntatori:

- non essendo stata riservata una struttura di memoria predefinita come avviene negli array, non è necessario che tutte le righe abbiano la medesima lunghezza per i vettori di puntatori (negli array tale struttura viene comunque riservata e se non sfruttata si ha uno spreco di memoria)

[4] Per adesso non ho trovato altri modi di inizializzare il vettore di puntatori ad interi; ad esempio mi da warning e infine segfault la dichiarazione `const int * numbers[2] = {{1, 3, 5},{7, 9} };`

- nel caso desideriamo ordinare delle stringhe puntate dal vettore invece di spostare fisicamente le stringhe (come avremmo fatto ad esempio in una array a due dimensioni con una parola per riga) spostiamo all'interno del vettore i rispettivi puntatori

6.4.3 Argomenti dalla riga di comando

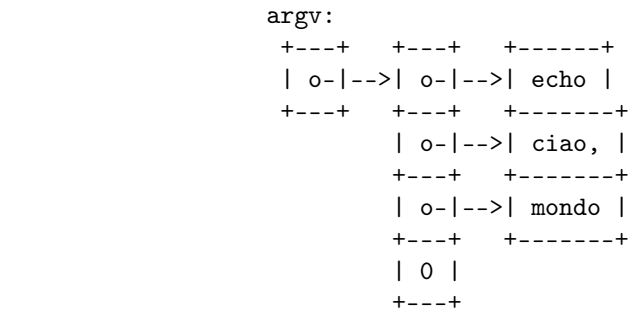
È possibile passare argomenti dalla riga di comando[5] al momento dell'esecuzione; in tal caso la funzione `main` presenta una intestazione del tipo

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

ricevendo gli argomenti:

- `argc` (*argument count*) è un intero che contiene il numero di parole che compongono il comando. È sempre almeno pari a 1;
- `argv` (*argument vector*) è un puntatore ad un vettore di puntatori che si riferiscono alle stringhe fornite. Per convenzione `argv[0]` punta al nome del comando; il primo argomento facoltativo è raggiungibile da `argv[1]` e l'ultimo è `argv[argc-1]`. In corrispondenza dell'ultimo elemento del vettore `argv` (`argv[argc]`) lo standard stabilisce sia posto il puntatore nullo `0`, in maniera tale che non sia possibile il deferimento.

Ad esempio se abbiamo compilato un binario `echo` e passiamo in input `ciao, mondo`, `argv` sarà così strutturata:



In funzione della corrispondenza tra vettori e puntatori, capita anche di vedere l'intestazione di main come

```
int main( int argc, char **argv )
```

Un esempio di utilizzo

```
   for(i=0; i<argc; i++)
      printf("%s\n", *(argv + i));
l@a6k:~$ ./a.out  caio sempronio
./a.out
caio
sempronio
```

[5] La libreria `getopt` permette una gestione più evoluta degli input di parametri.

6.5 Vettori e puntatori non sono la stessa cosa

Sebbene nella pratica ci si possa spesso riferire ad elementi interni ad un vettore mediante la notazione classifca del vettore o mediante puntatore più offset, bisogna apprezzare i rischi di equiparare.

Nello specifico una definizione e una dichiarazione come le seguenti porteranno ad errori

```
1 int mango[100];  /* Definizione, nel FILE 1 */
2 extern int *mango;  /* Dichiarazione in un altro file */
```

Qui nel primo file definiamo mango come un vettore, ma nel secondo lo dichiariamo come un puntatore a dati esterni (ovvero residenti in un altro file). Se usiamo il puntatore nel secondo file, se siamo fortunati il programma crasha e si blocca.

La **soluzione** per questo inconveniente sta nel **dichiarare e definire nello stesso modo i medesimi dati** (in file differenti).

Per capire perchè il problema si verifica dobbiamo apprezzare le differenze tra vettori e puntatori e dicordare la differenza tra **dichiarazione** e **definizione**. Gli oggetti nel C devono avere esattamente una definizione e possono avere molteplici dichiarazioni esterne:

- la **definizione** avviene in un unico posto, specifica il tipo di un oggetto, riserva spazio di memoria per i dati ed è usato per creare nuovi oggetti, ad esempio

  ```
  1     int my_array[100];
  ```

- la **dichiarazione** può occorrere più volte e serve a descrivere il tipo di un oggetto; è utilizzata per riferirsi ad oggetti definiti da qualche altra parte (altri file). Specifica come deve essere utilizzato un determinato simbolo all'interno del file

  ```
  1     extern int my_array[];
  ```

  qui non c'è bisogno di specificare la lunghezza, perchè possiamo non specificare al massimo un indice.

La differenza è che l'indirizzo di ciascun identificatore (es un array) è conosciuto a tempo di compilazione, mentre il contenuto (es l'indirizzo contenuto da un puntatore) è conosciuto solo in run time.

Se in un file scriviamo

```
1 char a[9] = "abcdefgh";
2 c = a[i];
```

nella tabella dei simboli supponiamo che `a` ha indirizzo 9980; gli step della seconda istruzione sono:

- ottieni il valore `i` e aggiungilo a 9980
- ottieni il contenuto dall'indirizzo `9980 + i`

Se scriviamo

```
1 char *p;
2 c = *p;
```

nella tabella dei simboli p ha indirizzo `4624`; gli step della seconda istruzione sono:

- ottieni il valore (indirizzo) contenuto in `4624` (es `5081`)
- aggiungi il contenuto dall'indirizzo `5081`

Se iniziamo a mischiare le cose

```
1 char *p = "abcdefgh";
2 c = p[i];
```

qua ancora andiamo bene perchè il puntatore `p` (es che ha indirizzo `4624`) contiene l'indirizzo di memoria del primo elemento della stringa (es `5081`) . Gli step della seconda istruzione sono:

- ottieni il valore contenuto dall'indirizzo `4624`, ovvero `5081`
- ottieni `i` e aggiungilo a `5081`
- ottieni il contenuto della cella `5081 + i`

Se invece

```
1 /* nel primo file definiamo un array che contiene char*/
2  char p[10];
3
4  /* nel secondo file lo dichiariamo come puntatore*/
5  external char *p;
6  a = p[3];
```

dato che in questo file abbiamo dichiarato `p` come un puntatore, il lookup avverrà accordingly, indipendentemente dal fatto che `p` sia effettivamente un puntatore o un vettore come nel nostro caso. Supponiamo che `p` abbia indirizzo `5000` e come prima lettera contenga `a`; i passi svolti dall'ultima istruzione

- vai all'indirizzo `5000` e ottieni il valore (ovvero a, che non è un puntatore)
- aggiungi ad `a` il valore `i`
- usa il valore di memoria ottenuto per deferenziare

cosi facendo usiamo un carattere come indirizzo di memoria e iniziamo a fare macelli.

Capitolo 7

Ulteriori tipi di dati complessi

In questa sezione completiamo la panoramica dei tipi complessi, presentando:

1. enumerazioni come tipo di dato;
2. strutture in senso stretto (`struct`);
3. unioni
4. campi di bit

Premettiamo a queste una sezione su `typedef`, che torna spesso volentieri con i tipi di cui sopra.

7.1 `typedef`

Il C permette di definire esplicitamente **sinonimi per i tipi** di dati, tramite la parola chiave `typedef`. L'uso di `typedef`:

- consente di rendere il **codice più leggibile**, andando a sostituire i tipi più complessi ed evoluti con nomi più di più immediata comprensione;
- **aumenta la portabilità** del codice; per modificare un tipo nel passaggio da una architettura ad un'altra basterà modificare l'istruzione `typedef`.

La **sintassi generale** è

```
1 typedef dichiarazione
```

dove dichiarazione è la dichiarazione di un qualcosa, di cui:

- il **tipo della dichiarazione** è considerato come tipo di cui creare il sinonimo;
- l'**identificatore** è considerato come nuovo sinonimo da associare al tipo originale specificato.

Ad esempio (non particolarmente utile sul piano pratico)

```
typedef int foo;
```

in questo modo assegnamo al tipo `int` il nuovo nome `foo`: non creiamo un nuovo tipo. Dopo aver definito il sinonimo potremo riferirci al tipo di dato alternativamente con `int` o `foo`.

Un esempio più complesso La dichiarazione di un puntatore a funzione che restituisce `void` e non prende parametri sarebbe:

```
void (* v_fPtr_v) (void);
```

Se vogliamo semplificarci la vita utilizzando `typedef` e indicando `v_fPtr_v` come sinonimo di puntatore a funzione con quelle caratteristiche, l'istruzione sarà:

```
typedef void (* v_fPtr_v) (void);
```

e non

```
typedef void (* v_fPtr_v) (void)  v_fPtr_v;
typedef void (* ) (void)          v_fPtr_v;
```

o altro.

La differenza tra typedef e l'uso di macro `typedef` può essere visto come la creazione di un tipo di dato incapsulato, ovvero al quale non si può aggiungere in seguito alla definizione:

```
1 #define peach int
2 unsigned peach i;  /* funziona*/
3
4 typedef int banana;
5 unsigned banana i;  /* non funziona*/
```

Inoltre un nome `typedef`fato fornisce il tipo per ogni identificatore in una dichiarazione:

```
1 #define int_ptr int *
2 int_ptr chalk, cheese;
3 /*dopo l'espansione abbiamo int *chalk, cheese */
4 typedef int * int_ptr;
5 /* sono considerati come int * chalk, * cheese; */
```

7.2 Enumerazioni come tipo di dato

A parte la proprietà di associare nomi a costanti, una enumerazione è considerabile come un tipo di dato definito dall'utente.
La **definizione del tipo** è effettuata, come visto, mediante una istruzione del tipo

```
1 enum bool {FALSE = 0, TRUE};
```

Tuttavia il **tag** *bool* identifica il tipo di dato proprio per le successive definizioni di variabili.
La **dichiarazione** di una variabile `bool`:

```
1 enum bool test;
```

Se una variabile è definita con tipo `enum`, è **considerata dal compilatore come variabile intera**, e **può memorizzare qualsiasi intero assegnatole**, sebbene si utilizzi tipicamente uno dei valori specificati nella definizione del tipo.

```
1  #include <stdio.h>
2
3  enum bool {FALSE = 0, TRUE = 1};
4
5  int main(void){
6
7     enum bool asd = 2;
8     printf("%d\n", asd);    /* stampa 2*/
9     asd = TRUE;
10    printf("%d\n", asd);    /* stampa 1*/
11    return 0;
12 }
```

Modi alternativi di dichiarazione Tornando alla definizione dell'`enum`, alternativamente avremmo potuto utilizzare `typedef`:

```
typedef enum {FALSE = 0, TRUE} bool;
```

Si può avere **definizione e dichiarazione insieme** come:

```
enum {FALSE = 0, TRUE} test;
enum bool {FALSE = 0, TRUE} test;
```

la seconda forma, inclusiva del tag, permette di dichiarare in seguito altre variabili di quel tipo.

7.3 `struct`

Le `struct` sono insiemi di dati di tipo non necessariamente omogeneo, accessibili tutti mediante uno stesso; servono per la memorizzazione e la gestione di dati complessi.

7.3.1 Dichiarazione del tipo e dichiarazione di variabili

7.3.1.1 Dichiarazione del tipo

La parola chiave `struct` introduce la dichiarazione di una struttura; è (eventualmente) seguita da un *identificatore* della struttura (in questo caso `us`) e da un elenco di dichiarazioni racchiuse tra parentesi graffe. L'identificatore serve se si vogliono definire in un secondo momento delle occorrenze di questa struttura. Un esempio:

```
1 struct us {
2      int id;
3      char nome[30];
4      char cognome[30];
5      int eta;
```

```
6       int reddito;
7 };
```

Mediante una dichiarazione del genere non si sta riservando spazio in memoria per contenere una variabile: semplicemente si definisce un nuovo tipo di dato derivato. Normalmente la definizione di una struttura sarà inserita in un file header e inclusa in tutti i file sorgente che utilizzino quella struttura.

Non si deve pensare che la **dimensione del tipo di dato** struttura sia la somma delle dimensioni dei propri membri: a causa dei requisiti di **allineamento** per i diversi oggetti, in una struttura potrebbero esserci "buchi"[1]. Si usi l'operatore `sizeof` per determinare la dimensione corretta del tipo di dato.

Le variabili menzionate nella struttura sono chiamate **membri** della stessa. La dichiarazione della struttura termina con un punto e virgola (eccezione alla regola che un blocco di istruzioni non necessita di punto e virgola). Un membro o il contrassegno di una struttura possono avere lo stesso nome di una variabile ordinaria, perché e sempre possibile distinguerli in base al contesto.
Le strutture possono essere **annidate**: ovvero una struttura può avere come membro un'altra struttura, ma non una struttura dello stesso tipo. Può al contrario essere incluso un puntatore a struttura dello stesso tipo (una struttura di questo tipo si chiama *ricorsiva* ed è alla base delle strutture (in senso lato) di dati dinamiche).

```
struct rettangolo {
   struct punto pt1;        /* ok */
   struct punto pt2;        /* ok */
   struct rettangolo rt1;   /* No, questo non è possibile */
   struct rettangolo *rt1;  /* ok */
};
```

7.3.1.2 Dichiarazione di variabili

Una volta definito il tipo possiamo **dichiarare variabili** di quel tipo mediante la forma:

```
struct us luca;        /* singola variabile */
struct us uss[100];    /* vettore di struct */
struct us *usPtr;      /* puntatore ad una struttura */
```

abbiamo così dichiarato la variabile `us001` di tipo `struct us`.

7.3.1.3 L'uso di typedef con le `struct`

È idioma spesso utilizzato quello di dare un sinonimo al tipo di dato della struttura definita in precedenza mediante `typedef`, come in

```
typedef struct us person;
```

o provvedere a direttamente a dichiarazione e sinonimo mediante[2]

[1] Ad esempio una struttura con un `char` e un `int` potrebbe richiedere otto byte al posto di 5

[2] In questo caso, a maggior ragione, l'identificatore `us` è superfluo.

```
1 typedef struct us {
2       int id;
3       char nome[30];
4       char cognome[30];
5       int eta;
6       int reddito;
7 } person;
```

Il beneficio in entrambi i casi è che la susseguente dichiarazione di variabile sarà semplificata a:

```
person luca;        /* singola variabile */
```

7.3.1.4 Definizione di tipo e dichiarazione di variabili congiunta

È anche possibile definire la struttura e dichiarare alcune variabili contemporaneamente. La sintassi,

```
struct us {...} us001, uss[100], *usPtr;
```

con `us` sempre opzionale (tuttavia necessario se si vorranno dichiarare altre variabili di quel tipo).

7.3.2 Operazioni su strutture

Le operazioni ammesse su una struttura sono:

1. l'**accesso** ai singoli **membri**;
2. **copia/assegnamento dell'intera struttura** a differenza dei vettori
3. l'ottenimento dell'**indirizzo** (tramite l'operatore `&`);
4. utilizzo dell'operatore `sizeof` per determinare la **dimensione**.

Le strutture nel loro complesso **non possono esser confrontate** utilizzando gli operatori == e `!=`, in quanto i membri della stessa non sono necessariamente immagazzinati in byte di memoria consecutivi.

7.3.2.1 Inizializzazione

Le strutture può esser inizializzata al momento della dichiarazione o in un secondo momento (accedendo ai campi).

L'inizializzazione in sede di dichiarazione della variabile si attua ponendo le costanti (una per ciascun campo, nessuno saltato) in una lista di inizializzatori tra parentesi graffe, come avviene per i vettori:

```
struct us us001 = { 1,"Luca","Braglia",26,0}
```

Se nell'inizializzazione, tra le graffe, si specificano un numero inferiore di costanti rispetto ai membri, i membri mancanti saranno inizializzati a 0 o `NULL` se puntatori.
L'inizializzazione in tal modo avviene anche per le variabili dichiarate in sede

di definizione della struttura, se in detto contesto oltre ad esser dichiarate non vengono anche inizializzate.

Per i vettori di strutture l'inizializzazione ha forma simile a quella delle matrici:

```
struct us uss[100] = {  { 1,"Luca","Braglia",26,0} ,
                        { 2,"Andrea","Sarri",23,12} ,
                        ...
                        { 100,"Marco","Neri",18,11}
                     }
```

È lecito, anche se deprecabile, non specificare le graffe più interne. Come al solito, se il numero di elementi del vettore non è specificato, sarà desunto dall'inizializzazione.

7.3.2.2 Accesso ai membri

Per accedere ai membri (o campi) di una struttura, il C fornisce l'operatore `.` (detto *operatore punto*) e l'operatore `->` (*operatore puntatore a struttura*). È bene **non porre spazi** attorno a questi due operatori.
L'utilizzo dell'operatore punto per accedere a dati in strutture differenti:

```
us001.reddito           /* struttura "semplice" */
rettangolo.pt1.x        /* struttura annidata */
uss[3].nome             /* vettore di strutture */
```

L'operatore puntatore a struttura accede ai membri di una struttura attraverso un puntatore alla stessa: ad esempio

```
1  #include <stdio.h>
2
3  typedef struct {
4     int id;
5     char nome[30];
6     char cognome[30];
7     int eta;
8     int reddito;
9  } person;
10
11 int main(void){
12
13    person luca = { 1,"Luca","Braglia",23,0};
14    person *pPtr = &luca;
15
16    /* Esempio in scrittura ...  */
17    pPtr->id = 2;
18    /* ... oppure (esempio in lettura) */
19    printf("%d\n", (*pPtr).id);     /* stampa 2 */
20    return 0;
21 }
```

Nel secondo caso le parentesi sono necessarie perché l'operatore punto ha priorità più alta rispetto a quello di deferimento.

7.3.3 Strutture e funzioni

Ad una funzione è possibile:

- passare per valore i membri singoli[3];
- passare per valore l'intera struttura;
- passare un puntatore alla struttura.

I vettori di strutture sono passati per riferimento.

Nel seguito, alcuni esempi di funzioni facenti uso di strutture:

```
1  /* makepoint crea e restituisce una struttura
2     struct point*/
3
4  struct point makepoint(int x, int y)
5  {
6     struct point temp;    /* Creazione */
7     temp.x = x;           /* Inizializzazione */
8     temp.y = y;
9     return temp;          /* Ritorna la struttura */
10 }
11
12 /* Funzione che prende in input una struttura
13    e restituisce un'altra struttura */
14
15 struct point addpoint(struct point p1, struct point p1)
16 {
17    p1.x += p2.x;
18    p1.y += p2.y;
19    return p1;         /* ritorna la struttura*/
20 }
21
22 /* Funzione che prende in input un puntatore
23    a struttura */
24
25 void foo (struct point *p1)
26 {
27     /* ... */
28     p1->x = 1;
29 }
```

7.4 Unioni

Si definisce unione una viariabile che può contenere (in momenti diversi) oggetti di tipo e dimensioni differenti. Come la struttura è un tipo di dato derivato, ma diversamente da questa (che alloca memoria sequenzialmente per i propri membri e può contenere allo stesso tempo dati diversi) alloca un quantitativo di

[3] L'unico modo per passare un vettore per valore è che questo sia membro di una struttura

memoria sufficiente ad immagazzinare il tipo più grande (i membri condividono la stessa area di memoria). Per il resto le analogie con le strutture, anche a livello di sintassi di programmazione sono molteplici.
Una unione è utile nei casi in cui un dato possa assumere tipi diversi a seconda dei casi. In base a ciò si collocherà il dato in una opportuna unione, tenendo opportunamente nota del tipo memorizzato. Spesso i diversi oggetti sono delle strutture (es strutture alternative di dati da manipolare in strutture alternative di memoria). Tutto ciò permette un notevole risparmio di memoria.

7.4.1 Dichiarazione del tipo e dichiarazione di variabili

La dichiarazione del tipo:

```
1 union number {
2     int i;
3     float f;
4 };
```

definisce il tipo `union number` e che potrà contenere valori di tipo `int` o `float`. Dopo la graffa chiusa della definizione è possibile, come avviene per le `struct` inserire identificatori che dichiarano variabile aventi il tipo appena definito. Allo stesso modo il tag `number` è superfluo.
Al pari di una definizione di `struct` quella di union non alloca della memoria, ma crea semplicemente un nuovo tipo di dato. Come nelle strutture, la definizione di una unione sarà inserita in un file header e inclusa in tutti i file sorgente che utilizzino quel tipo di unione.

La **dichiarazione di una variabile** unione

```
1 union number prova;
```

alloca lo spazio necessario per contenere il tipo di dato più grande.

7.4.2 Operazioni su unioni

Le operazioni permesse sulle unioni sono le medesime valide sulle strutture:

- accesso a un membro.
- assegnare un'unione ad un'altra dello stesso tipo,
- richiesta dell'indirizzo mediante l'operatore `&`

Le unioni non possono esser confrontate per le stesse ragioni per cui non possono esserlo le strutture.

7.4.2.1 Inizializzazione e accesso ai membri

L'**inizializzazione** può avvenire contestualmente alla dichiarazione o in seguito:

- considerando la definizione di sopra, si potranno assegnare all'unione o un valore di tipo `float` o un `int`;

- il valore da assegnare si trova in una lista (di lunghezza unitaria, ma racchiusa tra graffe);
- in caso di inizializzazione in dichiarazione, l'unione viene inizializzata solo con un valore del tipo del proprio membro iniziale (intero, nel caso di `number`), che viene assegnato al membro corrispondente (`number.i`).

```c
#include <stdio.h>

typedef union {
   int i;
   float f;
} number;

int main(void){

   number asd = {1.5};
   printf("%d\n%f\n", asd.i, asd.f );
   return 0;

}

/* Se si assegna un float viene troncato perche`
   il primo membro e` un intero. Il programma
   stampa

   1
   0.000000

   Quindi e` meglio evitare di inizializzare in
   dichiarazione una unione
 */
```

Dopo l'inizializzazione sarà possibile utilizzare l'unione nelle espressioni, purché lo si faccia correttamente: il tipo usato deve essere quello memorizzato per ultimo.

L'accesso ai membri dell'unione, come per le strutture avviene mediante:

```
nome_unione.membro
puntatore_unione->membro
```

È responsabilità del programmatore tenere traccia di quale tipo sia di volta in volta memorizzato in un unione, ad esempio mediante l'uso di una variabile come segue

```c
#include <stdio.h>

enum union_type {INT, FLOAT};

union number {
   int i;
```

```
7     float f;
8  };
9
10 int main (void)
11 {
12    union number prova;
13    int prova_type;
14    int test=1;
15
16    if (test==1) {
17       prova.i = 1;
18       prova_type = INT;
19    } else if (test==2) {
20       prova.f = 1.5;
21       prova_type = FLOAT;
22    } else {
23       return 1;        /* error */
24    }
25
26    printf("Valore contenuto %f.\n",
27           prova_type==INT ?
28           (float) prova.i :
29           prova.f);
30
31    return 0;
32 }
```

Un'altra strategia possibile è porre l'unione in una struttura che hanno come membro aggiuntivo la variabile ausiliaria in questione (in tal modo si accede a tutti i dati dell'unione tramite un unico nome, quello della struttura).
Il rischio se non si fa ciò è di utilizzare valori spazzatura:

```
1  #include <stdio.h>
2
3  typedef union {
4     int i;
5     float f;
6  } number;
7
8  int main(void){
9
10    number foo;
11    foo.i = 1;
12    printf("%d\n%f\n", foo.i, foo.f );
13    foo.f = 2.0;
14    printf("%d\n%f\n", foo.i, foo.f );
15
16    return 0;
17 }
18
```

```
/*
   Il programma stampa

   1
   0.000000
   1073741824
   2.000000

   e mostra (alla seconda e terza linea)
   il rischio di usare il membro di
   una unione che non e` stato
   l'ultimo ad essere assegnato

 */
```

7.5 Campi di bit

Un campo di bit (o semplicemente campo), è un insieme di bit contigui contenuti in una unità di memoria, la **parola**, definita dall'implementazione (coincide con la dimensione in byte di un `int`).
I campi di bit consenton un migliore uso della memoria, immagazzinando i dati nel minor numero di bit necessario.
Quasi tutto ciò che concerne i campi di bit è legato a sistema e implementazione, essendo pertanto **poco portabile**[4]: per maggiore portabilità si utilizzino gli operatori bitwise (bit shift e masking) per selezionare porzioni specifiche di bit.
Inoltre per quanto consentano di risparmiare spazio, i campi di bit potrebbero costringere il compilatore a generare un codice in linguaggio macchina più lento, perchè accedere a singole porzioni di una unità indirizzabile sono necessarie operazioni aggiuntive in linguaggio macchina (vi è un trade off tra spazio in memoria e tempo di elaborazione).

Dichiarazione La sintassi di **dichiarazione** si basa, similmente a ciò che avviene per le union, sulle `struct`

```
struct date {
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year : 14;
};
```

questo definisce una tipo di variabile `date` che contiene 3 campi di dimensione rispettivamente pari a 5 bit (potrà memorizzare 32 valori, da 0 a 31), 4 (da 0 a 15) e 14 bit (0-16384).
Il **tipo del singolo campo** può essere `unsigned int` o (signed) `int`.
La **dimensione del singolo campo** dovrà essere un intero compreso tra 0 e il numero totale di bit usato per immagazzinare un `int` sul proprio sistema (se int è di quattro byte, il singolo campo potrà essere al massimo.

[4]Sono altamente deprecati da Kernighan e Pike in The practices of programming.

È possibile specificare un **campo di bit non denominato** da usare come **riempitivo**; ad esempio su una macchina con una word a 4 byte, la dichiarazione

```
struct date {
    unsigned int a : 13;
    unsigned int   : 19;
    unsigned int b : 4;
};
```

fa si che i primi due membri (che con 32 bit, riempiono una parola) stiano sulla prima parola mentre il membro `b` sarà immagazzinato in un'altra unità di memoria.
Inoltre è possibile utilizzare un **campo di bit non denominato con dimensione zero**, usato per allineare il campo di bit successivo su un nuovo confine di unità di memoria, come ad esempio

```
struct date {
    unsigned int a : 13;
    unsigned int   : 0;
    unsigned int b : 4;
};
```

dove di fatto si ottiene lo stesso effetto di piazzare il secondo membro a 19.
Il **razionale** è queste tecniche (campi di bit non denominati) tornano utile per portabilità in quanto alcuni computer permettono che i campi di bit attraversino una parola, mentre altri no.

Dichiarazione e utilizzo di variabili Vediamone l'utilizzo in programmazione. Ai singoli campi si accede come ai membri di una struttura; i campi si comportano come piccoli interi e possono far parte delle espressioni aritmetiche così come gli altri interi:

```
struct date birthday;          /* dichiarazione di variabile*/

birthday.year = 1983;
birthday.month = 11;
birthday.day = 4;

if (current_year - birthday.year >= 18){
...
}
```

Se si assegna un intero la cui rappresentazione è più grande dei bit a disposizione di un determinato campo, verranno memorizzati solo i bit meno significativi.
I campi di bit non sono vettori di bit: tentare di accedere ai singoli bit di un campo coem se fossero elementi è un errore; allo stesso modo i campi di bit non hanno indirizzi, per cui non è possibile applicarvi `&`.

Utilizzi dei campi di bit Le applicazioni principali:

- importazione di dati in formati particolari (es. interi a 9 bit non sarebbero gestiti bene da char ne da int)

```
struct packed-struct {
    unsigned int f1:1;
    unsigned int type:4;
    unsigned int funny_int:9;
};
```

Il campo di tipo `packed-struct` contiene 3 elementi: 1 flag da 1 bit (f1), 1 da 4 (`type`) e `funny_int` da 9 bit;

- comunicazione a basso livello con registri di dispositivi[5].

[5] Ad esempio `http://www.cs.cf.ac.uk/Dave/C/node13.html`.

Capitolo 8

Gestione dinamica della memoria

8.1 Introduzione

La memoria di un computer, in runtime può essere concettualmente suddivisa in:

- memoria **automatica** (**stack**): memoria allocata/deallocata automaticamente quando si entra/esce da un blocco. Si caratterizza per avere dimensionid di allocazione prefissate;

- memoria **dinamica** (**heap**): memoria che viene allocata (prenotata per la memorizzazione di dati) e deallocata (resa libera) dal programmatore in sede di esecuzione. In questo caso nulla avviene automaticamente ma è tutto sulle spalle del programmatore

I vantaggi derivanti dall'utilizzo della memoria dinamica rispetto a quella automatica sono:

- **durabilità**: dato che il programmatore controlla esattamente quando la memoria è allocata e de allocata, è possibile costrutire una struttura di dati all'interno di una funzione e restituirla attraverso un puntatore al chiamante;

- **efficienza di memoria**: dato che la dimensione allocata può essere controllata esattamente (a contrario, ad esempio delle allocazioni statiche di array per contenere stringhe, che si spera bastino).

Gli svantaggi:

- **più lavoro** da fare tra allocazioni e deallocazioni

- **maggior rischio di bug** nel caso che il processo di allocazione/deallocazione non sia gestito correttamente (la memoria statica è rigida, ma almeno non sbaglia mai)

8.2 Allocazione e deallocazione

Si usano le funzioni definite in `stdlib.h`

8.2.1 Allocazione

malloc Al fine di allocare/richiedere memoria per il nostro programma dobbiamo utilizzare la funzione di allocazione della heap, che nel caso del C è la funzione `malloc`

```
1 void *malloc(size_t size);
```

`malloc` prende per argomento il numero di byte che dovranno essere allocati (si usa tipicamente `sizeof`) e restituisce un puntatore a `void` che punta all'area di memoria allocata, o `NULL` nel caso non vi riesca (poichè la memoria della heap è terminata). Tale puntatore potrà poi essere assegnato a qualsiasi altro tipo di puntatore.
Va detto che la memoria puntata non è inizializzata quindi contiene valori spazzatura, quindi bisogna prestare attenzione per evitare di leggervi prima di una inizializzazione.
È pratica sicura, inoltre, incapsulare `malloc` in una funzione che fa abortire il programma, stampando errore, se l'allocazione non va a buon fine (se è finita la memoria). Un esempio in ambiente Unix[1]:

```
void *
xmalloc (size_t size)
{
  void *alloc = malloc(size);

  if (alloc == 0)
    {
      syslog (LOG_ERR, "out of memory (malloc)");
      exit (EXIT_FAILURE);
    }

  return alloc;
}
```

calloc La funzione `calloc`

```
1 void *calloc(size_t nmemb, size_t size);
```

alloca spazio per un vettore di `nmemb` oggetti, di dimensioni pari a `size`. Lo spaizo allocato sarà inzializzato con tutti i bit a zero. La funzione restituisce un puntatore valido o `NULL` (nel caso non riesca ad allocare).
A livello di performance è lievemente peggio di `malloc`, dovendo inizializzare.

realloc La funzione `realloc`

```
1 void *realloc(void *ptr, size_t size);
```

[1] Nel listato la funzione `xmalloc` del programma `flcd`: `http://www.nongnu.org/flcd/`.

cambia la dimensione del blocco di memoria puntato da `ptr`.
La funzione può muovere il blocco di memoria ad una nuova locazione (il cui indirizzo è ritornato); Se gli viene passato un `ptr` che è `NULL` la funzione si comporta come `malloc`, assegnando un nuovo blocco di memoria e restituendo relativo puntatore.

8.2.2 Deallocazione

Si utilizzi `free` appena l'area di memoria non sia più necessaria: non rilasciare la memoria potrebbe causare un esaurimento della memoria heap (*memory leaks*).
La funzione `free`

```
1 void free(void *ptr);
```

dealloca la memoria, allocata in precedenza da `malloc`, `calloc` o `realloc`, puntata da `ptr`.

```
1 free(ptr)
```

E' importante passare a free esattamente lo stesso puntatore restituito dalle funzioni di prima (non ad esempio un puntatore ad una sezione interna del blocco di memoria riservato), perchè è da tale indirizzo che il sistema operativo inizierà a markare la memoria come disponibile.
Utilizzare il puntatore della memoria deallocata in deferenziazione costituirà errore, a meno che non si sia prima nuovamente utilizzata `malloc`, per assegnargli un nuovo valore.

8.3 Alcuni esempi

Un intero Una semplice un blocco pari ad un `int` nella heap e vi memorizza un numero, dopodichè dealloca:

```
1 void Heap1(){
2
3   int *intPtr;
4
5   intPtr = malloc(sizeof(int));
6   *intPtr = 42;
7   free(intPtr);
8 }
```

Un array Nel C è comodo allocare un array nell'heap: il codice che segue alloca un array di 100 `struct fraction`, le imposta tutte a 22/7 e dealloca l'array

```
1 void HeapArray(){
2
3   struct fraction* fracts;
4   int i;
5
6   /* allocazione */
```

```
7   fracts = malloc(sizeof(struct fraction) * 100);
8   /* ora lo si usa come un array */
9   for(i = 0; i < 99; i++){
10    fracts[i].numerator = 22;
11    fracts[i].denominator = 7;
12  }
13  /* deallocazione */
14  free(fracts);
15 }
```

Una stringa La funzione prende una stringa, ne crea una copia nell'heap e ritorna un puntatore alla nuova stringa. Il chiamante ha la responsabilità di deallocare in seguito

```
1  char* StringCopy(const char* string){
2    char *newString;
3    int len;
4
5    len  = strlen(string) + 1;
6    newString = malloc(sizeof(char) * len);
7    assert(newString != NULL);
8    strcpy(newString, string);
9    return(newString);
10 }
```

La funnzione specifica, a tempo di esecuzione, l'esatta dimensione del blocco necessario da allocare (mentre questo non è possibile con la memoria statca).

Capitolo 9

Strutture di dati dinamiche

9.1 Introduzione

Le strutture di dati dinamiche:

- si differenziano da quelle con dimensione fissa, per il fatto che le loro dimensioni e la capacità di immagazzinamento variano a seconda delle necessità durante l'esecuzione del programma;
- sono basate sull'uso di puntatori e sull'allocazione dinamica della memoria.

9.1.1 Principali strutture dinamiche

Linked list Una lista è un insieme di "nodi" collegati linearmente. I nodi sono costituiti da `struct` che contengono dati ed un puntatore all'elemento successivo della lista (liste *unidirezionali*). L'ordine con cui sono collegati i nodi definisce un ordinamento tra di loro. Un nodo funge da testa della lista, e da questo è possibile accedere a tutti i nodi della lista. Conoscendo un nodo interno alla lista, è possibile accedere ai nodi successivi, ma non a quelli precedenti.

Albero Ogni nodo contiene due (o più) puntatori ad altri nodi che sono detti suoi "figli". Dato un nodo, è possibile accedere a tutti i suoi discendenti.

Grafo E' una generalizzazione dell'albero. Ogni nodo ha un numero arbitrario di nodi "vicini", può contenere cicli. In generale, può essere associato un carico utile sia ai nodi che ai collegamenti tra di loro.

Hash table Struttura dati utile per cercare velocemente un elemento all'interno di un insieme sulla base di una chiave (un sottoinsieme dalle caratteristiche dell'elemento). Di ciascun elemento da memorizzare viene calcolato un hash della chiave. Viene poi costruito un array, che ha come indice il valore dell'hash, come elementi puntatori a liste di nodi che corrispondono a quel valore dell'hash. Per cercare un elemento, si calcola il suo valore di hash, si seleziona l'elemento dell'array corrispondente e si percorre la lista fino a quando non lo si trova.

9.1.2 Strutture ricorsive e allocazione dinamica della memoria

Le `struct` ricorsive sono alla base di tutte le strutture di dati dinamiche: esse contengono come membro un puntatore ad un'altra struttura dello stesso tipo. Ad esempio

```
1 struct classmate {
2
3    unsigned int id;
4    char name[20];
5    char phone_number[10];
6    struct classmate * next;
7
8 };
```

il membro `next` permette di collegare una `struct classmate` ad un'altra del medesimo tipo.
Creare e gestire strutture di dati dinamiche richiede l'allocazione dinamica della memoria, che nella maniera più semplice viene affrontata come segue per l'allocazione:

```
1 nextPtr = malloc( sizeof(struct classmate) );
```

assegna a `newPtr` l'indirizzo di memoria di una area appena allocata che può esser riempita con nuovi dati. L'assegnamento è corretto sebbene i tipi di lvalue e rvalue siano diversi in quanto malloc restituisce un puntatore a `void`.

9.2 Linked list

Si parte con alcuni esempi di base per illustrare il funzionamento minimo delle liste.

9.2.1 Esempi di base

Iniziamo con il creare il più semplice **costruttore** di un equivalente ad array di interi che possa modificarsi in sede di esecuzione:

```
1  #include <stdlib.h>
2
3  struct node {
4    int  data;
5    struct node * next;
6  };
7
8  struct node* buildSimple(){
9
10   struct node* head = NULL;
11   struct node* second = NULL;
12   struct node* third = NULL;
13
```

```
14   head = malloc(sizeof(struct node));
15   second = malloc(sizeof(struct node));
16   third = malloc(sizeof(struct node));
17
18   head->data = 1;
19   head->next = second;
20
21   second->data = 2;
22   second->next = third;
23
24   third->data = 3;
25   third->next = NULL;
26
27   return head;
28 }
```

La funzione, dopo aver allocato i tre nodi, inizializza la parte di dati e il collegamento dei nodi stessi; infine ritorna il puntatore alla testa della lista. Sarà in seguito responsabilità del chiamare deallocare la lista.

Una seconda funzione utile è quella che calcola la **lunghezza della lista**:

```
1  unsigned int listLength(struct node* head){
2
3    unsigned int count = 0;
4    struct node * current = head;
5    /* Idioma comune per scorrere una lista */
6    while (current != NULL){
7      count++;
8      current = current->next;
9    }
10   return count;
11 }
```

alternativamente alcuni preferiscono

```
1  for(current = head; current != NULL; current = current->next) count++;
```

Scriviamo anche una funzione per **stampare il contenuto**:

```
1  void printList(struct node* head){
2
3    struct node * current = head;
4
5    while(current != NULL){
6      printf("%d --> ", current->data);
7      current = current->next;
8    }
9    printf("NULL\n");
10
11 }
```

L'utilizzo delle funzioni sinora viste in un programma sarà analogo al seguente:

```
1 struct node * head = buildSimple();
2 printf("%u\n", listLength(head));
3 printList(head);
```

che stamperà

```
l@np350v5c:~$ ./a.out
3
1 --> 2 --> 3 --> NULL
```

9.2.2 Aggiungere un nodo all'inizio della lista

`buildSimple` è abbastanza ok, ma ovviamente non costituisce un meccanismo generale per costruire delle liste: la cosa migliore sarebbe una funzione indipendente che aggiunge un singolo nodo a qualsiasi lista (specifica per questa struttura).
Vi sono 3 classici step che **aggiungono un singolo nodo all'inizio di una lista**:

1. **allocare** il nodo nell'heap e impostare il membro `data`
2. **linkare il next** della nuova struttura all'inizio della lista
3. **cambiare l'head** della lista

Creiamo ora una prima funzione `wrong_push1` che svolge i primi due step:

```
struct node* wrong_push1(struct node * head, int value){
  /* 1 */
  struct node *newNode = malloc(sizeof(struct node));
  newNode->data = value;
  /* 2 */
  newNode->next = head;
  return(newNode);
}
```

La funzione è abbastanza ok e può essere utilizzata così:

```
    struct node * head = buildSimple();
    printf("%u\n", listLength(head));
    printList(head);
    head = wrong_push1(head, 4);
    printList(head);
```

e porta correttamente a stampare

```
l@np350v5c:~$ ./a.out
3
1 --> 2 --> 3 --> NULL
4 --> 1 --> 2 --> 3 --> NULL
```

Tuttavia l'ideale sarebbe **cambiare l'`head` nella chiamata in maniera incapsulata**, in maniera da poter evitare assegnazioni nella chiamante stessa.
Una prima versione è

```
void wrong_push2(struct node * head, int value){
  struct node *newNode = malloc(sizeof(struct node));
  newNode->data = value;
  newNode->next = head;
  /* wrong 3 */
  head = newNode;
}
```

Il problema è che non funziona nella `main` di prima

```
l@np350v5c:~$ ./a.out
3
1 --> 2 --> 3 --> NULL
1 --> 2 --> 3 --> NULL
```

perchè all'ultima istruzione assegnamo ad una variabile locale che contiene il puntatore dell'`head` e non modifichiamo l'`head` nella chiamante. Quindi abbiamo allocato correttamente un nodo a monte della lista ma stiamo continuando ad utilizzare il vecchio `head`.

Per una versione corretta dobbiamo un puntatore alla variabile che contiene l'`head` e implementare una funzione come segue:

```
1 void push(struct node ** head, int value){
2
3   struct node *newNode = malloc(sizeof(struct node));
4   newNode->data = value;
5   newNode->next = *head;
6   *head = newNode;
7 }
```

che usata come

```
1     struct node * head = buildSimple();
2     printList(head);
3     push(&head, 4);
4     printList(head);
```

funziona correttamente

```
l@np350v5c:~$ ./a.out
1 --> 2 --> 3 --> NULL
4 --> 1 --> 2 --> 3 --> NULL
```

Tale funzione è generale e lavora bene anche se gli si passa un puntatore inizializzato a `NULL` che funge da head della lista nella chiamante.

9.2.3 Aggiungere un nodo alla fine della lista

Un modo in cui non si sbaglia di certo è scorrere l'intera lista ma se le operazioni di inserimento sono numerose conviene mantenere traccia della coda.
Utilizziamo una funzione del genere

```
1  void append(struct node **headptr,
2              struct node **lastptr,
3              int x){
4
5    struct node * new = malloc(sizeof(struct node));
6    new->data = x;
7    new->next = NULL;
8
9    if ( (*headptr) != NULL) {
10     /* ! first element (most common) */
11     (*lastptr)->next = new;
12     *lastptr = new;
13   } else {
14     /* first element */
15     *lastptr = *headptr = new;
16   }
17
18 }
```

e utilizzandola in una `main` del genere

```
1  int main (){
2      struct node * head = NULL;
3      struct node * last = NULL;
4      append(&head, &last, 1);
5      append(&head, &last, 2);
6      printList(head);
7      return 0;
8  }
```

stamperà correttamente

```
l@np350v5c:~$ ./a.out
1 -> 2 -> NULL
```

9.2.4 Liberare una lista

Deallocare completamente una lista Dobbiamo deallocare tutti i suoi nodi. Una funzione che ne da un esempio, utilizzabile in maniera compatta per la struttura trattata è:

```
1  void freeList(struct node ** head){
2
3      struct node * p, * next;
4
5      for(p = *head; NULL != p; p = next){
6          next = p->next;
7          free(p);
8      }
9      *head = NULL;
10 }
```

Eliminare uno o più elementi Occorre solitamente individuarli sulla base dei dati della struttura, dopodichè

- salvare il puntatore a `next` dell'elemento che vogliamo eliminare nell'ultimo membro utile per non interrompere la lista;
- eliminare l'elemento

Una funzione che elimina un singolo elemento alla volta è la seguente

```
1  int delete(struct node **s, int x){
2
3      struct node *previous;
4      struct node *current;
5      struct node *temp;
6
7      /* Elimina il primo nodo */
8      if (x == (*s)->data){
9          temp = *s;
10         *s = (*s)->next;
11         free(temp);
12         return x;
13
14     } else {
15
16         previous = *s;
17         current = (*s)->next;
18
19         /* fai un ciclo per trovare la posizione corretta */
20         while (current != NULL && current->data != x){
21             previous = current;
22             current = current->next;
23         }
24
25         /* elimina il nodo puntato da current ptr */
26         if (current != NULL) {
27             temp = current;
28             previous->next = current->next;
29             free(temp);
30             return x;
31         }
32     }
33
34     return 0;
35 }
```

che utilizzata in una main del tipo

```
1  int main (){
2
3      struct node * head = NULL;
4      struct node * last = NULL;
```

```
5
6       append(&head, &last, 1);
7       append(&head, &last, 2);
8       append(&head, &last, 2);
9       append(&head, &last, 3);
10
11      delete(&head, 2);
12      printList(head);
13      return 0;
14  }
```

stampa

```
l@np350v5c:~$ ./a.out
1 -> 2 -> 3 -> NULL
```

9.2.5 Varianti delle liste

Alcune varianti:

- Le **liste bidirezionali** contengono un puntatore sia al nodo precedente che al successivo, consentendo di muoverci in entrambe le maniere.
- Liste **circolari**: liste che invece di aver settato l'ultimo next a `NULL`, lo settano all'indirizzo del primo nodo. Quindi un puntatore a qualsiasi elemento della lista permette di accedere a tutta la lista
- Le **stack** (pile) sono liste nelle quali l'inserzione e l'eliminazione di elementi sono eseguite solo da una estremità (strutture LIFO, *last in first out*). Si serve tipicamente delle funzioni `push` per porre un elemento all'inizio della lista e `pop` per estrarlo
- Le **queue** (code) sono liste in cui l'inserzione avviene ad una estremità mentre le eliminazioni dall'altra (struttura FIFO, *first in first out*). La funzione `enqueue` pone in coda mentre `dequeue` estrae dalla stessa.

In queste ultime due, mentre `push` ed `enqueue` si preoccupano di inserire l'elemento al punto giusto le funzioni `pop` e `dequeue`

- si pongono all'estremità di interesse della lista
- memorizzano i dati in una variabile temporale
- preparano i puntatori circostanti affinchè la deallocazione del nodo non comprometta l'utilizzo della lista
- deallocano il nodo
- restituiscono alla chiamante i dati del nodo

9.3 Alberi

Liste, pile e code sono strutture di dati lineari

In un albero ogni nodo contiene due (o più) puntatori ad altri nodi che sono detti suoi "figli"; dato un nodo è possibile accedere a tutti i suoi discendenti.
In un albero

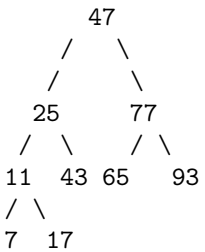
- ciascun nodo, oltre ai puntatori ai nodi figli, ha normalmente una sezione di dati utile per il problema applicativo da risolvere;
- non ci debbono essere **cicli**: un nodo non può essere discendente di sé stesso, ovvero non deve essere possibile partire da un nodo, seguire i puntatori ai figli e ritornare al nodo di partenza;
- ciascun nodo deve essere figlio di un solo padre;
- tra i figli di un nodo esiste normalmente una **relazione d'ordine** (ad esempio i figli a sinistra hanno un valore di dato inferiore a quello del padre mentre quelli a destra superiore);
- spesso ogni nodo ha un numero fissato di figli, ad esempio due o tre (si parla in questo caso di alberi binari o ternari).
  In altri casi, il numero di figli di un nodo è arbitrario; questo può essere gestito memorizzando i figli di un nodo in una lista.

9.3.1 Alberi di ricerca binaria

In questa sezione ci concentriamo sugli **alberi binari**: essi sono veloci sia in inserzione che in lookup. Essi sono pertanto utili per problemi tipo dizionario dove la chiave inserita viene confrontato con un dato dei nodi per ricercare il nodo di interesse (ed estrarre eventualmente l'altro dato richiesto).
Tra gli alberi binari si distinguono gli **alberi di ricerca binaria** per il fatto che:

- non vi sono nodi duplicati (ovvero che contengono lo stesso dato)
- ogni nodo può essere inserito solamente come nodo figlio;
- i figli a sinistra hanno un valore di dato inferiore a quello del padre mentre quelli a destra superiore;

La rappresentazione di un albero di ricerca binario che viene riempito con valori interi può essere la seguente:



E' possibile visitare tale albero in tre modi:

- in ordine
- con visita simmetrica anticipata
- con visita simmetrica differita

Tipicamente le funzioni che si occupano di gestire un albero sono ricorsive. Iniziamo a vedere la funzione per **alimentare un nodo**:

```
1  void insertNode(struct node **p, int x){
2
3      if (*p == NULL){     /* albero vuoto */
4
5          *p = malloc(sizeof(struct node));
6
7          if ( *p != NULL ){
8              (*p)->data = x;
9              (*p)->left = NULL;
10             (*p)->right = NULL;
11         } else
12             printf("%d not inserted", x);
13
14     } else {            /* albero non vuoto */
15
16         if (x < (*p)->data ){
17             insertNode( &((*p)->left), x);
18         } else if (x > (*p)->data){
19             insertNode( &((*p)->right), x);
20         } else
21             printf("Duplicato scartato: %3d\n", x);
22
23     }
24 }
```

Vediamo ora le funzioni di visualizzazione, partendo dalla **visualizzazione anticipata**

```
1  void preOrder(struct node *p){
2
3      if (p != NULL) {
4          printf("%3d", p->data);
5          preOrder(p->left);
6          preOrder(p->right);
7      }
8
9  }
```

visualizzazione in ordine:

```
1 void inOrder(struct node *p){
2
3     if (p != NULL) {
4         inOrder(p->left);
5         printf("%3d", p->data);
6         inOrder(p->right);
7     }
8
9 }
```

la **visualizzazione differita**

```
1 void postOrder(struct node *p){
2
3     if (p != NULL) {
4         postOrder(p->left);
5         postOrder(p->right);
6         printf("%3d", p->data);
7     }
8
9 }
```

e la `main` è similare a

```
1  int main (){
2
3      int i;
4      int values[DIM] = { 27, 13, 42,  6, 17,
5                          6, 33, 54, 13, 43};
6      struct node * head = NULL;
7
8      for(i = 0; i <DIM; i++)
9          insertNode(&head, values[i]);
10
11     printf("Visita anticipata\n");
12     preOrder(head);
13
14     printf("\nVisita simmetrica\n");
15     inOrder(head);
16
17     printf("\nVisita differita\n");
18     postOrder(head);
19     printf("\n");
20
21     return 0;
22 }
```

che stampa

```
l@np350v5c:~$ ./a.out
```

```
Duplicato scartato:   6
Duplicato scartato:  13
Visita anticipata
 27 13  6 17 42 33 54 43
Visita simmetrica
  6 13 17 27 33 42 43 54
Visita differita
  6 17 13 33 43 54 42 27
```

Una **funzione di ricerca** per testare la presenza di un valore sarà:

```
1  int lookup(struct node* node, int x){
2
3      if (node == NULL){
4          return 0;
5      } else {
6          if (x == node->data)
7              return 1;
8          else {
9              if (x < node->data)
10                 return(lookup(node->left, x));
11             else
12                 return(lookup(node->right, x));
13         }
14
15     }
16 }
```

La **ricerca** di un valore in un albero binario è **un'operazione rapida**: qualora l'albero sia stato riempito bene, allora ogni livello conterrà circa il doppio degli elementi di quello precedente. Di conseguenza, un albero di ricerca binaria con n elementi avrà un massimo di livelli e, quindi, dovranno essere eseguiti un massimo di confronti per trovare una corrispondenza o per determinare che non ne esistano.
Cio significa per esempio che per un'indagine in un albero di ricerca binaria (riempito bene) con 1000 elementi, dovranno essere eseguiti non piu di 10 confronti perchè $2^{10} > 1000$; oppure per un'indagine con 1000000 elementi dovranno essere eseguiti non più di 10 confronti perchè $2^{20} > 1000000$.

Capitolo 10

Gestione dei bit

Il C mette a disposizione operatori applicabili a tipi interi che lavorano direttamente sui bit; partiamo da un ripasso di sistemi numerici (in particolare binario ed esadecimale) e della rappresentazione di interi nei calcolatori, per passare poi alla elaborazione dei bit in programmazione.

10.1 Sistemi numerici

I sistemi di numerazione sono insiemi di simboli (cifre) mediante i quali è possibile rappresentare qualunque numero, e possono distinguersi in sistemi:

- **additivi**: ad ogni cifra è associato un valore numerico prefissato e il valore del numero è stabilito dalla somma del valore delle sue cifre (es sistema di numerazione romano)

- **posizionali**: il valore del numero dipende sempre dalla somma del valore delle sue cifre, ma il valore di queste dipende dalla posizione

I sistemi posizionali sono oggi i più utilizzati, perchè permettono rappresentazione più compatta e svolgimento calcoli in maniera più rapida.

I sistemi posizionali possono distinguersi in ragione della **base di riferimento**: essa consiste nel numero di cifre utilizzate per la rappresentazione di numeri. I sistemi più comuni sono

- sistema **binario** (base 2): utilizza $\{0, 1\}$

- sistema **ottale** (base 8): utilizza $\{0, \ldots, 7\}$

- sistema **decimale** (base 10): utilizza $\{0, \ldots, 9\}$

- sistema **esadecimale** (base 16): utilizza $\{0, \ldots, 9, A, \ldots, F\}$

Una tabella (10.1) di conversione tra sistemi per i primi 16 numeri è riportata. Il vantaggio di utilizzare sistemi di numerazione con una base maggiore consta nel fatto che, come si vede, per rappresentare uno determinato valore (soprattutto se elevato) occorre un minore numero di cifre.

Decimale	Binario	Ottale	Esadecimale
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Tabella 10.1: Conversione tra sistemi numerici per i primi 16 numeri

10.1.1 Rappresentazione di un numero in base b

In generale indichiamo con b la base del sistema di numerazione (che può essere $2, 8, 10, 16$).
Un generico intero N in base p è rappresentato da una sequenza di cifre

$$a_n,\ a_{n-1},\ \ldots,\ a_0$$

dove ciascuna a_i rappresenta un numero compreso tra 0 e $b-1$. Si dice che:

- a_n è la **cifra più significativa**
- a_0 la **meno significativa**

La base di un numero va specificata come pedice: se non è specificata si intende $b = 10$.
Se x è un numero e $a_n, a_{n-1}, \ldots, a_0$ sono le cifre della sua rappresentazione in base b, allora è

$$x_b = a_n b^n + a_{n-1} b^{n-1} + \ldots + a_0 b^0 \tag{10.1}$$

Ad esempio, con $b = 10$

$$1213 = 1 \cdot 10^3 + 2 \cdot 10^2 + 1 \cdot 10^1 + 3 \cdot 10^0 = 1213$$

10.1.2 Conversioni di base

Per effettuare una conversione dalla base b_1 ad un'altra base b_2, con $b_1, b_2 \neq 10$ è quasi sempre consigliabile operare in due fasi: prima si converte da b_1 a $b = 10$, poi da questa a b_2.
Costituiscono eccezione alla regola le conversioni da binario a ottale o esadecimale e viceversa.

10.1.2.1 Conversione da base b a 10

La conversione a base 10 è abbastanza immediata in quanto si applica la 10.1 (es con base $b = 2$):

$$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 2 + 1 = 11_{10}$$

10.1.2.2 Conversione da base a 10 a b

Al contrario, per convertire da base 10 a base b il numero x:

- si divide il numero in base 10 per b; il resto della divisione costituirà l'ultima cifra a destra, in base b di x
- si divide il risultato della divisione di prima ancora per b; il resto costituirà la cifra immediatamente a sinistra di quella ottenuta prima;
- si va avanti così fino ad ottenere un quoziente nullo.

Ad esempio per convertire 7512_{10} in base $b = 16$:

$$\begin{aligned} 7512 : 16 &= 469 \quad \text{resto } \mathbf{8} \\ 469 : 16 &= 29 \quad \text{resto } \mathbf{5} \\ 29 : 16 &= 1 \quad \text{resto } \mathbf{13} \\ 1 : 16 &= 0 \quad \text{resto } \mathbf{1} \end{aligned}$$

... dunque

$$7512_{10} = 1D58_{16} \tag{10.2}$$

10.1.2.3 Binario, Ottale, Esadecimale

Per convertire:

- **da binario ad ottale** un numero si raggruppano a tre a tre le cifre binarie, a partire da quella più a destra. Se le cifre del gruppo più a sinistra sono solo due o una sola si completa tale gruppo con uno o due zeri rispettivamente. Infine ad ogni gruppo di tre cifre binarie si sostituisce la cifra ottale corrispondente.
- **da ottale a binario** sarà fatto sostituendo a ciascuna cifra ottale una tripletta di binarie equivalente.
- **da e verso sistema esadecimale** si lavorerà come con l'ottale, soltanto che al posto delle triplette di cifre binarie utilizzeremo gruppi da 4.

Per l'informatica il sistema di numerazione ottale e soprattutto esadecimale sono importanti perchè permettono esprimere compattamente un numero binario (l'unità memorizzabile da un elaboratore), attraverso le regole di sopra.

10.2 La gestione dei bit nel C

I bit sono i singoli elementi informativi che costituiscono le variabili: tutti i dati sono memorizzati mediante bit.
Il C prevede sei **operatori per la manipolazione dei bit** applicabili a **operandi interi**, solitamente `unsigned`.
Va infine detto che **i risultati delle operazioni bitwise dei dati dipendono dalla macchina**.

10.2.1 Operatori bitwise

Gli operatori bitwise operano sui bit degli operandi **on the fly**, senza modificare gli operandi stesso.
Per ciascuno degli operatori che si va a vedere, ad eccezione di `~`, esiste l'equivalente **operatore di assegnamento** che si crea ponendo un `=` dopo l'operatore bitwise, ed ha significato analogo a quelli di assegnamento per le operazioni aritmetiche.
Nei seguenti esempi si considera come esempio l'applicazione degli operatori bitwise ad `unsigned char` (8 bit con valore rappresentabile che spazia da 0 a 255).

AND Rappresentato da `&` imposta a 1 i bit del risultato se entrambi quelli corrispondenti nei suoi due operandi siano 1;

```
01001011 &  (75)
00010101 =  (21)
--------
00000001     (1)
```

Pertanto la seguente espressione

```
printf("%u\n", 75 & 21 );
```

stampa proprio 1.

OR Rappresentato da `|`, imposta a 1 se almeno uno dei rispettivi bit dei due operandi vale 1;

```
01001011 |
00010101 =
--------
01011111   (95)
```

XOR (OR esclusivo) Rappresentato da `^`, imposta a 1 se solo uno dei rispettivi bit dei due operandi vale 1;

```
01001011 ^
00010101 =
--------
01011110   (94)
```

NOT (complemento a 1) Rappresentato da `~` è un operatore unario e imposta a 0 i bit a 1 e viceversa.

```
~01001011
---------
 10110100
```

Scorrimento a sinistra Rappresentato da `«`, fa scorrere a sinistra i bit del primo operando, per un numero di volte specificato dal secondo operando; i bit rimasti vuoti a destra sono riempiti con 0;

```
01001011 <<  (75)
       2
--------
00101100     (44)
```

Lo spostamento a sinistra di `n` bit corrisponde a moltiplicare il valore per 2^n, a meno di overflow (che accadrebbero anche mediante l'operazione aritmetica).

Scorrimento a destra Rappresentato da `»`, fa scorrere a destra i bit del primo operando per un numero di volte specificato dal secondo operando: con cosa (0 o 1) vengono riempiti i bit vuoti a sinistra dipende dalla macchina;

```
01001011 >>
       2
--------
??010010
```

10.2.2 La visualizzazione dei bit

Quando si utilizzano gli operatori bitwise è utile visualizzare i valori nella loro rappresentazione binaria; per fare questo si costruisce una funzione, che utilizza l'operatore `&` e una maschera. La maschera serve per nascondere alcuni bit mentre si selezionano gli altri: tutto questo avviene mediante l'applicazione di `&`. La gestione della maschera di selezione avviene mediante gli operatori di spostamento.

```
unsigned int counter;
unsigned char mask = 1 << 7;
unsigned char displayme = 194;

for (counter=1; counter <= 8  ; counter++){
   putchar(displayme & mask ? '1' : '0');
   displayme <<= 1;
}
```

```
l@a6k:~$ ./a.out
11000010
```

La maschera viene impostata con il bit più significativo a 1 (gli altri 0) mediante lo scorrimento a sinistra della seconda istruzione; dopo si entra nel ciclo e verrà stampato 1 se la condizione è valida. Nel momento in cui `displayme` e `mask`

saranno combinate mediante `&` tutti i bit di `displayme` ad eccezione di quello più significativo saranno "mascherati" poichè ogni bit messo in `AND` con 0 produce 0. Pertanto l'espressione valuterà diverso da 0, quindi vero, se il bit considerato di `displayme` è 1. L'espressione sarà vera e verrà stampato`1`; `0` in caso contrario.

10.2.3 Alcuni idiomi utili con operatori bitwise

10.2.3.1 OR

L'operatore operatore è spesso usato per azzerare un insieme di bit: ad esempio

```
n = n & 0177
```

imposta a 0 tutti i bit di n eccetto i 7 meno significativi (quelli più a destra). Alternativamente serve per verificare se alcuni bit sono a 1 (nell'esempio il quarto bit):

```
if (b & 0x10) {
} else {
}
```

10.2.3.2 OR

Questo operatore invece si usa per assicurarsi che certi bits siano impostati a 1; ad esempio

```
x = x | SET_ON
```

imposta a uno tutti i bit di x che corrispondon a bit di `SET_ON` di valore 1. Costanti come `SET_ON` o `0177`, impiegate per impostare i valori di bit di un certo campo, sono dette *maschere* (sono valori interi, strumentali in queste operazioni, in cui determinati bit sono impostati a 1);

10.2.3.3 XOR (OR esclusivo)

Un applicazione consiste nell'invertire bit limitrofi, supponiamo di voler invertire il terzo e il quarto bit, li estraiamo con una maschera e applichiamo `^`:

```
01001011 ^  (75)
00011000    0x18
--------
01010011
```

10.2.3.4 NOT

A parte l'inversione, viene utilizzato insieme a `&` per assicurarsi che certi bit siano settati a zero. Se ad esempio vogliamo che il quarto bit sia 0:

```
50 & ~0x10

00110010 & (50)
11101111   (~0x10)
--------
00100010
```

10.2.3.5 TODO

http://stackoverflow.com/questions/276706/what-are-bitwise-operators
http://bits.stephan-brumme.com/

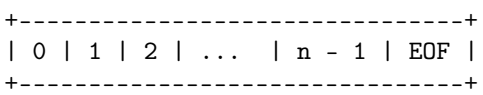
Capitolo 11

Input e output

In questa sezione vedremo la parte di programmazione che riguarda l'interazione con l'esterno. L'header di riferimento è `stdio.h`.

11.1 Stream

Per interfacciarsi con il mondo il C considera qualsiasi forma di input o di output come uno **stream**), ovvero una sequenza di byte terminati dalla costante `EOF` (acronimo di *end of file*).



Tre **stream di default** sono aperti automaticamente quando comincia l'esecuzione del programma (e costituiscono i canali di default di comunicazione con il mondo esterno del programma stesso):

1. **standard input**: di default associato alla tastiera
2. **standard output**: di default associato al monitor
3. **standard error**: stream di output separato dallo standard output specificamente per indicare situazioni di malfunzionamento del programma.

Questi stream sono referenziati attraverso i puntatori `stdin`, `stdout` e `stderr`.

11.2 Buffering

La libreria standard effettua il **buffering**, ovvero crea area di memoria temporanee al fine di limitare le operazioni di lettura di basso livello e in meno chiamate possibili di esse gestire tutta l'informazione richiesta.
Il buffer è ottenuto dalle funzioni di input e output, che chiamano `malloc` la prima volta che operano su uno stream. Il termine **flush** indica la scrittura di un buffer: un buffere può essere flushed automaticamente (dalle routine standard I/O, come quando il buffer si riempie) o manualmente mediante la funzione `fflush`.
Tre tipi di buffering sono possibile:

- **buffering completo**: l'I/O di basso livello avviene solamente quando il buffer è pieno;
- **line buffering**: l'I/O di basso livello avviene quando si incontra il carattere `\n`. Cioè se desideriamo inviare 15 caratteri e un `\n` allo `stdout`, ciò non avverrà separatamente per ogni carattere ma verrà riempito un buffer sino a giungere al newline e dopo avverrà l'output di basso livello;
- **no buffering**: la libreria non effettua il buffering, quindi desideriamo scrivere 15 caratteri questi saranno inviati alle procedure di basso livello il prima possibile, senza passare per un'area di memoria temporanea

Tipicamente:

- lo `stderr` non ha buffering
- tutti gli altri stream hanno un buffering per linea se si riferiscono ad un terminale (`stdin`); alternativamente hanno buffering completo.

Se desideriamo **cambiare il buffering** di uno stream occorre utilizzare le funzioni `setbuf` o `setvbuf` dopo che lo stream è stato aperto ma prima che venga fatta qualsiasi operazione di lettura/scrittura.

11.3 Gestione degli stream

11.3.1 Apertura

Per aprire uno stream ci possiamo servire di due funzioni[1]:

```
1 FILE *fopen(const char *path, const char *mode);
2 FILE *fdopen(int fd, const char *mode);
```

Nello specifico

- `fopen` (standard ISO C) serve per aprire file : il primo argomento sarà una stringa che specifica il `path`, il secondo un'altra stringa che specifica le modalità di accesso.
- `fdopen` (standard POSIX) serve per aprire uno stream sulla base di un *file descriptor* (primo parametro) che possiamo ottere mediante le funzioni `open` (per file), `pipe`, `socket` (e altre).
  Qui ci concentriamo su `fopen`: per le differenze di `fdopen` in merito a `mode` (dato che in tal caso il file è stato già aperto) consultare la pagina di manuale della funzione.

Il secondo parametro di `fopen`, `mode`, riguarda le modalità di apertura del file, riassunte in tabella 11.1:

- e si aggiunge `b` (es `rb`, `w+b` o lo stesso `wb+`) si permetterà al sistema di I/O di differenziare tra un file testuale e un file binario[2]

[1] Vi è anche `freopen`, dello standard C, che apre uno stream richiudendolo se era precedentemente aperto; si usa se mai con `stdin`, `stdout` e `stderr`.

[2] Dato che il kernel UNIX non effettua tale distinzione, specificare il carattere `b` non ha effetto su sistemi del genere.

mode	Apertura file
r	apri per leggere
w	scrivi o sovrascrivi
a	*append*: apri per la scrittura al termine del file
r+	lettura e scrittura
w+	lettura e scrittura, elimina eventuali contenuti precedenti
a+	lettura e scrittura in coda, partendo dalla fine del file

Tabella 11.1: Modalità di apertura di stream

- quando viene specificato + si applicano le seguenti restrizioni:
  1. l'output su stream non può essere direttamente seguito da input, senza che si chiami `fflush`, `fseek`, `fsetpos` o `rewind`;
  2. l'input da stream non può essere direttamente seguito da output, `fseek`, `fsetpos` o `rewind` o una operazione di input che incontra l'`EOF`.

Gestione degli errori In merito alla gestione degli errori:

- se **non ci sono errori** in apertura la valutazione restituirà un puntatore a struttura `FILE *`; `FILE` è una struttura del sistema operativo che contiene le informazioni utilizzate su quella connessione.
- nel caso di **errori d'apertura** del file verrà prodotto il valore `NULL`.

Pertanto un semplice esempio di defensive programming per l'apertura di file potrebbe essere:

```
1  FILE * fp;
2  if ( (fp = fopen("path","r")) != NULL) {
3
4    do_stuff();
5
6  } else {
7
8    printf("Errore apertura file\n");
9    return 1;
10
11 }
```

11.3.2 Chiusura

È bene **chiudere lo stream** appena è terminata la necessità[3] mediante la funzione `fclose`, che accetta come unico argomento la variabile puntatore associato allo stream

[3]Dato che quasi tutti i sistemi operativi pongono un limite al numero di file che un programma può tenere aperti contemporaneamente, è buona idea liberare i puntatori a file quando non servono più. Comunque sia `fclose` è invocata automaticamente per ciascun file aperto quando un programma termina in maniera normale.

```
1 int fclose(FILE *fp);
```

Così facendo si stoppa il collegamento fra `fp` e il file (ad esempio); la variabile `fp` potrà esser riutilizzata per collegarsi ad un'altro file.
In unix almeno, quando un processo termina normalmente, attraverso la chiamata di `exit` o dal ritorno di `main`, tutti i buffer non scritti sono flushed e tutti gli stream sono chiusi.

11.4 Lettura e scrittura su stream

Una volta aperto uno stream possiamo scegliere tra funzioni di input output:

- **non formattato**: le funzioni costituiscono le funzioni più di base (es prendi un carattere, scrivi una linea, scrivi tot byte) e non prevedono la possibilità di specificare un formato da interpretare nello svolgimento delle operazioni
- **formattato**: prevedono la possibilità di specificare un formato

11.4.1 I/O non formattato

Vi sono tre tipologie di funzioni per **input/output non formattato**:

- I/O con **un carattere alla volta**: si usa, tra le altre, `fgetc` per l'input ed `fputc`
- I/O con una **linea di testo** alla volta: si usano, tra le altre, `fgets` e `fputs`
- I/O **diretto**: si usano `fread` e `fwrite` per la scrittura o lettura diretta (ciè si specifica la quantità di byte desiderata o necessaria) di dati.

11.4.1.1 I/O con un carattere alla volta

```
1 int fputc(int c, FILE *stream);
```

`fputc` scrive il carattere `c` sul flusso `stream`, restituendo il carattere scritto o `EOF` in caso di errore.

```
1 int fgetc(FILE *stream);
```

`fgetc` legge il prossimo carattere dallo `stream` e lo ritorna come `int` o `EOF` se si è giunti alla fine del file.

11.4.1.2 I/O con una linea di testo alla volta

```
1 int fputs(const char *s, FILE *stream);
```

`fputs` scrive la stringa `s` sul flusso `stream`, restituendo un valore non negativo o `EOF` in caso di errore.

```
1 char *fgets(char *s, int size, FILE *stream);
```

questa funzione memorizza nella stringa `s` al massimo `size-1` caratteri dallo `stream`, ponendo come carattere finale `\0` . La lettura si stoppa se si incontra `EOF` o `\n` (in quest'ultimo caso viene memorizzata nel buffer).

11.4.1.3 I/O diretto

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

`fwrite` scrive nello `stream`, `nmemb` elementi consecutivi di dimensione `size` byte, letti a partire dalla posizione di memoria specificata dal puntatore `ptr`.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

`fread` legge dal *file offset* (punto in cui si è arrivati) dello `stream` al massimo `nmemb` oggetti consecutivi di dimensione `size`, e li memorizza a partire dall'indirizzo puntato in `ptr`.

Queste funzioni hanno due utilizzi comuni:

- leggere o scrivere **un array**, ad esempio

```
    FILE * fp;
    char x[10] = "ABCDEFGHI\n";
    fp = fopen("asd.bin", "w");
    if (fwrite(x, sizeof(x[0]), sizeof(x)/sizeof(x[0]), fp) !=
        sizeof(x)/sizeof(x[0]))
            printf("fwrite error");
    fclose(fp);
```

```
l@a6k:~$ file asd.bin
asd.bin: ASCII text
l@a6k:~$ cat asd.bin
ABCDEFGHI
```

- leggere o scrivere **una struttura di dati**

```
    struct {
      short count;
      long total;
      char name[NAMESIZE];
    } item;

    if (fwrite(&item, sizeof(item), 1, fp) != 1)
      printf("fwrite error");
```

Il **problema** con della lettura di dati binari è che essa è possibili solamente da dati che sono stati scritti dallo stesso sistema. Leggere dati binari scritti da un altro sistema operativo/architettura/compilatore pone problemi perchè:

- le strutture possono essere allineate in maniera differente all'interno della memoria
- la memorizzazione di interi che hanno più di un byte e valori con virgola mobile differiscono tra architetture

Carattere	Tipo argomento	Stampato come
`d`	`int`	intero decimale
`c`	`char`	singolo carattere
`s`	`char *`	stringa
`f`	`double`	numero reale
`p`	`void *`	puntatore

Tabella 11.2: Alcuni caratteri di conversione per `printf`.

11.4.2 I/O formattato

Le funzioni principali di output che si vedranno sono `fprintf` e `snprintf` per l'output, `fscanf` e `sscanf` per l'input.

11.4.2.1 Output

```
1 int fprintf(FILE *stream, const char *format, ...);
```

fprintf è una versione generalizzata di `printf` che permette di scrivere non solo su `stdin` ma su qualsiasi stream dopo conversione operata secondo `format`. Restituisce il numero di caratteri visualizzati.

```
1 int snprintf(char *str, size_t size, const char *format, ...);
```

`snprintf` è equivalente a `fprintf`, eccetto che i caratteri (al massimo `size-1`) sono immagazzinati nella stringa `str` (invece che in uno stream) la quale viene terminata con `\0`.

Per entrambe le funzioni, la stringa `format` contiene due tipi di oggetti: caratteri ordinari, che vengono copiati sul flusso in uscita, e *specifiche di conversione*, che determinano come viene convertito e visualizzato il corrispondente argomento. Ogni specifica di conversione inizia con un % e termina con un *carattere di conversione* (i principali caratteri di conversione sono posti in tabella 11.2).

Tra `%` e il carattere di conversione possono esserci, nell'ordine:

- un segno meno (-) per *l'allineamento* a sinistra
- un numero che specifica *l'ampiezza* minima del campo: l'argomento sara' visualizzato in un campo di ampiezza almeno pari a questa. Se necessario, saranno aggiunti degli spazi alla sua sinistra (o a destra, se si vuole l'allineamento a sinistra) per rispettare l'ampiezza del campo
- un punto, che separa l'ampiezza del campo dalla sepcifica del grado di precisione
- un altro numero, il *grado di precisione*, che specifica: per le stringhe il numero massimo di caratteri visualizzabili; per i numeri a virgola mobile il numero di cifre dopo il punto decimale; per gli interi il numero minimo di cifre.
- `h` se l'intero deve esser visualizzato come `short`, o `l` come long.

Ampiezza e grado di precsione possono esser specificati anche come `*`, e in questo caso il valore sarà dato dall'argomento correlato (che deve esser un `int`). Ad esempio per visualizzare un massimo di "max" caratteri di una stringa "gigi", si puo' dare:

```
int max=2;
printf("%.*s\n", max, "gigi");

$ ~/prog/c/src : tester
gi
```

11.4.2.2 Input

```
1 int fscanf(FILE *stream, const char *format, ...);
```

`fscanf` è la versione generalizzata di `scanf`: legge i caratteri in ingresso dallo `stream` specificato, li interpreta secondo le specifiche di composizione `format`, e ne memorizza i risultati negli argomenti specificati.

```
int sscanf(const char *str, const char *format, ...);
```

`sscanf` è equivalente a `fscanf`, eccetto che i caratteri in ingresso sono presi dalla stringa `str`, invece che da uno stream.

L'argomento `...` consiste in un set di puntatori a variabili separati da virgola, che immagazzineranno il risultato dell'estrazione.
La stringa `format` specifica la struttura del testo da cui estrarre e cosa estrarre: puo' contenere

- spazi bianchi[4]: gli spazi nel corrispondente input vengono saltati;
- caratteri normali: devono corrispondere letteralmente ai medesimi caratteri in ingresso[5];
- *specifiche di conversione*: specificano stabiliscono l'interpretazione da dare al campo in ingresso[6] e che valore andare a memorizzare a partire dalla sequenza estratta[7].

Le specifiche di conversione iniziano con `%`. Seguono:

- il carattere (facoltativo) `*` di soppressione dell'assegnamento. Normalmente il risultato è collocato nella variabile a cui punta l'argomento, ma se vi è soppressione non viene fatto nessun assegnamento.
- un numero (facoltativo), che precisa l'ampiezza massima del campo;
- i caratteri `'h'` o `'l'` che indicano la dimensione dello spazio riservato alla memorizzazione del dato in ingresso;

[4] Vi appartengono spazi bianchi, il carattere di tabulazione verticale o orizzontale, newline ritorno del carrello e salto di pagina

[5] Se ad esempio dobbiamo estrarre le componenti di una data nel formato gg-mm-aaaa, possiamo specificare `%d-%d-%d`, che include anche i trattini

[6] Ad esempio la sequenza `1.5` potrebbe benissimo essere intero (1) e poi float (`.5`), oppure solo `float` (per `1.5`).

[7] Ad esempio se optiamo per il valore con la virgola lo salviamo desideriamo produrre un `float` o un `double` da tale estrazione?

Carattere	Dati in ingresso	Argomento (memorizzazione)
`d`	intero decimale	`int *`
`c`	carattere	`char *`
`s`	stringa	`char *`
`f`	virgola mobile	`float *`

Tabella 11.3: Alcuni caratteri di conversione per `scanf`

- un *carattere di conversione* (obbligatorio): i caratteri di conversione principali sono presentati in tabella 11.3. Nota: `d` può esser preceduto da `h` per `short int`, o da `l` per `long` compare nella lista degli argomenti. Similmente `f` può esser preceduta da `l` per indicare che la lista di argomenti contiene un puntatore a `double` anziche a `float`.

Alcuni *esempi*, dato il loro input:

```
25 Dicembre 2009

int day, year;
char monthname[20];
scanf("%d %s %d", &day, monthname, &year);

04/11/1983

int day, month, year;
scanf("%d/%d/%d", &day, &month, &year);
```

11.5 Funzioni di supporto

11.5.1 Posizionamento

Mentre si elabora uno stream il sistema operativo mantiene in memoria la posizione del byte in corrispondenza della quale sarà eseguita la prossima azione di lettura o scrittura. Modificando questa sarà possibile muoversi nella struttura di dati liberamente, realizzando un accesso efficiente in lettura e scrittura.

```
int fseek(FILE *stream, long offset, int whence);
```

`fseek` imposta la posizione corrente per `stream`: un'operazione successiva di lettura o scrittura accederà ai dati a partire da quella posizione. Restituisce un valore non nullo in caso di errore.
Se lo stream è

- binario: la posizione corrisponde a `offset` byte a partire da `whence`; quest'ultimo parametro può essere `SEEK_SET` (inizio del file), `SEEK_CUR` (posizione corrente) o `SEEK_END` (fine del file);
- testo: `offset` deve valere 0, o un valore restituito da `ftell`.

```
long ftell(FILE *stream);
```

`ftell` restituisce la posizione corrente del file `stream`, o `-1L` in caso di errore. Per uno stream:

- binario: il valore corrisponde al numero di byte dall'inizio del file;
- testo: valore inutile, strumentale solo a `fseek` per il riposizionamento.

11.5.2 Stato delle operazioni e gestione degli errori

Molte funzioni della libreria, nel caso incorrano in errori o incontrino la fine del file, attivano indicatori dello stato delle operazioni. Ecco alcune funzioni utili per sfruttare tali informazioni.

```
1 int feof(FILE *stream);
```

`feof` verifica se nello `stream` fornitogli si è raggiunto (è impostato l'indicatore di) `EOF`; se sì ritorna un valore diverso da 0, 0 se no. Es idioma di uso corrente per procedere fino a che non si è arrivati a `EOF` per `stdin`:

```
while ( ! feof(stdin) )  {...}
```

```
1 int ferror(FILE *stream);
```

`ferror` restituisce un valore diverso da 0 se si sono verificati errori nello `stream` dato in argomento (ovvero se è attivo l'indicatore di errore).

```
1 void perror(const char *s);
```

`perror` serve per gestire gli errori; stampa un messaggio sullo standard error, che descrive l'ultimo errore contenuto in `errno` (`errno` è una variabile di sistema speciale che viene settata, se una system call non riesce ad eseguire il proprio compito, con un intero identificativo del tipo di fallimento avvenuto):

```
1 FILE * fp;
2 if ((fp=fopen("foo.txt", "r"))==NULL){
3    perror("fopen(foo,txt)");
4    return(1);
5 }
```

```
l@a6k:~$ touch foo.txt
l@a6k:~$ ./a.out
l@a6k:~$ chmod -r foo.txt
l@a6k:~$ ./a.out
fopen(foo.txt): Permission denied
l@a6k:~$ mv foo.txt bar.txt
l@a6k:~$ ./a.out
fopen(foo.txt): No such file or directory
```

La formattazione dei **messaggi d'errore** è una cosa che permette un più veloce debugging: i messaggi d'errore in un programma non interattivo dobrebbero avere una sintassi del tipo:

```
nome_programma:nome_file_c:numero_linea: messaggio
```

Per scrivere qualcosa di simile si sfruttino alcune macro definite dallo standard, e in particolare:

- `__LINE__`: numero di linea corrente nel codice sorgente (costante intera);
- `__FILE__`: nome del file sorgente (stringa).

Ad esempio:

```
FILE * fp;
char ermsg[1024] = {0};

if ((fp=fopen("foo.txt", "r"))==NULL){
   sprintf(ermsg, "mioprog:%s:%d:fopen(foo.txt)",
           __FILE__,
           __LINE__ -3
      );
   perror(ermsg);
```

```
l@a6k:~$ ./a.out
mioprog:prova.c:8:fopen(foo.txt): No such file or directory
```

In programmi interattivi si può anche evitare di mettere il nome del programma (ovvio); così non è per i programmi non interattivi, soprattutto se sono posti all'interno all'interno di una *pipe*.

11.5.3 Creazione di file temporanei

```
1 FILE *tmpfile(void);
```

`tmpfile` apre un file temporaneo unico in modalità `w+b` che verrà automaticamente eliminato quando chiuso o il programma termina.

Capitolo 12

Il preprocessore

Il preprocessore è un sostitutore di testo che interviene prima della compilazione; tutte le direttive del preprocessore iniziano con `#`, che può esser preceduto solo da spazi bianchi.

12.1 `#include`

L'inclusione agevola l'organizzazione delle istruzioni `#define` e delle dichiarazioni (es. variabili globali). Ogni riga di testo ambivalentemente nella forma

```
1 #include "nomefile"
2 #include <nomefile>
```

è sostituita dai contenuti di `nomefile`. Si usa tipicamente per gli *header*, di estensione `.h`, ma anche per i file `.c`.
I caratteri che racchiudono `nomefile` determinano dove il preprocessore va a cercare i file da sostituire:

- se `nomefile` risiede in `/usr/include` (path di default delle librerie in Linux), si può specificare il *path* a partire da tale cartella tra `<>`;
- altrimenti per includere un generico file, indipendentemente da dove essa sia posta nel filesystem, si pone il suo path tra virgolette: se relativo verrà interpretato a partire dalla directory del file sorgente considerato.

Ad esempio per includere `/usr/include/gsl/random.h` si può utilizzare:

```
#include <gsl/random.h>
#include "/usr/include/gsl/random.h"
#include "../../usr/include/gsl/random.h"  /* dalla $HOME */
```

Un file incluso può a sua volta contenere direttive `#include`.
Spesso si trovano numerose direttive `#include` all'inizio di un file sorgente, per allegare istruzioni `#define` e dichiarazioni comuni (strutture, unioni, prototipi, variabili globali).

12.2 #define

Presenta la sintassi

```
1 #define nome testo_sostitutivo
```

e serve per creare:

1. macro semplici (per K&R) o costanti simboliche (DD);
2. macro con argomenti (per K&R) o macro (DD).

Ha l'effetto che ad ogni occorrenza dell'oggetto `nome` sostituisce `testo_sostitutivo`.
Il testo sostitutivo è costituito dal testo residuo della riga; una definizione lunga può esser spezzata su più righe, spezzandola e ponendo una *backslash* alla fine di ogni riga ad eccezione dell'ultima.
La sostituzione avverrà dal punto di definizione sino al termine del file sorgente.
Diverse "funzioni" della libreria standard in realtà sono macro con argomenti; benefici delle macro è che non richiede il tempo di una chiamata.
Una definizione può usare definizioni precedenti

12.2.1 Macro semplice

Viene definita tipicamente quando `testo_sostitutivo` è una costante. Per esempio

```
#define PI 3.14159
```

sostituirà tutte le occorrenze di `PI` nel sorgente con `3.14159`. Qualora si avesse necessità di modificare tale costante si potrà farlo modificando solamente il testo di sostituzione della `define`. *Errori tipici*:

```
#define FOO 3.1;       /* sbagliato */
#define FOO = 3.1      /* sbagliato */
```

Nel primo caso anche il punto e virgola verrà posto nella sostituzione, creando guai nell'interpretazione, poi; nel secondo caso verrà sostituito anche l'uguale.

Le sostituzioni coinvolgono solo token e non riguardano le stringhe tra virgolette: ad esempio

```
#define FOO bar
...
printf("FOO");
```

non stamperà `bar`.

Un esempio del fatto che una definizione può usare definizioni precedenti:

```
#define TRUE 1
#define FALSE !TRUE
```

Costanti simboliche predefinite L'ANSI C mette a disposizione delle costanti simboliche predefinite; gli identificatori di ognuna di esse cominciano e terminano con due underscore, alcune delle quali sono presentate in tabella 12.1.

Identificatore	Significato
`__LINE__`	numero della riga corrente nel codice sorgente
`__FILE__`	nome del file sorgente
`__DATE__`	data di compilazione (stringa)
`__TIME__`	ora di compilazione

Tabella 12.1: Alcune costanti ANSI predefinite

12.2.2 Macro con argomenti

Se invece `nome` ha una specificazione simile a una chiamata di funzione (ovvero con argomenti posti tra parentesi), allora si ottiene una *macro con argomenti*, in cui il testo sostitutivo cambia a seconda dei parametri della macro utilizzati nel sorgente. La macro

```
#define MAX(a,b)  (  (a) > (b)  ?  (a) : (b)  )
```

con `a` e `b` parametri della macro, sarà utilizzata nel sorgente

```
int a = MAX(3,2);
```

espandendolo in

```
int a = ( (3) > (2)  ?  (3) : (2)  )
```

la cui valutazione condurrà ad assegnare correttamente il valore 3 ad `a`.

Importante è che nella definizione della macro:

- `MAX` sia *uppercase*, per riconoscere facilmente all'interno del sorgente che si tratta di una macro e non di una funzione.
- utilizzare copiosamente le parentesi sia per inglobare i singoli argomenti della macro, che per il risultato nel suo complesso, per non ottenere valutazioni erronee dovute alla precedenza spontanea degli operatori: ad esempio

  ```
  #define square(x) x*x
  ```

  crea problemi se invocata come `square(z+1)`; stessa cosa se il testo sostitutivo è una costante negativa, può inficiare la correttezza delle valutazioni successive.
- prestare estrema attenzione agli spazi, infatti mentre con

  ```
  #define a(y) a_expanded(y)
  ```

  un `a(x);` si espande correttamente in

  ```
  a_expanded(x);
  ```

  una definizione del tipo

  ```
  #define a (y) a_expanded (y)
  ```

un `a(x);` si trasforma invece in

```
(y)  a_expanded (y)(x);
```

che non è probabilmente quello che il programmatore intendeva.

Un esempio del fatto che una definizione può usare definizioni precedenti:

```
#define PI 3.1415
#define SQUARE(x) ((x)*(x))
#define CIRCLE_AREA(raggio)  ( SQUARE(raggio) * PI )
```

12.2.3 #undef

Si può annullare l'effetto di una eventuale definizione precedente tramite la direttiva `#undef`, solitamente al fine di garatire che un certo identificatore denoti una funzione, e non una macro, es

```
#define getchar() getc(stdin)
...
#undef getchar
...
int getchar(void){..}
```

Pertanto una macro rimane visibile fino alla fine del file o fino ad un `#undef`; una volta dimenticato, un nome potrà esser ridefinito con `#define`.

12.2.4 Gli operatori ## e

Un nome di parametro preceduto da `#` nel testo sostitutivo, il valore fornito alla chiamata sarà trasformato in una stringa tra virgolette che contiene l'argomento:

```
#define HELLO(foo)   printf("Hello" #foo)
HELLO(world);
```

sarà equivalente a

```
printf("Hello" "world");
```

L'operatore `##` (binario) concatena due parametri:

```
#define PARAMCAT(x,y)  x ## y
```

ad esempio nel programma seguente `PARAMCAT(nome,1)` sarà sostituito da `nome1`.

12.3 #if e compilazione condizionale

L'inclusione condizionale, realizzata con `#if`, permette di includere codice in modo selettivo, a seconda del calore delle condizioni esaminate.
La direttiva `#if` valuta una espressione costante intera (che non può comprendere però istruzioni `sizeof`, cast o costanti `enum`). Se l'espressione è diversa da 0, le righe a seguire sono accluse al codice da compilare fino a che non si giunge ad una direttiva `#endif`, `#elif` o `#else`. Ad esempio, la sequenza esamina la costante `SYSTEM`, che deve esser definita in precedenza da qualche parte (o dal compilatore), per decidere quale header includere.

```
#if SYSTEM == LINUX
    #define HDR "linux.h"
#elif SYSTEM == WINDOWS
    #define HDR "windows.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Ad esempio può essere utilizzato per non compilare sezioni di codice che contengono commenti (dato che non è possibile porre due volte tra commento la stessa sezione)

```
#if 0
   old_code();
#endif
```

Utilizzo comune di `#if` è quello per garantire che un header sia incluso solo una volta nel processo di compilazione.
L'espressione `defined(nome)` in una direttiva `#if` vale 1 se `nome` è stato definito in precedenza, 0 altrimenti.

```
#if !defined(HDR)
#define HDR
... (contenuti di hdr.h)
#endif
```

Scrittura equivalente si ha utilizzando l'abbreviazione `#ifndef`:

```
#ifndef HDR
#define HDR
...
#endif
```

è possibile utilizzare anche l'abbreviazione `#ifdef`; rispettivamente verificano se un nome non è o è stato definito in precedenza.

Ogni `#else` ed `#endif` dovrebbe avere un commento esplicativo soprattutto se l'`#if` che viene prima è molto più su. Nel caso di un `ifdef`

```
/* 1 */
#ifdef foo
  ...
#endif /* foo */

/* 2 */
#ifdef foo
  ...
#else /* not foo */
  ...
#endif /* not foo */
```

Nel caso invece di un `ifndef` risulta più facile leggere qualcosa del genere:

```
/* 3 */
#ifndef foo
  ...
#endif /* not foo */

/* 4 */
#ifndef foo
  ...
#else /* foo */
  ...
#endif /* foo */
```

12.4 Macro e linea di comando

É possibile specificare in sede di compilazione del programma la definizione di costanti; questo fornisce un po' di flessibilità. Si utilizza l'opzione `-D` del compilatore. Ad esempio

```
gcc -DLINELENGTH=80 prog.c -o prog
```

equivale a

```
#define LINELENGTH 80
```

All'interno del sorgente (`prog.c`), però, qualsiasi `#define` o `#undef` avrà precedenza su ciò che è stato specificato nel comando di compilazione.

È possibile anche specificare simboli senza valore (sarà assunto ad 1), come ad esempio in:

```
gcc -DDEBUG prog.c -o prog
```

L'utilizzo di un tale costrutto è utile per il debugging. Ad esempio si possono porre sezioni di codice come:

```
#ifdef DEBUG
     print("Debugging: Program Version 1\");
#else
     print("Program Version 1 (Production)\");
#endif
...

x = y *3;
#ifdef DEBUG
     print(``Debugging: Variables (x,y) = \",x,y);
#endif
```

e più in generale differenziare la compilazione finalizzata al debugging da quella per l'utilizzo.

12.5 #error

Accompagnato da una testo che può o meno esser tra virgolette, stampa il messaggio di errore e genera un errore di compilazione (facendola fallire) qualore il flusso del controllo del preprocessore vi giunga.

```
#ifdef WINDOWS
    ... /* Windows specific code */
#elif defined(UNIX)
    ... /* Unix specific code */
#else
    #error "Operating system not supported... sorry."
#endif
```

12.6 assert

La macro `assert`, presente in `assert.h`, controlla il valore di una espressione. Qualora sia falso, `assert` visualizzerà un mesaggio di errore e richiamerà la funzione `abort`, che terminerà l'esecuzione di un programma.

```
#include <assert.h>

int main (void) {
   int a = 1;
   assert(a==2);
   return 0;
}
```

```
l@a6k:~$ ./a.out
a.out: prova.c:5: main: Assertion `a==2' failed.
Abortito
```

Qualora la direttiva del preprocessore

```
#define NDEBUG
```

appaia in un file sorgente in cui sia stato incluso `assert.h`, tutte le asserzioni del file saranno ignorate.

Parte II

Misc

Capitolo 13

Miscellanea

13.1 Tips

Come interpretare le dichiarazioni Le dichiarazioni[1] si leggono:

- parti dall'identificatore più a sinistra e poi leggi secondo le regole di precedenza
- la precedenza dalla più alta alla più bassa è:
  1. parentesi che raggruppano assieme parti di una dichiarazione
  2. operatori postfissi:
     - parentesi `()` indicano una funzione e
     - parentesi graffe `[]` indicano un array
  3. operatori prefissi: l'asterisco si legge "puntatore a"
- se un qualificatore `const` o `volatile` è in prossimità di uno specificatore di tipo (es `int`, `long` ecc) si applica a tale specificatore di tipo. Alternativamente `const` e/o `volatile` si applica all'asterisco del puntatore all'immediata sinistra dello specificatore `const`/`volatile`

Ad esempio

```
1  char * const *(*next)();
2  /* next e' un puntatore a funzione che ritorna un
3     puntatore ad un puntatore const ad un carattere. */
4
5  char (*j)[20];
6  /* j e' un puntatore ad un'array di 20 char */
7
8  char *(*c[10])(int **p)
9  /* c e' un array di 10 puntatori a funzioni che
10    prendono un int**p come argomento e
11    restituiscono un puntatore a char */
```

[1] PVDL, pag74

static nelle funzioni Usare `static` copiosamente nelle funzioni (valore di ritorno) che non serve rende disponibile ad altri moduli/file per garantire il principio di minimo privilegio. (Pvdl, pag 42)

Usare le parentesi in ogni dove Non bisogna conoscere, ne considerare, le precedenze dei vari operatori C.

Uso degli interi unsigned Nella scelta di interi preferire interi con segno (suggerito da Van Der Linden). Questo perchè nelle espressioni aritmetiche, quando si mischiano operandi con e senza segno, il tipo di risultato è con o senza segno a seconda della dimensione del tipo degli operandi (si applicano regole della sezione 6.2.1.1).
Se si vuole evitare problemi evitare complessità non necessaria minimizzando l'uso di `unsigned`; ovvero non usarlo solamente perchè la quantità rappresentata non può essere negativa (es età). Usare `int` e non ci si dovrà preoccupare.
Usare gli `unsigned` solo per bitfields e maschere binarie; **utilizzare i `cast` nelle espressioni che mischiano tipi differenti** per evitare di lasciare al compilatore la scelta del tipo del risultato.
Un esempio delle sottigliezze che originano problemi segue:

```
1  #include <stdio.h>
2
3
4  int main(void){
5
6      int numbers[10] = {0};
7
8      printf("%d\n", (sizeof(numbers) / sizeof(numbers[0])) > -1 );
9      return 0;
10
11 }
```

In compilazione da `warning` e in esecuzione stampa 0

```
test.c: In function ‘main’:
test.c:8:58: warning: comparison between signed and unsigned integer
expressions [-Wsign-compare]
    printf("%d\n", (sizeof(numbers) / sizeof(numbers[0])) > -1 );
                                                          ^
l@p6k:~$ ./a.out
0
```

Questo perchè il test sta comparando un intero signed con uno unsigned; `-1` viene per qualche motivo promosso ad `unsigned int` che nella fattispecie è un numero altissimo (dato il segno negativo).
Quindi alternativamente:

- limitare gli `unsigned`
- tenere i warning accessi
- effettuare il confronto effettuando dei cast

```
printf("%d\n", ((int) (sizeof(numbers) / sizeof(numbers[0]))) > -1 );
```

- evitare di mischiare unsigned e signed nei confronti

Macro vs funzioni inline Al posto di mimare funzioni con macro scrivere funzioni con `inline` ; riservare le macro per la sostituzione di costanti e magic number (suggerito da CERT)

13.2 Modularizzazione dei programmi

Prendiamo in considerazione per migliore manutenibilità la necessità di separare il programma nel suo complesso in più file sorgente.
Il problema si inquadra nella **modularizzazione del software**, ovvero in un metodo di organizzare programmi di dimensioni non banali in parti più piccole, i **moduli**:

- ogni modulo ha una interfaccia verso altri *moduli client* che specifica come i servizi del modulo sono forniti a terzi
- ogni modulo ha una parte di implementazione separata della quale i moduli client possono anche non avere conoscenza

Per modularizzare del software in C porre innanzitutto la funzione **main** nel file `main.c` per veloce reperimento.
Considerando ogni singolo modulo: dopo avere trovato un nome significativo (es qui vediamo il modulo `foo`) occorre separare le interfacce dalle implementazioni:

- le **interfacce** vanno poste file `foo.h`: in questo file poniamo solamente dichiarazioni di costanti, tipi e typedefs, strutture, variabili globali, macro (con define), istruzioni `pragma` e prototipi di funzione che devono poter essere visibili ad altri moduli.
  Le variabili che devono poter essere utilizzate anche da altri moduli debbono essere definite con `extern` (non necessario per le funzioni che sono sempre "considerate" extern). Per evitare inclusioni molteplici da parte del preprocessore strutturiamo del file `foo.h` nella seguente forma

```
1  #ifndef FOO_H
2  #define FOO_H
3  /* Qui il contenuto di foo.h */
4  #endif /* FOO_H */
```

  Una volta completato, l'header diviene l'interfaccia tra moduli; includendo l'header si ottiene l'accesso a tutte le strutture, prototipi di funzioni, costanti ecc di quel sottosistema

- tutte le **implementazioni** vanno poste in file `foo.c`, insieme alle definizioni di oggetti che vogliamo rendere disponibili solo all'interno del modulo. Se si desidera che una funzione sia disponibile solo all'interno di un dato modulo `.c` e non in altri, bisogna dichiarare il suo scope come `static` per far si che sia chiamabile solamente nel file c nella quale è definita; allo stesso modo per le variabili.
  Includiamo nella implementazione `foo.c` l'header corrispondente

```
1    #include "foo.h"
```

Se altri moduli necessitano degli oggetti dichiarati in `foo.h` procederemo allo stesso modo aggiungendo l'`include` di sopra (es al modulo `bar.c`).

- aggiungere a `main.c` gli header necessari

Un esempio segue; `Module1.c` è

```
1  #include "Module1.h"
2
3  static void MyLocalFunction(void);
4  static unsigned int MyLocalVariable;
5
6  void MyExternFunction(void)
7  {
8    MyLocalVariable = 1u;
9
10   /* Do something */
11
12   MyLocalFunction();
13 }
14
15 static void MyLocalFunction(void)
16 {
17   /* Do something */
18
19   MyExternVariable = 2u;
20 }
```

`Module1.h` è:

```
1  #ifndef __MODULE1.H
2  #define __MODULE1.H
3
4  extern unsigned int MyExternVariable;
5
6  void MyExternFunction(void)
7
8  #endif
```

`Module2.c` è

```
1  #include "Module.1.h"
2
3  static void MyLocalFunction(void);
4
5  static void MyLocalFunction(void)
6  {
7    MyExternVariable = 1u;
8    MyExternFunction();
9  }
```

Capitolo 14

GCC

14.1 Compilazione

Abbiamo foo.c: il seguente comando *esegue le direttive del preprocessore* (l'output viene stampato su stdout, quindi lo guardiamo con less).

```
gcc -E asd.c | less
```

Il seguente comando produce l'equivalente *assembly* (opzione `-S`)

```
gcc -S foo.c
```

La *compilazione* in file oggetto si ha mediante l'opzione `-c`

```
gcc -c foo.c
```

il risultato sarà foo.o.
Per fare il *linking* si comanda

```
gcc foo.o -o foo
```

Il **tutto** è fattibile in una sola volta con

```
gcc foo.c -o foo
```

Qualora si disponessero di **più file su cui si articola il programma** è (tipicamente una main, dei file di funzione ed uno o più header) possiamo compilare in una volta unica, creando anche l'eseguibile mediante:

```
gcc main.c function_a.c function_b.c -o myprogram
```

Non si cita l'header nel comando perchè esso viene incluso dal preprocessore in main.
Se si sviluppa a pezzi, non serve compilare tutto tutte le volte: basta compilare (e basta, non linkare) i singoli componenti del programma (mediante `-c`)

```
gcc -c main.c
gcc -c function_a.c
gcc -c function_b.c
```

Poi quando desideriamo, creare l'eseguibile con

```
gcc main.o function_a.o function_b.o -o myprogram
```

Il passo successivo a questo, in termini di automazione, consta nell'utilizzo di GNU Make.

14.2 Opzioni comuni

Importante avere **sempre l'opzione** `-Wall` (per i warning attivati, che aiutano il debugging). Ad esempio:

```
$: cat prova.c
#include <stdio.h>

intmain (void)
{
  printf ("Two plus two is %f\n", 4);
  return 0;
}

$: gcc -Wall prova.c
prova.c: In function ‘main’:
prova.c:6: warning: format ‘%f’ expects type ‘double’, but
           %argument 2 has type ‘int’

$: ./a.out
Two plus two is -0.712002
```

Gcc notifica *errori* (per cui la compilazione si interrompe, non producendo il risultato atteso) *o se richiesti warning* (in tal caso la compilazione non viene bloccata); entrambi vengono presentati nel formato `file:line-number:message`.
Altri warning, oltre a Wall, sono utili: **-W** che segnala altri tipi di errori comuni (es confronto tra due tipi diversi ecc).
L'opzione **-Wconversion** avverte se ne vi sono conversioni di tipo implicite nel programma.
Altre opzioni di buona abitudine sono **-Wshadow -Wcast-qual -Wwrite-strings**.

Per avere una **compilazione ANSI 89** con vari alert

```
gcc -Wall -ansi -pedantic asd.c
```

più in generale gcc setta lo standard da utilizzare nella compilazione mediante l'opzione `-std`.

Per vedere un resoconto del tempo impiegato nella compilazione

```
$ gcc -time file.c
# cc1 0.05 0.00
# as 0.01 0.00
# collect2 0.07 0.00
```

14.3 Compilazione per il Debug

L'opzione **-g** produce informazioni di debug su cui GDB può lavorare. Questo fa si che l'eseguibile sia di più grandi dimensioni, e adatto più che altro all'analisi del programmatore, non tanto come release. È buona regola includere -g sia nella programmazione di librerie che di eseguibili; una volta che queste avranno raggiunto una buona affidabilità togliere il -g dalla compilazione e rilasciare il binario.
Se volessimo togliere le informazioni di debug da un eseguibile compilato con -g, possiamo utilizzare `strip`

```
strip mybinary
```

E' tuttavia meglio rilasciare binari non strippati, bensi' compilati normalmente (senza -g), perchè in questo modo includono un minimo di informazioni di debug che possono esser utili a beta tester che non compilano il sorgente.

14.4 Path di ricerca

Gcc cerca librerie qui (il library search path o link path)

```
/usr/local/lib/ ...poi
/usr/lib/
```

e gli header qui (l'include path)

```
/usr/local/include/ ... poi
/usr/include/
```

in maniera ordinata; utilizza la prima occorrenza valida.
Tuttavia se le **librerie**/**header** necessarie/i al programma sono **posti in locazioni non di default**, in sede di compilazione occorrera' specificare:

- mediante `-L/path/to/dir` le directory da aggiungere al library path
- mediante `-I/path/to/dir` le directory di include aggiuntive

In entrambi i casi i path posti saranno analizzati prima di quelli di default per gli header e le librerie da utilizzare. Questo problema è risolvibile anche mediante **variabili d'ambiente Bash**

14.5 Variabili d'ambiente

Gcc utilizza alcune variabili d'ambiente (da porre in `.bash_profile`) nel proprio funzionamento:

- `TMPDIR`: directory temporanea dove porre file intermedi/temporanei.
- `LIBRARY_PATH`: lista di directory separata da `:` dove effettuare la ricerca di librerie (aggiuntive);
- `C_INCLUDE_PATH`: lista di directory separata da `:` dove effettuare la ricerca di header;

- `CPLUS_INCLUDE_PATH` idem come sopra ma per il C++;
- `LD_LIBRARY_PATH`: path delle shared libraries.

Ad esempio:

```
C_INCLUDE_PATH=/opt/gdbm-1.8.3/include  ... per gli header c
C_INCLUDE_PATH=.:/opt/gdbm-1.8.3/include:/net/include ... per piu' cartelle

CPLUS_INCLUDE_PATH=/opt/gdbm-1.8.3/include  ... per gli header c++

LIBRARY_PATH=/opt/gdbm-1.8.3/lib   ... per le librerie (sia c che c++)
LIBRARY_PATH=.:/opt/gdbm-1.8.3/lib:/net/lib    ... per piu' cartelle
```

Ciò che viene specificato qui, puo' non esser specificato in sede di compilazione mediante opzioni del comando.
Queste locazioni verranno analizzate in cerca di header/librerie dopo quelle fornite eventualmente da linea di comando, e prima di quelle di default.

Capitolo 15

Debugging, GDB e Valgrind

Gdb può esser utilizzato sia per il C che per il C++.

15.1 Introduzione al debugging

Vi sono tre modi per effettuare il debugging:

- utilizzare delle `printf` dove si vuol controllare il valore di alcune variabili: diventa però difficile mantenere tutto se la quantità di codice del progetto aumenta.

- impostare dei parametri da linea di comando e degli `if(debug)` che introducano sezioni di check eseguite solo se i relativi parametri sono stati attivati (es si può specificare -Dabc, dove a b e c sono differenti aspetti di cui vogliamo eseguire il codice di debug

- utilizzare un debugger

Alcune note sul debugging in generale: utilizzare sempre gli strumenti migliori:

- per un Segfault o problemi coi puntatori serve il debugger

- per problemi di memoria il valgrind

Chiedersi se si sta affrontando il problema nella maniera piu' semplice possibile o se sia possibile riformulare la situazione in maniera più semplice; non focalizzarsi necessariamente sul codice già scritto, ma eventualmente riscriverlo puo' condurre a versioni più pulite e migliori.
Evitare copia e incolla di sezioni di codice: fare funzioni.
Scrivere funzioni e testarle una ad una. Riutilizzare le funzioni già scritte per passati progetti. Quando si debugga una sezione di codice commentare le altre non strettamente necessarie, per evitare di concentrarsi su una sezione che magari funziona (e l'errore è dovuto alle altre).

15.2 Memoria

Per poter utilizzare un debugger è meglio ripassare alcuni concetti sulla memoria di un Pc. Quando un processo viene creato, il kernel gli alloca un po' di memoria, che possa contenere i dati necessari alle elaborazioni: le uniche cose che sanno i processi è quanta memoria hanno a disposizione e che indirizzo ha. Ogni chunk di memoria presenta la seguente struttura:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
high address       *args and env vars*        command line args
                                              and env vars
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   *stack*  (cresce vs basso)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   *unused mem*
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   *heap  (cresce vs alto)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   *uninitialized data        initialized to
                    segments*                 zero by exec
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   *initialized data          read from the
                    segments*                 program file by exec
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
low address        *text segments*            read from the
                                              program file by exec
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- il text segments contiene il codice che deve esser eseguito: è settato a memoria read only cosicchè il programma non possa modificare il proprio comportamento accidentalmente, una volta compilato
- l'initialized data segment contiene le variabili globali inizializzate dal programmatore
- l'initialized data segment contiene le variabili globali non inizializzate; tutte le variabili qui sono inizializzate a puntatori a NULL prima che il comando inizi la propria esecuzione
- lo stack contiene lo stack dei frames del programma; quando è necessario aggiungere un frame (perchè si chiama una funzione, ad esempio) lo stack cresce verso il basso
- l'heap è quella porzione di memoria che server per le allocazioni dinamiche di `malloc` e `new`. Cresce verso l'alto

Una volta che abbiamo compilato del codice in binario (libreria, eseguibile o altro) possiamo vedere la dimensione delle varie componenti di memoria mediante `size`

```
luca@eee:~$ gcc 08_11.c
luca@eee:~$ size a.out
   text   data    bss    dec    hexfilename
   1698    284      8   1990    7c6a.out
```

- text è text segment
- data è initialized e uninitialized insieme
- bss è l'uninitialized (da solo)
- dec ed hex sono la dimensione del file in decimali ed esadecimali
- filename è intuibile

Si noti che la somma dei primi 3 da 1990 ossia il risultato di dec; in questo esempio poi non ci sono ne stack ne heap (che sono sezioni della memoria di un processo che crescono quando esso è eseguito, non esistono in un binario "fermo").

Altre info si possono ottenere mediante `objdump -x a.out`.

15.3 Symbol table e compilazione per il debugging

Bisogna innanzitutto compilare il binario esplicitamente per il debugging:

```
gcc -ggdb3 hello.c
```

così facendo incorporiamo nell'eseguibile informazioni necessarie (una symbol table che ci permetta di chiamare le variabili col loro nome in sede di debugging e non con le locazioni di memorizzazione interna). Un binario con symbol table generalmente occupa spazio maggiore su disco, ma non richiede tempi di esecuzione più lunghi (la symbol table è letta solo all'interno di gdb e solo al suo interno l'esecuzione potrà risultare rallentata)

Per eliminare la symbol table su un binario si utilizzi `strip`.

Come regola, evitare le ottimizzazioni (flag `-O9` quando si compila per il debug.

15.4 Comandi comuni di Gdb

Tutti i comandi e i parametri di comando di GDB possono esser abbreviati fino a che non sono ambigui. Per la guida in linea si utilizzi `help`.

E' meglio eseguire un solo processo di Gdb alla volta.

Se si desidera impostare delle opzioni si veda `help set`: il file di init di gdb è `$HOME/.gdbinit`. I vari set li comandi di set li possiamo metter qui e saranno eseguiti ogni volta alla partenza. Li dentro si commenta con `#`.

Per iniziare a debuggare avviare `gdb` con argomento il nome dell'eseguibile da controllare:

```
gdb a.out
```

Se oltre al nome si passeranno parametri aggiuntivi, questi saranno interpretati come parametri del programma da debuggare. Alternativamente si invochi `gdb` e in seguito si comandi

```
load nomeprog
```

Per eseguire il programma si comandi `run`; comandare `run asd qwe` eseguirà il programma con, come opzioni da linea di comando, si utilizzerà `asd` e `qwe`. La volta successiva che si farà correre il programma gdb utilizzerà gli stessi parametri; se si vorrà cancellarli si dia `set args`. Senza parametri cancella tutti gli argomenti passati in precedenza (altrimenti servirebbe per settarli, i parametri).

Premere Ctrl-c in un qualsiasi momento di esecuzione del programma ci riporterà al prompt di GDB. Per stoppare la `run` (magari per farla ripartire da 0) si comandi `kill` e poi ancora `run`.

Per listare dieci righe di programma si utilizzi `list`, abbreviabile a `l`

```
(gdb) list
```

Comandato più volte, `list` fa vedere in successione di 10 righe tutto il programma; se invece diamo `list -` torniamo indietro nello stampare. Se si vuole guardare attorno ad una determinata riga (un po' di codice sopra e un po' sotto) si dia come parametro la riga di interesse

```
(gdb) list 20
```

`list a,b` dove `a` e `b` sono numeri, lista dalla linea `a` alla linea `b`. `list a,` stampa dieci linee a partire da `a`; `list ,a` stampa dieci linee terminando con la linea numero `a`. Per listare una funzione

```
(gdb) list nome_funzione
```

E' possibile vedere, se il programma è composto di più file, la riga x del file y attraverso:

```
(gdb) list y:x
```

in tal caso se dopo listiamo (semplicemente con `l`, senza specificare il file), listeremo della roba di quel file y; se vogliamo tornare al file da cui saremo partiti, dobbiamo listare specificando il file di prima, o listare una funzione presente in quel file.

15.5 Breakpoint

Vi sono tre tipi di break in gdb:

- breakpoint
- watchpoint
- catchpoint

ci concentreremo sul primo.

Per impostare un breakpoint vi sono differenti modi

- si comandi `break num_linea`, o `bp #`: in questo modo si ferma la successiva esecuzione comandata da `run` fino alla riga del sorgente precedente `num_linea`. `bp +3` imposterà un bp tre linee sotto la corrente; `bp -2` due prima.

- per nome funzione: `bp asd` imposta un break all'inizio della funzione asd.
- `bp fgets.c:10` imposta alla riga 10 del file `fgets`
- `break ... if cond`: il breaking condizionale dove ... è uno dei modi precedenti e cond è una espressione valutabile (es a==1)

Generalmente si impostano più di un bp; si comanda la prima volta `run` per eseguire sino al primo, e `continue` per continuare sino al successivo (abbreviabile in `c`).
Ad ogni break viene assegnato un numero progressivo, a partire da 1. Per vedere i bp comandare

```
info breakpoints
```

abbrebiabile in `i b`:

- `Num` da l'id del bp
- `Type` il tipo (se breakpoint o altro)
- `Disp` dice cosa succede al breakpoint la prossima volta che si passa per quel break (`keep` dice che non succederà nulla ma se disable o remove cambiano la solfa).

I bp possono essere abilitati (e lo sono di default quando vengono creati) o disabilitati (in tal caso non ci si fermerà presso la linea specificata). Per disabilitare un bp:

```
disable n
```

ove n è il numero progressivo del bp; per abilitarlo

```
enable n
```

Lo stato di un bp riguardo ad abilitazione/disabilitazione si nota in `info breakpoint` nella colonna Enb (se = n, il bp è disabilitato).

Per eliminare i bp:

- per locazione si usa clear: es `clear asd` cancella il bp sulla funzione `asd`
- per identificatore cancella il bp numero specificato da info: `delete n` cancella il bp numero `n`. `delete` senza argomenti cancella tutti i bp.

15.6 Stampare il valore delle variabili

Per vedere il valore assunto da una variabile (nel frame corrente) in un dato punto del programma si utilizza `print` (abbreviabile a `p`):

```
(gdb) print a
(gdb) print myarray[5]
(gdb) print struct.label
```

Per conoscere il tipo di una variabile si utilizzi `ptype`, abbreviabile in `pt`. Per stampare le strutture si dia

```
set pretty print
```

che facilita la visualizzazione.

È possibile stampare specificando il formato di output:

```
print /FMT variabile
```

dove `FMT` può esser:

- `o` ottale
- `x` esadecimale
- `d` decimale
- `u` decimale senza segno
- `t` binario
- `f` float
- `a` indirizzo
- `c` char

Si provi a stampare una variabile char in vari modi.
Per stampare l'indirizzo di una variabile

```
p &nomevar
```

Print supporta anche l'applicazione di operatori, ad esempio.

```
p *(&nomevar)
```

che ad esempio stamperà `nomevar`.

15.7 Steppare attraverso il programma

`next` serve per eseguire la prossima riga di codice: se si specifica il parametro `n`, le prossime `n` righe di codice. Tratta le chiamate di funzione come una singola riga e quindi le esegue tutte di un botto.
Se vogliamo controllare anche le funzioni (perchè le abbiamo fatte noi e non sono quelle della libreria standard), per steppare si usi `step` che a differenza di `next` entra nelle chiamate di funzioni e steppa anch'esse.

15.8 Altro

Il comando `watch` serve per far stoppare l'esecuzione del binario da gdb quando il valore di una data variabile cambia.
`call` serve per chiamare una determinata funzione.
Per vedere dove siamo nell'esecuzione del codice si comandi `where`.

15.9 Modificare il valore delle variabili

Per modificare una variabile (del frame corrente) si usa `set variable`

```
set variable a = 1
```

15.10 Stack e backtrace

Lo stack è composto da diversi frame; in un determinato momento punto dell'esecuzione c'è un frame per ogni funzione chiamata e non ancora ritornata. Quando una funzione viene chiamata, viene aperto un nuovo frame; quando tale funzione ritorna, il frame viene chiuso.
Nel singolo frame si trova:

- lo spazio epr le variabili automatiche per le funzioni chiamate
- il numero di linea della funzione chiamante ove ritornare quando bisogna restituire il controllo
- argomenti/parametri della funzione chiamata

Per vedere i frame dello stack presenti in memoria si utilizza `backtrace` o `bt`; verranno stampate una lista di funzioni che non hanno ancora ritornato, dalla più recentemente invocata (#0) a quella più vecchia (numero maggiore).

```
(gdb) bt
#0  main () at 08_11.c:44
```

C'è un frame nello stack, il numero 0, che appartiene a main (che è stata invocata senza parametri, assenti fra parentesi). Se facciamo ancora uno step azioniamo la funzione dice:

```
(gdb) step
dice (faces=5) at 08_11.c:73
(gdb) bt
#0  dice (faces=5) at 08_11.c:73
#1  0x08048588 in main () at 08_11.c:44
```

Ora siamo nel frame di dice, una funzione chiamata da main con il parametro `faces` a 5. Si vede che in ogni frame è listata la linea di codice in cui ci troviamo (in dice siamo a 73 e in main nella linea 44. Per conoscere in quale frame ci si trova si utilizza `frame`

```
(gdb) frame
#0  dice (faces=5) at 08_11.c:73
73   if(i){
```

Per tornare in un altro frame `frame numero`.

```
(gdb) frame 1
#1  0x08048588 in main () at 08_11.c:44
44         strcat(
```

Si può utilizzare `up` e `down` per spostarci sul frame superiore o inferiore dello stack, al fine di vedere la situazione della funzione chiamante/chiamata.

```
(gdb) down
#0  dice (faces=5) at 08_11.c:73
73   if(i){
```

Capitolo 16

Librerie

16.1 Librerie statiche

Una libreria statica è una collezione di routine salvate in file oggetto precompilati, tipicamente salvati in file compressi speciali aventi estensione ".a". Queste vengono creati da file oggetto mediante il tool GNU `ar`.
Le librerie statiche sono incorporate nell'eseguibile in sede di compilazione (per cui per eseguire il programma, avremo solo bisogno del file binario creato).

16.1.1 Come creare una libreria statica

Ad esempio abbiamo la funzione mypow in mypow.c (dove non metto main), pongo il prototipo di tale funzione in mymath.h

```
emacs mypow.c mymath.h
```

Sebbene si possano mettere piu' funizoni in uno stesso file e' meglio creare un file apposta per ogni funzione c. un esempio di header potrebbe essere

```
#ifndef H_MYMATH_
#define H_MYMATH_ 1
int mypow(int base, int exponent);
#endif
```

Compilo (e basta, non linko) mypow.c (creando mypow.o)

```
gcc -c mypow.c
```

Se pero' i file di funzione diventano più di uno si può decidere di creare una libreria mediante il tool ar.
Per creare la mia libreria statica matematica, o per aggiungere file oggetto addizionali ad una libreria esistente utilizzo un comando del genere

```
ar rcs libmymath.a mypow.o mysqrt.o
```

e' meglio che il nome della libreria inizi per "lib".
Ogni volta che si aggiorna una libreria statica, è meglio (in alcuni casi questo è fatto di default) aggiornarne anche l'indice mediante

```
ranlib asd.a
```

Per vedere la roba contenuta in una libreria

```
ar t libmymath.a
```

16.1.2 Utilizzo di libreria statica

Infine affinche' un programma, chiamato main.c possa utilizzare la funzione presente in libmymath.a dobbiamo costruirlo come:

```
#include <stdio.h>
#include "mymath.h" /*includiamo in nostro header*/
int main(void)
{
   printf("%d\n", mypow(2,2) );  /*ed utilizziamo la nstr funz*/
   return 0;
}
```

Si noti che in sede di compilazione bisogna specificare dove andarla a prendere la definizione di mypow (nell'header c'e' solo il prototipo): un modo, scomodo, potrebbe essere specificare il path al file.a desiderato.

```
gcc -Wall  main.c /home/luca/libmymath.a
```

Specifichiamo pertanto il nome della libreria con l'opzione `-l` e il suo basename, qualora non fosse nelle directory standard delle librerie, con `-L`.

```
gcc -Wall main.c -lmymath -L/home/luca
```

L'opzione del compilatore `-lNAME` cercherà di linkare alla libreria di nome `libNAME.a` nella directory standard delle librerie. Directory aggiuntive di ricerca possono esser specificate mediante `-L` o variabili d'ambiente.
Una volta che l'header e la libreria avranno raggiunto una dimensione consistente si potrà porli nelle directory di default, per semplificare l'inclusione e la compilazione: il primo sotto `/usr/include` eventualmente con una sua cartella, se vi sono piu' header; la libreria andra' invece in `/usr/lib` (credo).

16.1.3 Ordine di specificazione delle librerie

Generalmente un programma più complesso avrà necessità di molteplici librerie: occorre spendere due parole sul **comando di compilazione**, in relazione all'ordine che deve essere mantenuto tra le sue componenti.
Il *comportamento tradizionale dei linker* è di cercare funzioni esterne richieste da un programma o da una libreria stessa **da sinistra a destra**: ad esempio un programma data.c che utilizza funzioni di libglpk.a, la quale a sua volta utilizza funzioni di libm.a (entrambe nelle dir di default) dovrebbe esser compilato come

```
gcc -Wall data.c -lglpk -lm
```

Ipotizzando poi che libm utilizzi roba di libglpk allora sarebbe opportuno mettere un altro -lglpk dopo -lm.

I linker moderni dovrebbero cercare in tutte le librerie, indipendentemente dall'ordine, ma dato che alcuni non fanno così, è meglio mantenere la convenzione (da ricordare soprattutto se si incontrano strani/inattesi problemi di referenze non definite e tutte le librerie necessarie appaiono nel comando di compilazione.

16.2 Shared Library

Le shared library sono piu' avanzate delle static; le shared library non vengono incorporate nel programma in compilazione, bensì questo le carica in memoria solo al momento del proprio avvio. Quindi per eseguire il programma occorre il binario e la libreria installata opportunamente.
Questo permette, tra l'altro, di aggiornare la libreria una volta sola, propagando i miglioramenti a tutti i programmi che ne fanno utilizzo. Inoltre quando una libreria viene caricata in memoria per l'utilizzo di un dato binario, tale copia sarà utilizzabile da eventuali altri programmi che ne necessitano (al contrario una statica incorporata in tanti binari, sarebbe caricata ogni volta, insieme ai relativi eseguibili).
In generale se gcc trova due librerie, una shared e una static, con lo stesso nome, utilizza di default la shared.
Su sistemi GNU-glibc, il far eseguire un binario ELF fa si che il loader (`/lib/ld-linux.so.X`) cerchi e carichi le shared libs richieste dai programmi. Il loader solitamente cerca nelle directory specificate in `/etc/ld.so.conf`..
In realtà se le librerie sono numerose e le cartelle di path anche, dato che sono linkate in esecuzione, questa verrebbe rallentata. Quindi si utilizza un sistema di caching: quando ldconfig crea i link necessari, nel momento dell'installazione della libreria, scrive una cache in `/etc/ld.so.cache` (file binario) utilizzata dagli altri programmi che accedono a tal file. Quindi il loader guardera' a questo file e poi accedera' alla libreria necessaria.

16.2.1 Convenzioni

Bisogna seguire anche qui un po' di convenzioni riguardanti il *"soname"*, il *"real name"*, dove piazzarle ecc.

- Ogni libreria ha un nome speciale, il *soname*, che è composto da il prefisso `lib` (mancante solo nelle librerie di più basso livello), il nome della libreria e l'estensione `.so` (che sta per *shared object*), seguiti da un punto e un numero di versione della libreria che viene aumentato quando l'interfaccia cambia (es `libgsl.so.0`). Spesso il soname viene creato con un link, nella cartella di default delle libreria, legato alla libreria.

- Oltre al soname, vi è il *real name*, ossia in nome del file che contiene attualmente il codice; questo è composto da

  `soname + . + minor_number + . + release_number`

  (es `libgsl.so.0.10.0`).

- Infine vi è il nome che il compilatore (il linker) utilizza quando cerca una libreria (chiamiamolo linker name), che è il soname senza numeri

Ad esempio per la GNU GSL abbiamo, tra gli altri, i seguenti file

```
$: cd /usr/lib
$: ls -gGh libgsl*
lrwxrwxrwx 1   16 2007-11-03 09:13 libgsl.so -> libgsl.so.0.10.0
lrwxrwxrwx 1   16 2007-11-03 09:13 libgsl.so.0 -> libgsl.so.0.10.0
-rw-r--r-- 1 1,9M 2007-09-14 23:40 libgsl.so.0.10.0
```

si nota che soname e linker name sono link alla libreria (real name).
La chiave per la gestione delle librerie è proprio questa diversa specificazione di nomi: i programmi specificano il soname della libreria desiderata.

Quando si crea un libreria la si piazza nella sua postazione e si esegue `ldconfig` che crea i soname e i links necessari. `ldconfig` non setta il linker name; questo è tipicamente fatto in fase di installazione della libreria, quando viene settato come link alla versione più recente di soname o real name.

16.2.2 Creazione di una Shared library

:

- innanzitutto bisogna creare i file oggetto che verranno incorporati nella libreria, utilizzando i flag `-fPIC` o `-fpic` (pic sta per "position indipendent code"[1], una cosa necessaria per le librerie shared): l'opzione `-fPIC` genererà codice piu' grosso, meno veloce ma più portabile. `-fpic` codice piu' snello e veloce, ma con limitazioni dovute alla piattaforma. Il linker dira' se va bene al momento della creazione della libreria.
  Quando in dubbio utilizzare `-fPIC`.
  Quindi ad esempio:

  ```
  gcc -fPIC -g -c -Wall asd.c
  gcc -fPIC -g -c -Wall foo.c
  ```

- dopodichè si creerà la libreria. Un modo semplice

  ```
  gcc -shared asd.o foo.o -o mylib.so
  ```

  Utilizzando l'opzione `-Wl` per specificare la versione (credo)

  ```
  gcc -shared -Wl,-soname,your_soname \
      -o library_name file_list library_needed
  ```

  Ad esempio:

[1] http://users.actcom.co.il/ choo/lupg/tutorials/libraries/unix-c-libraries.html la spiega cosi' "Compile for "Position Independent Code" (PIC) - When the object files are generated, we have no idea where in memory they will be inserted in a program that will use them. Many different programs may use the same library, and each load it into a different memory in address. Thus, we need that all jump calls ("goto", in assembly speak) and subroutine calls will use relative addresses, and not absolute addresses. Thus, we need to use a compiler flag that will cause this type of code to be generated."

```
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 asd.o foo.o -lc
```

`-Wl` da indicazioni al linker del soname da utilizzare (le virgole non sono errore tipografico, bisogna includere caratteri non di escape per separare)

In **fase di sviluppo** si potrebbe voler creare una libreria, lasciando la versione vecchia in modo che i programmi che la utilizzino non passino subito a quella sperimentale, ma che questa venga utilizzata la un programma ad hoc per il testing.
In questo caso si può utilizzare l'opzione di link `-rpath` nella compilazione del programma adibito al testing. Questo fa si che il programma avrà nel proprio binario l'indirizzo di path dove risiede la libreria (e la libreria, ovviamente deve trovarsi laddove si è specificato con rpath). Ovviamente il path specificato verrà indagato prima di quello settato con le variabili d'ambiente o quello standard.

```
gcc -Wl,-rpath,/home/luca/ asd.c
```

16.2.3 Installazione ed utilizzo di shared libs

Si crea la libreria, la si copia in una directory default (`/usr/bin`) e si comanda `ldconfig` da root. Questo generalmente crea il link libfoo.so.1 alla libreria creata in precedenza libfoo.so.1.0. Il passo successivo è creare un link (se non è fatto di default, non lo so) al "nome-linker"

```
ln -sf libfoo.so.1 libfoo.so
```

Se si vuole installare solo le librerie di una determinata directory si può dare

```
ldconfig -n directory_dove_sta_la_libreria
```

Se non si vuole o non si può modificare le directory del path standard, bisogna installare da qualche parte la libreria e poi dire, in sede di compilazione dell'eseguibile, dove essa si trova; si può utilizzare l'opzione `-L` di gcc, o l'`rpath` o si può settare la variabile d'ambiente. La **variabile d'ambiente** per settare il library path delle shared library e'

```
LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib:/opt/gtk-1.4/lib
```

utilizzabile soprattutto in fase di creazione/debugging.

Poi per utilizzarla

```
gcc -lmialib mioprog.c -o mioprog
```

Il file linkato sara' mialib.so (che spesso e volentieri è un link a mialib.so.x.y.z, alla versione più recente); quando si eseguira' il file invece si cerchera' di caricare mialib.so.x, dove ATTENZIONE x è la versione con cui in sede di compilazione è stato linkato.

Si possono vedere le librerie shared utilizzate da un programma, comandando `ldd` sul suo eseguibile

```
ldd /bin/ls
```

Si vede che quasi tutti dipendono, almeno, da `/lib/ld-linux.so.N` (il loader delle altre librerie) e `libc.so.N` (la libreria c).
Non eseguire ldd su programmi non fidati.