# Introduzione alla programmazione in C

## Appunti delle lezioni di Complementi di Programmazione

*Giorgio Grisetti*
*Luca Iocchi*
*Daniele Nardi*
*Fabio Patrizi*
*Alberto Pretto*

*Dipartimento di Ingegneria Informatica, Automatica e Gestionale*
*Facoltà di Ingegneria dell'Informazione, Informatica, Statistica*
*Università di Roma "La Sapienza"*

*Edizione 2023/2024*

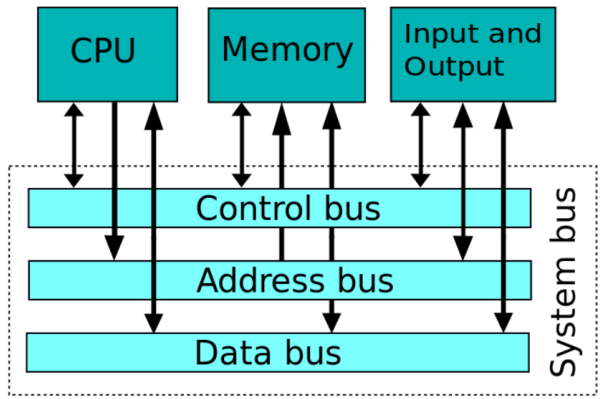

2024

# Indice

# 1. Da Python a C

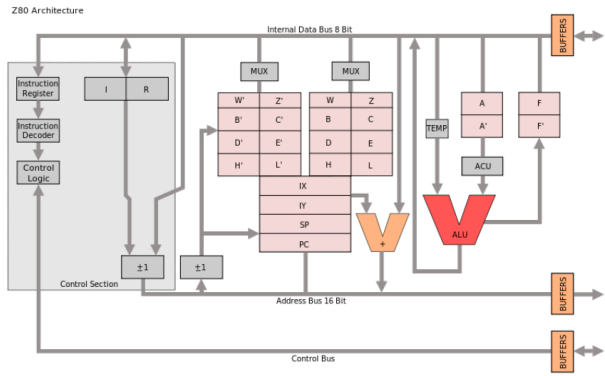
## 1.1. Architettura del calcolatore

L'architettura di un calcolatore (architettura di Von Neumann) è costituita da CPU, memoria e dispositivi di input/output connessi attraverso dei bus.




### 1.1.1. Architettura della CPU

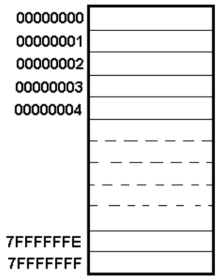
L'architettura di una CPU è costituita da una serie di registri di memoria, da un'unità di calcolo che svolge calcoli su dati contenuti nei registri, e un'unità di controllo che si interfaccia con i bus per leggere/scrivere dati in memoria.




Il trasferimento dei dati memoria-CPU è molto più lento dei calcoli all'interno della CPU.

### 1.1.2. Architettura della memoria

Per gli scopi di questo corso, la memoria (virtuale) può essere immaginata come una sequenza di celle indicizzate da un indirizzo.




### 1.1.3. Programmazione del calcolatore

La programmazione del calcolatore consiste nel sintetizzare procedure automatiche (algoritmi) per la soluzione di problemi mediante un linguaggio di programmazione.

Il programma risiede in memoria e viene eseguito tramite la CPU elaborando dati in memoria e interagendo con l'utente tramite i dispositivi di input/output.

La memoria deve essere organizzata e gestita in maniera opportuna per consentire l'esecuzione dei programmi nella CPU.

Scopo principale di questo corso: *comprendere le tecniche di programmazione per la gestione della memoria di un calcolatore durante l'esecuzione di programmi complessi*.

Esempio: somma di due numeri

Python:

```
a = b + c
print(a)
```

C:

```
int a = b + c;
printf("%d\n",a);
```

Questi due frammenti di programma svolgono la stessa funzione e sono molto simili sintatticamente, ma l'uso della memoria del calcolatore durante l'esecuzione dei due programmi è completamente diversa.

## 1.2. Linguaggi di programmazione

Esistono moltissimi linguaggi di programmazione: una vera e propria *torre di Babele*.

Infatti, per definire un linguaggio $X$ basta realizzare un programma $I$ che consente di utilizzare il linguaggio $X$, possibilmente su diversi tipi di calcolatore. Utilizzare un linguaggio significa poter eseguire programmi scritti nel linguaggio stesso.

Il programma $I$ deve tradurre i programmi scritti nel linguaggio $X$ in istruzioni direttamente eseguibili dal calcolatore. In alcuni casi, $I$ esegue direttamente le istruzioni del linguaggio.

### 1.2.1. Linguaggio macchina e linguaggi alto livello

- Linguaggio macchina
  `21 40 16 100 163 240`

- Linguaggio assemblativo (Assembly)
  ```
  iload intRate
  bipush 100
  if_icmpgt intError
  ```

- Linguaggi ad alto livello (ad es. Python e C)
  `if (intRate > 100) ...`

Il linguaggio del modello di elaboratore di Von Neumann è un esempio molto semplificato di linguaggio macchina. C è il linguaggio di alto livello più “vicino" alla macchina. Python è un linguaggio di alto livello, molto “vicino" al programmatore.

### 1.2.2. Interpreti e compilatori

Il programma *I* può essere realizzato in due diverse modalità:

- *Interprete*
- *Compilatore*

L’interprete (es. Python) considera le istruzioni del linguaggio di programmazione una alla volta traducendole direttamente in istruzioni in linguaggio macchina ed eseguendole direttamente.

Il compilatore di un linguaggio effettua la traduzione in linguaggio macchina dell’intero programma producendo un programma in linguaggio macchina. L’esecuzione del programma avviene separatamente dal processo di traduzione. C è un linguaggio compilato.

Principali vantaggi dei linguaggi compilati (ad es. C):

- Velocità di esecuzione
- Verifica presenza di errori prima dell’esecuzione

Principali vantaggi dei linguaggi interpretati (ad es. Python):

- Semplicità d’uso

### 1.2.3. Esempio programma Python

**calcololungo.py**

```
import time

s = 0
for i in range(0,100):
    s += 2**i
    time.sleep(0.1)

print "Il risultato e': "
print S
```

Individuazione di errori solo a tempo di esecuzione.

Nell'esempio precedente, solo dopo aver svolto tutti i calcoli l'interprete Python rileva e segnala l'errore causato dalla richiesta di stampare il valore di una variabile indefinita (S invece di s), senza poter stampare il risultato dell'elaborazione.

### 1.2.4. Paradigmi di programmazione

Un ulteriore modo di classificare i linguaggi di programmazione considera i *paradigmi* di programmazione. Questi si distinguono per l'enfasi che pongono sui due aspetti fondamentali: oggetti e operazioni e per il modo con cui operano sui dati.

I paradigmi di programmazione principali sono:

- **imperativo**: enfasi sulle operazioni intese come azioni, comandi, istruzioni che cambiano lo stato dell'elaborazione; gli oggetti sono funzionali alla elaborazione

- **funzionale**: enfasi sulle operazioni intese come funzioni che calcolano risultati; gli oggetti sono funzionali alla elaborazione

- **orientato agli oggetti**: enfasi sugli oggetti che complessivamente rappresentano il dominio di interesse; le operazioni sono funzionali alla rappresentazione

In genere in un programma sono utilizzati più paradigmi di programmazione. Quindi i linguaggi di programmazione forniscono supporto (in misura diversa) per i vari paradigmi.

## 1.3. Il linguaggio C

Queste dispense trattano il linguaggio di programmazione **C**. C è un linguaggio di programmazione di *alto livello*, compilato, che supporta i paradigmi di programmazione *imperativo* e *funzionale*. La sua estensione C++ supporta anche il paradigma di programmazione *orientata agli oggetti*.

Caratteristiche generali di C:

- estremamente efficiente;

- consente un accesso diretto alla memoria;

- molto usato in ambito scientifico e per la realizzazione di sistemi complessi;
- dispone di librerie di programma ricche e ben sviluppate.

### 1.3.1. Tecniche di programmazione in C

Scopo del corso è quello di approfondire i principali aspetti della programmazione imperativa e funzionale. Il linguaggio C è lo strumento utilizzato nella trattazione.

In particolare, attraverso il C vengono presentati i concetti di base dei linguaggi di programmazione, con particolare riferimento al modello di esecuzione e all'utilizzo della memoria.

La presentazione del C viene fatta in relazione alle conoscenze già acquisite in Python.

#### 1.3.1.1. Esempio di programma in C

**primo.c**

```c
/* definizione di funzioni di I/O (printf)*/
#include <stdio.h>

int main() {
  printf("Il mio primo programma C\n");
  printf("... e non sara' l'ultimo.\n");
  return 0;
}
```

Il programma è costituito da una serie di istruzioni o direttive per il compilatore.

Le istruzioni hanno il seguente significato:

```c
#include <stdio.h>
```

direttiva per includere la definizione di funzioni e variabili predefinite nel linguaggio relative all'input/output.

```c
int main() {
   ...
}
```

definizione della *funzione* main che racchiude il programma principale.

```
    printf("Il mio primo programma C ");
    printf("... e non sara' l'ultimo.\n");
```

istruzioni di stampa su video della frase:

```
Il mio primo programma C
... e non sara' l'ultimo.
```

printf è la funzione di stampa.
\n è il carattere di ritorno a capo nello schermo.

La sequenza di due istruzioni comporta l'esecuzione delle due istruzioni nell'ordine in cui sono scritte.

```
return 0;
```

è l'istruzione che termina l'esecuzione della funzione main e restituisce il risultato 0.

#### 1.3.1.2. Formato del programma

- Le parole in un programma C sono separate da spazi. Es. `int main`
- C (come Python) è **case-sensitive**, cioè distingue tra caratteri minuscoli e caratteri maiuscoli.
- Si possono lasciare un numero di spazi (e/o di linee vuote) a piacere. Andare a capo equivale a separare due elementi del programma (come uno spazio).
- In C, a differenza di Python, l'**indentazione** non ha alcun effetto sull'esecuzione del programma, essa è comunque molto importante perché rende i programmi più leggibili.
- In C tutte le istruzioni terminano con il carattere ;
- Le sequenze di escape comuni sono le stesse in C e Python (\b: backspace; \n: newline; \t: tab orizzontale; \\: backslash; \': apice singolo; \": apice doppio).

- Una differenza fondamentale tra i due linguaggi consiste nel fatto che il C non include le stringhe tra i tipi primitivi.

Il programma `primo2.c` è equivalente al programma `primo.c`.

**primo2.c**

```
#include <stdio.h>
int main() { printf("Il mio primo programma C\n");
      printf
("... e non sara' l'ultimo.\n"
      );
    return
      0;
                }
```

#### 1.3.1.3. Commenti

È possibile annotare il testo del programma con dei **commenti**. C dispone di due tipi di commento:

- `/* ... */` delimita un commento che può occupare più righe.
- `//` denota l'inizio di un commento che si estende solo fino alla fine della riga (non sempre supportato)

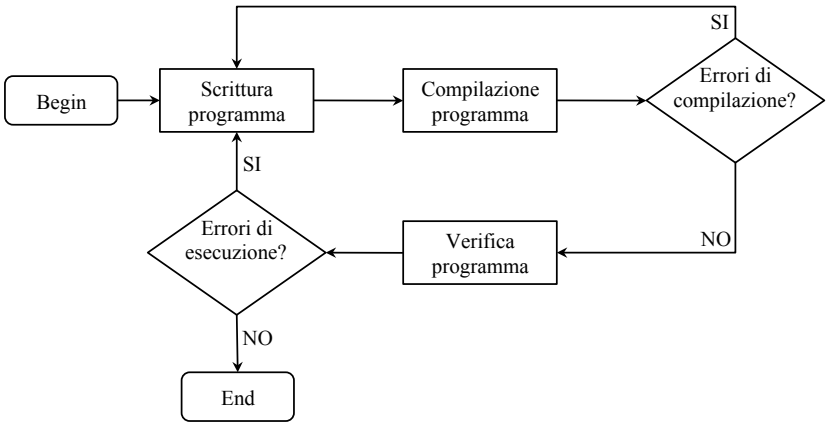
*I commenti non hanno alcun effetto sull'esecuzione del programma, sono usati per rendere il programma più leggibile.*

### 1.3.2. Scrivere, compilare ed eseguire un programma C

Le fasi principali dello sviluppo di un programma sono:

1. Preparazione del testo del programma
2. Compilazione del programma
3. Esecuzione del programma compilato

#### 1.3.2.1. Il processo di compilazione




Per un programma C il nome del file deve avere la seguente struttura:

```
nome.c
```

dove:

- *nome* è il nome del file contenente il testo del programma
- `c` è l'estensione che indica che il file contiene un programma C.

*Esempio:* `primo.c`

**1. Scrittura del programma** La scrittura di un programma può essere effettuata con qualsiasi programma che consenta la scrittura di un testo (*editor*). Ad esempio: notepad, emacs, gedit, ...

**2. Compilazione del programma** La compilazione del programma serve a tradurre il programma in una sequenza di comandi direttamente eseguibili dal calcolatore. Il compilatore C più usato è `gcc`, disponibile per diversi sistemi operativi. Per compilare programmi C si può usare anche un compilatore C++, ad esempio `g++`.

In questo corso useremo il compilatore `g++`.

Per compilare un programma C occorre eseguire il seguente comando:

```
g++ -o NomeEseguibile NomeFile.c
```

*Esempio:* Per compilare il programma `primo.c`, si può usare il comando:

```
g++ -o primo primo.c
```

La compilazione produce come risultato un file chiamato *`NomeEseguibile`*`.exe` in Windows, oppure *`NomeEseguibile`* in Unix/Linux/Mac, che contiene i comandi direttamente eseguibili dal calcolatore.

Se il processo di compilazione è corretto, il compilatore non stampa alcuna informazione, altrimenti stampa dei messaggi di errore.

Talvolta vengono forniti dei *warning*, cioè messaggi che indicano situazioni che potrebbero potenzialmente comportare errori ma che non impediscono al programma di essere compilato.

**3. Esecuzione del programma compilato** L'esecuzione di un programma si può effettuare solo dopo la compilazione, cioè quando il file `NomeEseguibile.exe` (oppure `NomeEseguibile`) è stato generato.

In C (al contrario di Python) non è possibile eseguire programmi sintatticamente errati.

L'esecuzione del programma avviene inserendo il nome del file eseguibile nel prompt di sistema. Nei sistemi operativi basati su Unix (Linux, Mac, ecc.) è necessario indicare anche il path del file eseguibile, ad esempio usando la sequenza `./` prima del nome del file eseguibile.

Esecuzione in ambiente Windows:

```
NomeEseguibile
```

Esecuzione in ambiente Unix:

```
./NomeEseguibile
```

*Esempio:* per eseguire il programma `primo.c` si può usare il comando

```
./primo
```

Il risultato dell'esecuzione del programma è quindi la seguente stampa su schermo:

```
Il mio primo programma C
... e non sara' l'ultimo.
```

#### 1.3.2.2. Errori in un programma

Gli errori nei programmi si possono classificare in tre categorie:

**Sintassi:** riguardano il mancato rispetto delle regole di scrittura del programma e sono individuati dal compilatore.

**Semantica:** sono dovuti all'impossibilità di assegnare un significato ad un'istruzione (es. variabile non dichiarata). Questi errori a volte sono individuati dal compilatore (errori di semantica statica), altre volte sono individuati a tempo di esecuzione (errori di semantica dinamica).

**Logica:** sono relativi alle funzionalità realizzate dal programma (differenti da quelle desiderate). Questi errori possono essere individuati solo analizzando il programma o eseguendo test di verifica del programma.

#### 1.3.2.3. Esempio: Soluzione equazioni di secondo grado

**equasecondo_err.c**

```c
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {

  printF("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  root = sqrt(b*b - 4.0 * a * c)
  root1 = 1/2 * (root + b) / a;
  root2 = -1/2 * (root + b) / a;
  printf("radici di %f x^2 + %f x + %f\n", a, b, c);
  printf("%f e %f\n", root1, root2);
  return 0;
}
```

Il programma precedente contiene diversi tipi di errore.

- printF è scritto in maniera errata (F maiuscola)
- l'istruzione root = sqrt (b*b - 4.0 * a * c) non termina con ;
- tutte le variabili non sono state dichiarate
- 1/2 calcola la divisione tra interi e quindi restituisce 0
- il calcolo di root1 è errato (non calcola correttamente la soluzione)

**equasecondo.c**

```
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {
  double a,b,c,root,root1,root2;
  printf("Inserisci coefficienti a b c: \n");
  scanf("%lf", &a);
  scanf("%lf", &b);
  scanf("%lf", &c);
  root = sqrt(b*b - 4.0 * a * c);
  root1 = 0.5 * (root - b) / a;
  root2 = -0.5 * (root + b) / a;
  printf("radici di %f x^2 + %f x + %f\n", a, b, c);
  printf("%f e %f\n", root1, root2);
  return 0;
}
```

## 1.4. Variabili

Al contrario di Python, in C tutti gli identificatori, in particolare le variabili, devono essere dichiarati prima dell'uso!

La dichiarazione di una variabile richiede di specificare il *tipo* della variabile (ad esempio, `int, double, char,...`) e il nome. Il tipo determina la quantità di memoria allocata alla variabile.

In Python la dichiarazione di variabili non è necessaria in quanto tutte le variabili sono di tipo riferimento ad oggetti.

### 1.4.1. Dichiarazione di variabili

Le variabili vengono introdotte nel programma attraverso **dichiarazioni di variabile**.

**Dichiarazione di variabile**

**Sintassi:**

```
tipo nomeVariabile;
```

- *tipo* è il *tipo* della variabile
- *nomeVariabile* è il nome della variabile da dichiarare

**Semantica:**
La dichiarazione di una variabile riserva spazio in memoria per la variabile e rende la variabile disponibile nella parte del programma (blocco) dove appare la dichiarazione. È indispensabile dichiarare una variabile prima di usarla.

*Esempio:*

```
int x;
```

- `int` è il tipo della variabile
- `x` è il nome della variabile

int è il tipo usato per i numeri interi. I tipi di dato primitivi sono illustrati nell'Unità 2.

Dopo questa dichiarazione la variabile x è disponibile per l'utilizzo nel blocco del programma dove appare la dichiarazione (ad esempio, all'interno della funzione main, se la dichiarazione appare lì).

**Dichiarazione multipla** Si possono anche dichiarare più variabili dello stesso tipo con una sola dichiarazione.

```
tipo nomeVar_1, ..., nomeVar_n;
```

Questa dichiarazione è equivalente a:

```
tipo nomeVar_1;
...
tipo nomeVar_n;
```

### 1.4.2. Notazione grafica per la rappresentazione di variabili

Una variabile è un riferimento ad un'area di memoria in cui è memorizzato un valore. La dimensione dell'area di memoria dipende dal tipo della variabile ed è assegnata al momento della compilazione.

Per rappresentare le variabili e i loro valori noi useremo la seguente notazione grafica


Il diagramma rappresenta il nome, il tipo, l'indirizzo di memoria e il valore di una variabile. Spesso ometteremo il tipo della variabile e l'indirizzo di memoria. Nel diagramma non viene rappresentata la dimensione dell'area di memoria allocata.

In Python, tutte le variabili sono associate ad aree di memoria della stessa dimensione e contengono riferimenti ad oggetti (che invece sono allocati in aree di memoria di dimensione variabile a seconda del tipo dell'oggetto).

### 1.4.3. Assegnazione

L'istruzione di assegnazione serve per memorizzare un valore in una variabile.

**Assegnazione**

**Sintassi:**

```
nomeVariabile = espressione;
```

- *nomeVariabile* è il nome di una variabile
- *espressione* è una espressione che, valutata, deve restituire un valore del tipo della variabile

**Semantica:**
Alla variabile *nomeVariabile* viene assegnato il valore dell'*espressione* che si trova a destra del simbolo =. Tale valore deve essere *compatibile* con il tipo della variabile (vedi dopo). Dopo l'assegnazione il valore di una variabile rimane invariato fino alla successiva assegnazione.

In C il risultato dell'espressione a destra dell'operatore = viene memorizzato nell'area di memoria associata alla variabile.

In Python l'assegnazione opera sui riferimenti agli oggetti: il riferimento dell'oggetto risultato dell'espressione a destra dell'operatore = viene associato alla variabile.

*Esempio:*

```
int x;
x = 3 + 2;
```

- x è una variabile di tipo int
- 3 + 2 è un'espressione intera

Il risultato dell'esecuzione della assegnazione è che dopo di essa x contiene il valore 5.

*Esempio:*

```
int x;
x = 3.0 + 2.0;
```

Il frammento di codice produce un errore a tempo di compilazione, in quanto il risultato dell'espressione è di tipo reale, mentre la viariabile è di tipo intero. Ulteriori dettagli sulla compatibilità dei tipi sono illustrati nell'Unità 2.

### 1.4.4. Inizializzazione delle variabili

***Inizializzare*** *una variabile significa specificare un valore da assegnare inizialmente (prima di qualsiasi utilizzo) ad essa.*

Si noti che una variabile non inizializzata non contiene ancora un valore definito e quindi utilizzarla prima di un'assegnazione può comportare errori nel programma. Se la variabile è stata dichiarata ma non ancora inizializzata, allora si può usare l'assegnazione per farlo.

In Python l'uso di una variabile non inizializzata provoca un errore a tempo di esecuzione. In C l'uso di una variabile dichiarata, ma non inizializzata, non genera errori di compilazione o di esecuzione, ma può comportare errori logici.

*Esempio:* Il seguente programma contiene un errore logico: x e y non sono state inizializzate prima dell'espressione. Il risultato risulta non

definito (dipende effettivamente dallo stato della memoria al momento in cui viene eseguito il programma).

```
int main() {
    int x, y, z;
    // ERRORE le variabili x e y sono indefinite
    z = x * y;
    ...
}
```

Le variabili possono essere inizializzate anche al momento della loro dichiarazione.

```
int main() {
    int x=1, y, z;
    y = 2;
    // OK le variabili x e y sono inizializzate
    z = x * y;
    ...
}
```

## 1.5. Funzioni

Le *funzioni* sono moduli di programma che svolgono un particolare compito.

Come in Python le funzioni possono essere predefinite o scritte dall'utente.

Vediamo innanzitutto come si invoca (chiama) una funzione.

**Invocazione di funzione**

*Sintassi:*

```
nomeFunzione(parametri)
```

- *nomeFunzione*(...) è la funzione invocata
- *parametri* sono i parametri passati alla funzione

*Semantica:*

Invoca una funzione fornendole eventuali parametri addizionali. L'invocazione di una funzione comporta l'esecuzione

dell'operazione associata ed, in genere, la restituzione di un valore.

*Esempio:*

```
sqrt(169)
```

- `sqrt` è la funzione che calcola la radice quadrata di un numero
- `169` è il parametro (o argomento) passato alla funzione
- la funzione restituisce un risultato (`13`) che viene usato dall'istruzione che ha chiamato la funzione o dall'espressione in cui la funzione è inserita.

### 1.5.1. Intestazione di una funzione

L'**intestazione** (o prototipo) di una funzione consiste nella segnatura (nome e parametri) e nel *tipo* del risultato.

*Esempi*:

```
double sqrt(double x)
```

```
double pow(double b, double e)
```

*Nota:* Al contrario di Python, in C bisogna definire esplicitamente il tipo del valore restituito da una funzione.

### 1.5.2. Parametri di una funzione

I parametri di una funzione sono i valori passati dal modulo chiamante alla funzione per poter svolgere i calcoli.

*Esempio:* la funzione `sqrt(double x)` deve essere invocata passandole un parametro che rappresenta il valore di cui vogliamo calcolare la radice quadrata.

In generale, i parametri passati come argomenti possono essere espressioni complesse formate a loro volta da invocazioni di altre funzioni.

*Esempio:*

```
sqrt(pow(5,2)+pow(12,2))
```

calcola la radice quadrata di $5^2 + 12^2$.

### 1.5.3. Risultato di una funzione

Il risultato calcolato da una funzione viene restituito al blocco di codice che ha chiamato la funzione stessa.

*Esempio:*

l'istruzione

```
printf ("%f", sqrt(pow(5,2)+pow(12,2)));
```

stampa il valore 13 (cioè il risultato di $\sqrt{5^2 + 12^2}$).

Infatti, la funzione `pow(5,2)` restituisce il valore 25, la funzione `pow(12,2)` restituisce 144, l'operatore `+` calcola la somma dei due valori e restituisce 169, la funzione `sqrt` restituisce il valore 13, e infine il valore 13 viene usato dalla funzione `printf` per stampare tale valore sulla console di output.

### 1.5.4. Funzioni definite in math.h

Le principali funzioni e costanti matematiche sono definite nel file di sistema `math.h` , che bisogna quindi includere mediante la direttiva `#include <...>`.

Per conoscere le funzioni definite in `math.h` (ed in generale le funzioni matematiche messe a disposizione dal C) si veda:

`http://www.gnu.org/software/libc/manual/html_mono/libc.html#Mathematics`

| Nome funzione | Significato |
|---|---|
| `cos(x)` | coseno |
| `sin(x)` | seno |
| `tan(x)` | tangente |
| `acos(x)` | arcocoseno |
| `asin(x)` | arcoseno |
| `atan(x)` | arcotangente |
| `atan2(x,y)` | arcotangente con due parametri ($\tan^{-1}(y/x)$) |

| Nome funzione | Significato |
|---|---|
| `pow(x,y)` | potenza ($x^y$) |
| `sqrt(x)` | radice quadrata |
| `exp(x)` | esponenziale ($e^x$) |
| `log(x)` | logaritmo naturale |
| `log10(x)` | logaritmo in base 10 |

| Nome funzione | Significato |
|---|---|
| `ceil(x)` | arrotondamento in eccesso |
| `fabs(x)` | valore assoluto |
| `floor(x)` | arrotondamento in difetto |
| `round(x)` | arrotondamento all'intero più vicino |

### 1.5.5. La funzione exit

La funzione `exit(int status)` fa terminare il programma, restituendo il valore passato come argomento al sistema operativo. Il valore 0, restituito con `exit(0)`, indica per convenzione che il programma è terminato regolarmente. Valori non nulli (nell'intervallo $[1-255]$) vengono associati a condizioni d'uscita anomale. Tipicamente, il valore 1 viene usato per indicare fallimento generico. Il risultato dell'esecuzione permette al sistema di rispondere in maniera specifica ai diversi tipi di errore.

### 1.5.6. Funzioni di Input/Output

In C, la principale funzione di output è `printf`, cui vanno specificati il formato caratterizzato dal simbolo `%` ed i parametri, cioè le stringhe o le espressioni da stampare.

*Esempio:* L'istruzione

```
printf("Primo = %d, secondo = %f", 5, 5.0);
```

esegue la stampa a video della stringa

```
Primo = 5, secondo = 5.000000
```

sostituendo alla stringa `%d`, il valore del secondo argomento (5) della chiamata ed alla stringa `%f` il valore del terzo argomento (5.0). La stringa

%d indica che il valore (5) da stampare è un intero. La stringa %f indica che il valore (5.0) da stampare è un reale. Non c'è limite al numero di argomenti che si possono passare alla funzione.

La funzione scanf permette di leggere input da tastiera, specificando il formato dei dati letti, caratterizzato dal simbolo %, ed i parametri, cioè le variabili da leggere, con la notazione &.

*Esempio:* Il seguente frammento di codice legge da tastiera un valore reale e li memorizza in una variabile di tipo double n (secondo l'ordine di inserimento).

```
double n; // Dichiara una variabile reale
scanf("%lf", &n);
```

La stringa %lf indica che il carattere da leggere è di tipo double.

Non ci sono limiti al numero di valori che si possono leggere in input, e quindi al numero di parametri che si possono passare alla funzione ma è consigliato, almeno in una fase iniziale di studio, di leggere un valore per volta.

Ulteriori dettagli sulle funzioni printf e scanf saranno visti in seguito.

## 1.6. Blocco di istruzioni

Un **blocco di istruzioni** raggruppa più istruzioni in un'unica istruzione composta.

**Blocco di istruzioni**

**Sintassi:**

```
{
  istruzione
  ...
  istruzione
}
```

- *istruzione* è una qualsiasi istruzione C

**Semantica:**
Le istruzioni del blocco vengono eseguite in sequenza. Le variabili dichiarate internamente al blocco non sono visibili all'esterno

del blocco.

*Esempio:*

```
int a, b;
...
{
  printf("dato1 = %d\n",a);
  printf("dato2 = %d\n",b);
}
```

### 1.6.1. Campo d'azione delle variabili

Un blocco di istruzioni può contenere al suo interno dichiarazioni di variabili. Una variabile dichiarata dentro un blocco ha come **campo di azione** il blocco stesso, inclusi eventuali blocchi interni. Questo significa che *la variabile è visibile nel blocco e in tutti i suoi blocchi interni, ma non è visibile all'esterno del blocco*.

In Python, una variabile può essere usata dopo la sua inizializzazione indipendentemente dal blocco di codice in cui viene inizializzata. Quindi, una variabile inizializzata in un blocco di codice interno può essere usata in blocchi di codice esterni successivi.

*Esempio:*

**campoDazione.c**

```
// campo d'azione
#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {
  double a = 2.3;      // a double
  int i = 1;           // i int
  printf("liv. 0  a = %f\n", a); // 2.3
  printf("liv. 0  i = %d\n", i); // 1
  {
    printf("liv. 1  a = %f\n", a); // 2.3
    printf("liv. 1  i = %d\n", i); // 1

    double i = 0.1;    //  i double
    {
      double r = 5.5; // r double
      i = i + 1;       // i double
      printf("liv. 2.1 r = %f\n", r); // 5.5
      printf("liv. 2.1 i = %f\n", i); // 1.1
```

```
    }
    // printf("liv. 1  r = %f\n",r);  // ERRORE
    printf("liv. 1  i = %f\n",i);     // 1.1
    {
      int r = 4;       // r int
      printf("liv. 2.2 r = %d\n", r); // 4
      printf("liv. 2.2 a = %f\n", a); // 2.3
    }
  }
  i = i + 1;           // i int
  printf("liv. 0  i = %d\n", i); // 2
  return 0;
}
```

## 1.7. Struttura di un programma

Non ci sono regole stringenti per organizzare un programma, ma la necessità di *definire prima di usare* suggerisce di usare il seguente schema generale:

```
direttive del compilatore
dichiarazione variabili globali (esterne)
dichiarazione intestazione di funzioni
dichiarazione main
dichiarazione funzioni
```

### 1.7.1. Variabili globali

In C si possono introdurre delle dichiarazioni di variabili prima della dichiarazione della funzione main.

Queste dichiarazioni vengono considerate *globali* (talvolta denominate *esterne*, ma noi utilizzeremo questo termine con un significato specifico che verrà illustrato più avanti).

Una dichiarazione globale ha come campo d'azione il file nel quale è definita.

*Esempio:*

**equasecondo_varglobale.c**

```
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
```

```c
#include <math.h>   /* funzioni matematiche */

float unmezzo = 0.5;

int main () {
  double a,b,c,root,root1,root2;
  printf("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  root = sqrt(b*b - 4.0 * a * c);
  root1 = unmezzo * (root - b) / a;
  root2 = - unmezzo * (root + b) / a;
  printf("radici di %f x^2 + %f x + %f\n", a, b, c);
  printf("%f e %f\n", root1, root2 );
  return 0;
}
```

L'uso di variabili globali diventa significativo quando si introducono nel programma diverse definizioni di funzione che possono quindi tutte fare riferimento alle variabili globali del file in cui sono definite.

Nel seguito vedremo come organizzare le definizioni del programma su più file. Questa funzionalità richiederà una rivisitazione del concetto di variabile globale.

### 1.7.2. Decomposizione in moduli

In C (come in Python) il meccanismo per modularizzare il programma è costituito dalla suddivisione in funzioni (non ci sono script, dato che il linguaggio è compilato).

La definizione di funzioni sarà trattata più avanti.

## 1.8. Istruzioni condizionali

In C le istruzioni condizionali if-else ed if sono simili a quelle di Python, ma ci sono alcune differenze da sottolineare:

- la sintassi è leggermente diversa (parentesi per la condizione e assenza del carattere :);
- quando ci sono più istruzioni in un ramo occorre racchiuderle in un blocco di istruzioni delimitate da { ... } (non serve l'indentazione! Ma è desiderabile);
- le condizioni in C sono espressioni il cui valore 0 corrisponde a false e qualunque altro valore corrisponde a true.

### 1.8.1. Soluzione equazioni secondo grado: rivista

**equasecondo-if.c**

```c
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {
  double a, b, c;
  printf("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  double delta = b*b - 4.0 * a * c;
  if (delta > 0.0) {
    double root = sqrt(delta);
    double root1 = 0.5 * (root - b) / a;
    double root2 = - 0.5 * (root + b) / a;
    printf("radici reali: %lf e %lf\n", root1, root2);
  } else if (delta < 0.0) {
    double root = sqrt(-delta);
    double real_part = - 0.5 * b / a;
    double imag_part = 0.5 * root / a;
    printf("radici complesse: %lf + i * %lf e %lf - i
    * %lf\n",
     real_part, imag_part, real_part, imag_part) ;
  } else {
    double root1 = -0.5*b/a;
    printf("radici coincidenti: %lf\n", root1);
  }
  return 0;
}
```

### 1.8.2. L'istruzione if-else

L'istruzione if-else consente di effettuare una *selezione* a 2 vie.

**Istruzione if-else**

*Sintassi:*

```
if (condizione)
  istruzione-then
else
  istruzione-else
```

- *condizione* è un'espressione booleana, valutata true oppure false

- *istruzione-then* è una singola istruzione (detta anche il *ramo-then* dell'istruzione if-else)
- *istruzione-else* è una singola istruzione (detta anche il *ramo-else* dell'istruzione if-else)

*Semantica:*
Viene valutata prima la *condizione*. Se la valutazione fornisce un valore diverso da 0 (true), viene eseguita *istruzione-then*, altrimenti viene eseguita *istruzione-else*. In entrambi i casi l'esecuzione prosegue con l'istruzione che segue l'istruzione if-else.

*Esempio:*

```
int a, b;
...
if (a > b)
  printf("maggiore = %d\n", a);
else
  printf("maggiore = %d\n", b);
```

L'esecuzione di questa istruzione if-else produce la stampa a video della stringa "maggiore = ", seguita dal valore del maggiore tra a e b.

### 1.8.3. La variante if

La parte else di un'istruzione if-else è opzionale.

Se manca si parla di istruzione if, la quale consente di eseguire una parte di codice solo se è verificata una condizione.

**Istruzione if**

*Sintassi:*

```
if (condizione)
  istruzione-then
```

- *condizione* è un'espressione booleana
- *istruzione-then* è una singola istruzione (detta anche il *ramo-then* dell'istruzione if)

*Semantica:*
Viene valutata prima la *condizione*. Se la valutazione fornisce un valore diverso da 0 (true), viene eseguita *istruzione-then* e si procede con l'istruzione che segue l'istruzione if. Altrimenti si procede direttamente con l'istruzione che segue l'istruzione if.

*Esempio:*

```
double a = 1.0;
if (a>0) printf("a positivo.\n");
```

L'esecuzione di questa istruzione if produce la stampa sul canale di output della stringa "a positivo" quando la condizione (a>0) è vera, altrimenti non stampa nulla.

### 1.8.4. Condizione nell'istruzione if-else

La condizione in un'istruzione if-else può essere costruita usando:
1) operatore di confronto (==, !=, >, <, >=, <=) applicato a variabili (o espressioni) di un tipo primitivo;
*Esempio:*

```
int a, b, c;
...
if (a > b + c)
   ...
```

2) chiamata ad una *funzione;*
*Esempio:*

```
int a,b;
...
if (verifica(a,b))
   ...
```

verifica(a,b) è una funzione che verifica una condizione sui valori a e b.

3) espressione booleana *complessa*, ottenuta applicando gli operatori booleani !, && e || a espressioni più semplici;

*Esempio:*

```
int a, b, c, d;
int trovato;
...
if ((a>(b+c)) || (a == d) && ! trovato)
   ...
```

### 1.8.5. Attenzione alle condizioni

Ci sono delle formulazioni delle condizioni che possono diventare facilmente sorgenti di errori:

```
int a;
double b;

if (a=0)     // assegnazione
   ...
if (b==0)    // approssimazione
   ...
if (a==b)    // conversione di tipo
   ...
```

Nel primo caso if (a=0) l'errore riguarda l'uso dell'operatore di assegnazione invece dell'operatore di confronto. In questo caso il compilatore non segnala alcuna anomalia, dato che il risultato dell'espressione (a=0) viene convertito nel valore booleano corrispondente cioè false, e sarà sempre eseguito il ramo else dell'istruzione condizionale. Una possibile contromisura potrebbe essere quella di scrivere sempre prima il valore costante (0=a). In questo modo, il compilatore segnala un errore dovuto ad un'assegnazione scorretta (non può esserci un valore nella parte destra dell'operatore).

Nel secondo caso if (b==0), occorre accertarsi che errori di approssimazione non portino ad avere sistematicamente il valore false.

Nel terzo caso bisogna notare che il simbolo == in C denota l'uguaglianza di valori contenuti nella memoria, mentre in Python indica l'uguaglianza *strutturale*.

### 1.8.6. Valutazione di una condizione complessa

La condizione di un'istruzione if-else può essere un'espressione booleana complessa, nella quale compaiono gli operatori logici &&, ||, e !.

Si deve tenere presente che le sottoespressioni relative a tali operatori vengono valutate da sinistra a destra al seguente modo:

- nel valutare $(e_1$ && $e_2)$, se la valutazione di $e_1$ restituisce false, allora $e_2$ **non viene valutata**.

- nel valutare $(e_1 \,||\, e_2)$, se la valutazione di $e_1$ restituisce true, allora $e_2$ **non viene valutata**.

Infatti, se $e_1$ è falso, $(e_1$ && $e_2)$ risulta falso, indipendentemente dal valore di $e_2$. Analogamente se $e_1$ è vero, $(e_1$ || $e_2)$ risulta vero, indipendentemente dal valore di $e_2$.

*In generale, bisogna sempre tenere conto del fatto che in questi casi* $e_2$ *non viene valutata.*

*Esempio:*

```
int i;
...
if (i > 0 && fact(i) > 100) {
  ...
}
```

Si noti che la funzione fact(i) non viene invocata nel caso in cui i abbia valore minore o uguale a 0.

Istruzioni if-else che fanno uso di espressioni booleane complesse potrebbero essere riscritte attraverso l'uso di if-else annidati. In generale questo comporta però la necessità di duplicare codice.

#### 1.8.6.1. Uso dell'operatore di congiunzione &&

```
if ((x < y) && (y < z))
  printf("y compreso tra x e z");
else
  printf("y non compreso tra x e z");
```

corrisponde a

```
if (x < y)
  if (y < z)
     printf("y compreso tra x e z");
  else
    printf("y non compreso tra x e z");
else
  printf("y non compreso tra x e z");
```

Si noti come in questo caso, eliminando la condizione composta, il codice del ramo-else debba essere duplicato.

#### 1.8.6.2. Uso dell'operatore di disgiunzione ||

```
if ((x == 1) || (x == 2))
  printf("x uguale a 1 o a 2");
else
  printf("x diverso da 1 e da 2");
```

corrisponde a

```
if (x == 1)
  printf("x uguale a 1 o a 2");
else if (x == 2)
  printf("x uguale a 1 o a 2");
else
  printf("x diverso da 1 e da 2");
```

Si noti come in questo caso, eliminando la condizione composta il codice del ramo-then debba essere duplicato.

### 1.8.7. Uso del blocco di istruzioni nell'if-else

Il ramo-then e il ramo-else di un'istruzione if-else possono essere una qualsiasi istruzione C, in particolare un blocco di istruzioni.

```
if (a>b) {
  printf("maggiore = %d\n",a);
  printf("minore = %d\n",b);
}
```

*Esempio:*

Dati mese ed anno, calcolare mese ed anno del mese successivo.

```
int mese, anno, mesesucc, annosucc;
...
if (mese == 12) {
  mesesucc = 1;
  annosucc = anno + 1;
}
else {
  mesesucc = mese + 1;
  annosucc = anno;
}
```

### 1.8.8. If annidati

Si hanno quando l'istruzione del ramo-then o del ramo-else è un'istruzione if-else o if.

*Esempio:*

```
int giorno,mese,anno, giornosucc,mesesucc,annosucc;
...
if (mese == 12) {
  if (giorno == 31) {
    giornosucc = 1;
    mesesucc = 1;
    annosucc = anno + 1;
  }
  else {
    giornosucc = giorno + 1;
    mesesucc = mese;
    annosucc = anno;
  }
}
else {
  ...
}
```

#### 1.8.8.1. If annidati, condizioni mutuamente esclusive

Un caso comune di utilizzo degli if annidati è quello in cui le condizioni degli if annidati si escludono mutuamente.

*Esempio:*

In base al valore della temperatura (intero) stampare un messaggio secondo la seguente tabella:

| temperatura $t$ | messaggio |
|---|---|
| $30 < t$ | molto caldo |
| $20 < t \leq 30$ | caldo |
| $10 < t \leq 20$ | gradevole |
| $t \leq 10$ | freddo |

```
int temp;
...
if (30 < temp)
  printf("molto caldo");
else if (20 < temp)
  printf("caldo");
else if (10 < temp)
  printf("gradevole");
else
  printf("freddo");
```

*Osservazioni:*

- al livello più esterno abbiamo un'unica istruzione if-else
- l'ordine in cui vengono specificate le condizioni è importante
- non serve che la seconda condizione sia composta, ad es. (20 < temp) && (temp <= 30)
- ogni else si riferisce all'if immediatamente precedente

### 1.8.9. Ambiguità if-else

Consideriamo il seguente frammento di codice:

```
if (a > 0) if (b > 0) printf("b positivo"); else
    printf("???");
```

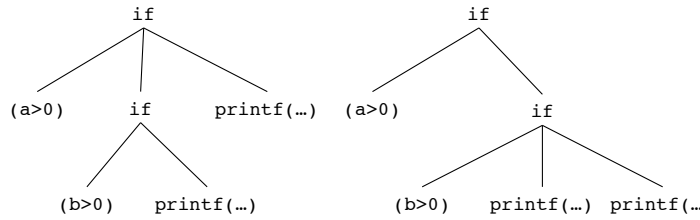
printf("???"); potrebbe essere la parte else

- del primo if: quindi printf("a negativo");
- del secondo if: quindi printf("b negativo");

#### 1.8.9.1. Grammatica delle istruzioni condizionali

```
istruzione = istruzione-semplice | istruzione-composta
istruzione-composta = istruzione-condizionale | ...
istruzione-condizionale = istruzione-if | istruzione-switch
istruzione-if = "if" ( condizione ) istruzione "else" istruzione
              | "if" ( condizione ) istruzione
```

Richiamo: Una grammatica si dice ambigua quando si possono associare 2 alberi sintattici diversi ad una stessa frase riconosciuta dal linguaggio




L'ambiguità si risolve considerando che **un else fa sempre riferimento all'if più vicino**:

```
if (a > 0)
  if (b > 0)
    printf("b positivo");
  else
    printf("b negativo");
```

*È sempre possibile usare il blocco di istruzioni { ... } per disambiguare istruzioni if-else annidate.* In particolare, perché un else si riferisca ad un if che non è quello immediatamente precedente, quest'ultimo deve essere racchiuso in un blocco:

```
if (a > 0) {
  if (b > 0)
    printf("b positivo");
}
else
    printf("a negativo");
```

### 1.8.10. Espressione condizionale

C mette a disposizione un operatore di selezione che permette di costruire un'**espressione condizionale**. L'uso di un'espressione condizionale può in alcuni casi semplificare il codice rispetto all'uso di un'istruzione if-else.

**Espressione condizionale**

*Sintassi:*

```
condizione?espressione-1:espressione-2
```

- *condizione* è un'espressione booleana
- *espressione-1* e *espressione-2* sono due espressioni qualsiasi, che devono essere dello stesso tipo

*Semantica:*
Valuta *condizione*. Se il risultato è true, allora valuta *espressione-1* e ne restituisce il valore, altrimenti valuta *espressione-2* e ne restituisce il valore.

*Esempio:*

```
printf("maggiore = %\n", (a > b)? a : b);
```

L'istruzione nell'esempio, che fa uso di un'espressione condizionale, è equivalente a:

```
if (a > b)
  printf("maggiore = %d", a);
else
  printf("maggiore = %d", b);
```

Si noti che l'operatore di selezione è simile all'istruzione if-else, ma agisce ad un livello sintattico diverso:

- l'operatore di selezione combina *espressioni* e restituisce un'altra espressione; quindi, può essere usato ovunque può essere usata un'espressione;

- l'istruzione if-else raggruppa *istruzioni*, ottenendo un'istruzione composta.

### 1.8.11. L'istruzione switch

Se dobbiamo realizzare una **selezione a più vie**, possiamo usare diversi if-else annidati. C mette però a disposizione un'istruzione specifica che può essere usata in alcuni casi per realizzare in modo più semplice una selezione a più vie.

**Istruzione switch**

*Sintassi:*

```
switch (espressione) {
  case etichetta-1: istruzioni-1
     break;
  ...
  case etichetta-n: istruzioni-n
     break;
  default: istruzioni-default
}
```

- *espressione* è un'espressione intera o di tipo char
- *etichetta-1*, ..., *etichetta-n* sono espressioni intere (o carattere) costanti; ovvero, possono contenere solo letterali interi (o carattere) o costanti inizializzate con espressioni costanti; una espressione non può essere ripetuta come etichetta in più case
- *istruzioni-1*, ..., *istruzioni-n* e *istruzioni-default* sono sequenze di istruzioni qualsiasi
- la parte default è opzionale

*Semantica:*

1. viene prima valutata *espressione*
2. viene cercato il primo *i* per cui il valore di *espressione* è pari a *etichetta-i*

3. se si è trovato tale *i*, allora vengono eseguite *istruzioni-i* altrimenti vengono eseguite *istruzioni-default*
4. l'esecuzione procede con l'istruzione successiva all'istruzione switch

*Esempio:*

```
int i;
...
switch (i) {
  case 0: printf("zero"); break;
  case 1: printf("uno"); break;
  case 2: printf("due"); break;
  default: printf("minore di zero o maggiore di due");
}
```

Quando i è pari a 0 (rispettivamente 1, 2) viene stampato zero (rispettivamente uno, due), mentre quando i è minore di 0 oppure maggiore di due, viene stampato minore di zero o maggiore di due.

Se abbiamo più valori per cui eseguire le stesse istruzioni, si possono raggruppare i diversi case:

```
case v1:
case v2:
...
case vn:
  <istruzioni>
  break;
```

*Esempio:* Calcolo dei giorni di un mese.

```
int mese, giorniDelMese;
...
switch (mese) {
case 4: case 6: case 9: case 11:
  giorniDelMese = 30; break;
case 1: case 3: case 5: case 7: case 8: case 10: case
    12:
  giorniDelMese = 31; break;
case 2:
  giorniDelMese = 28; break;
default:
  giorniDelMese = 0;
  printf("Mese non valido");
}
printf("Giorni del mese %d: %d \n", mese,
    giorniDelMese);
```

#### 1.8.11.1. Osservazioni sull'istruzione switch

L'*espressione* usata per la selezione può essere una qualsiasi espressione C che restituisce un valore *intero* o *carattere* (ma non un valore reale).

I valori specificati nei vari *case* devono invece essere *espressioni costanti*, ovvero il loro valore deve essere noto a tempo di compilazione. In particolare, non possono essere espressioni che fanno riferimento a variabili. Il seguente frammento di codice è sbagliato:

```
int a;
...
switch (a) {
  case a<0: printf("negativo");
            // ERRORE: a<0 non e' una costante
  case 0:   printf("nullo");
  case a>0: printf("positivo");
            // ERRORE: a>0 non e' una costante
}
```

Ne segue che l'*utilità dell'istruzione switch è limitata*.

#### 1.8.11.2. Omissione del break

In realtà, non si richiede che nei case di un'istruzione switch l'ultima istruzione sia break.

In assenza di break si prosegue con l'esecuzione delle istruzioni successive a quella corrispondente all'etichetta valutata, finché non si arriva ad una istruzione break o al termine dell'istruzione switch.

```
int lati;  //  massimo numero di lati del poligono (al
     piu' 6)
...
printf("Poligoni con al piu' %d lati: ",lati);
switch (lati) {
  case 6: printf("esagono, ");
  case 5: printf("pentagono, ");
  case 4: printf("rettangolo, ");
  case 3: printf("triangolo");
          break;
  case 2: case 1: printf("nessuno");
                  break;
  default: printf("\n");
           printf("Immetti un valore <= 6.\n");
```

Se il valore di lati è pari a 5, allora il precedente codice stampa:

```
pentagono, rettangolo, triangolo
```

*Nota:* quando si omettono i break, *diventa rilevante l'ordine* in cui vengono scritti i vari case. Questo può essere spesso causa di errori. Quindi, *è buona norma mettere break come ultima istruzione di ogni case*.

## 1.9. Istruzioni di ciclo

In C il ciclo while è simile a quello di Python, ma ci sono alcune differenze da sottolineare:

- la sintassi è leggermente diversa (parentesi per la condizione e assenza del carattere :);
- quando ci sono più istruzioni nel corpo del ciclo occorre racchiuderle in un blocco { ... } (non serve l'indentazione!);
- per le condizioni vale quanto detto a proposito dei condizionali.

### 1.9.1. Ciclo while

L'istruzione while consente la ripetizione di una istruzione.

**Istruzione while**

*Sintassi:*

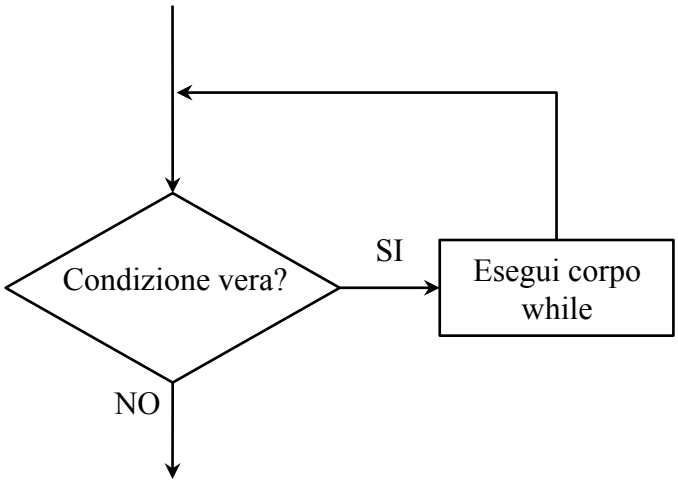
```
while (condizione)
  istruzione
```

- *condizione* è un'espressione booleana
- *istruzione* è una singola istruzione (detta anche il *corpo* del ciclo)

*Nota:* dato che è possibile, facendo uso di un blocco, raggruppare più istruzioni in una singola istruzione composta, è di fatto possibile avere più istruzioni nel corpo del ciclo.
*Semantica:*

- viene valutata prima la *condizione*
- se è true (valore diverso da 0), viene eseguita *istruzione* e si torna a valutare la *condizione*, procedendo così fino a quando *condizione* diventa falsa (valore uguale a 0)
- a questo punto si passa ad eseguire l'istruzione che segue il ciclo while

### 1.9.2. Semantica dell'istruzione di ciclo while




Quindi, il corpo del ciclo viene eseguito finché la *condizione* si mantiene vera. Non appena questa diventa falsa si esce dal ciclo e si continua l'esecuzione con l'istruzione successiva al while.

*Esempio:* Stampa di 100 asterischi.

```
int i = 0;
while (i < 100) {
    printf("*");
    i++;
}
```

### 1.9.3. Cicli definiti ed indefiniti

Concettualmente, si distinguono due tipi di ciclo, che si differenziano in base a come viene determinato il numero di *iterazioni* (ripetizioni del corpo del ciclo):

- Nei **cicli definiti** il numero di iterazioni è noto prima di iniziare l'esecuzione del ciclo.

  *Esempio:* per 10 volte ripeti la stampa di un *.

- Nei **cicli indefiniti** il numero di iterazioni non è noto prima di iniziare l'esecuzione del ciclo, ma è legato al verificarsi di una

condizione (questo dipende a sua volta dalle operazioni eseguite nel corpo del ciclo).

*Esempio:* finchè l'utente non sceglie di smettere, stampa un * e chiedi all'utente se smettere.

In C, come in altri linguaggi, entrambi i tipi di ciclo possono essere realizzati attraverso l'istruzione while.

### 1.9.4. Esempio di ciclo while: potenza

```
int base, esponente, potenza;
base = ...;
esponente = ...;
potenza = 1;
while (esponente > 0) {
    potenza = potenza * base;
    esponente--;
}
```

Si noti che si tratta di fatto di un *ciclo definito*, poiché il numero di iterazioni dipende solo dai valori delle variabili che sono stati fissati prima di iniziare l'esecuzione del ciclo.

### 1.9.5. Elementi caratteristici nella progettazione di un ciclo

```
inizializzazione
while (condizione) {
   operazione
   passo successivo
}
```

- *inizializzazione*: definizione del valore delle variabili utilizzate nel ciclo prima dell'inizio dell'esecuzione del ciclo (prima dell'istruzione di ciclo)
  *Es.* `potenza = 1;`
- *condizione*: espressione valutata all'inizio di ogni iterazione, il cui valore di verità determina l'esecuzione del corpo del ciclo o la fine del ciclo
  *Es.* `(esponente > 0)`

- *operazione del ciclo*: calcolo del risultato parziale ad ogni iterazione del ciclo (nel corpo del ciclo)
  *Es.* potenza = potenza * base;

- *passo successivo*: operazione di incremento/decremento della variabile che controlla le ripetizioni del ciclo (nel corpo del ciclo)
  *Es.* esponente–;

Una volta progettato il ciclo occorre verificarne la **terminazione**. In particolare, occorre verificare che l'esecuzione delle istruzioni del ciclo possa modificare il valore della condizione in modo da renderla falsa.

*Esempio:* l'istruzione esponente–; consente di rendere la condizione (esponente > 0) falsa, se esponente è un numero intero positivo.

### 1.9.6. Errori comuni nella scrittura di cicli while

*Dimenticarsi di inizializzare una variabile che viene utilizzata nella condizione del ciclo.* Si ricordi che la prima volta che la condizione viene verificata è prima di iniziare ad eseguire il corpo del ciclo.

*Esempio:*

```
int i;
while (i != 0) {
    printf("%d", i*i);
    printf("prossimo intero");
}
```

La variabile i non è inizializzata, quindi il risultato della condizione del ciclo while risulta indefinita.

*Dimenticarsi di aggiornare le variabili che compaiono nella condizione del ciclo.* Il ciclo non terminerà mai.

*Esempio:*

```
int i;
scanf("%d",&i);
while (i != 0) {
    printf("%d", i*i);
    printf("prossimo intero");
}
```

Manca l'istruzione di lettura del prossimo elemento da elaborare.

*Sbagliare di 1 il numero di iterazioni* Tipicamente è dovuto ad un errore nella condizione del ciclo o nell'inizializzazione di variabili usate nella condizione.

*Esempio:* Specifica: stampa di 10 asterischi.

```
int i = 0;
while (i <= 10) {  // corretto: (i < 10)
    printf("*");
    i++;
}
```

Ma il programma stampa 11 asterischi.

Per evitare questo tipo di errori, verificare con dati semplici. Nell'esempio, se si verifica con la stampa di 1 asterisco (sostituendo 10 con 1 nella condizione del ciclo), ci si accorge immediatamente che in realtà verrebbero stampati 2 asterischi.

### 1.9.7. Schemi di ciclo

Esistono alcune operazioni di base molto comuni che richiedono l'utilizzo di cicli:

- *contatore*: conta il numero di valori in un insieme
- *accumulatore*: accumula i valori di un insieme
- *valori caratteristici di un insieme*: determina un valore caratteristico tra i valori in un insieme (ad esempio, il massimo, quando sui valori dell'insieme è definito un ordinamento)

Ciascun tipo di operazione ha alla base uno schema comune dell'istruzione di ciclo.

#### 1.9.7.1. Ciclo controllato da contatore

Una situazione comune di utilizzo dei cicli è quella in cui il ciclo fa uso di una variabile (detta *di controllo*) che ad ogni iterazione varia di un valore costante, ed il cui valore determina la fine del ciclo.

*Esempio:* stampa i quadrati degli interi da 1 a 10.

```c
int i = 1;
while (i <= 10) {
    printf("%d\n", i*i);
    i++;
}
```

I seguenti elementi sono comuni ai cicli controllati da contatore:

- si fa uso di una *variabile di controllo* del ciclo (detta anche *contatore* o *indice* del ciclo)
  *Esempio* i
- *inizializzazione* della variabile di controllo
  *Esempio* i = 1;
- *incremento* (o decremento) della variabile di controllo ad ogni iterazione
  *Esempio* i++;
- verifica se si è raggiunto il *valore finale della variabile di controllo*
  *Es.* (i <= 10)

#### 1.9.7.2. Ciclo di accumulazione

Il ciclo di accumulazione è caratterizzato dall'uso di una variabile (detta *accumulatore*) che ad ogni iterazione accumula il risultato di una operazione ed il cui valore alla fine del ciclo è il risultato complessivo dell'operazione.

*Esempio:* somma i quadrati degli interi da 1 a 10.

```c
int i = 1; // contatore
int s = 0; // accumulatore
while (i <= 10) {
    s += i*i;
    i++;
}
printf("%d\n", s);
```

I seguenti elementi sono comuni ai cicli di accumulazione:

- si fa uso di una *variabile accumulatore*
  *Esempio* s

- *inizializzazione* della variabile accumulatore all'elemento neutro dell'operazione da svolgere (0 per somma, 1 per prodotto, true per AND logico, false per OR logico, ...)
  *Esempio* s = 0;
- *accumulazione* dei valori ad ogni iterazione
  *Esempio* s += *valore*;

### 1.9.8. Altre istruzioni di ciclo

In C ci sono tre forme di istruzioni di ciclo:

- ciclo while
- ciclo for
- ciclo do

Il ciclo while sarebbe sufficiente per esprimere qualsiasi ciclo esprimibile con for e do. In alcune situazioni è però più conveniente codificare un algoritmo utilizzando gli altri tipi di ciclo.

### 1.9.9. Ciclo for

**Istruzione for**

*Sintassi:*

```
for (inizializzazione; condizione; incremento)
  istruzione
```

- *inizializzazione* è un'espressione con side-effect di inizializzazione di una variabile di controllo (tipicamente un'assegnazione), che può anche essere una dichiarazione con inizializzazione
- *condizione* è un'espressione booleana
- *incremento* è un'espressione con side-effect che tipicamente consiste nell'incremento della variabile di controllo
- *istruzione* è una singola istruzione (detta anche il *corpo* del ciclo for)

*Semantica:* è equivalente a

```
{ inizializzazione; while (condizione) {
istruzione; incremento; } }
```

*Esempio:* Stampa di 100 asterischi

```
for (int i = 0; i < 100; i++)
    printf("*");
```

è equivalente a

```
int i = 0;
while (i < 100) {
    printf("*");
    i++;
}
```

#### 1.9.9.1. Osservazioni sul ciclo for

- Se la variabile di controllo è dichiarata nella parte di *inizializzazione*, allora il suo campo di azione è limitato all'istruzione for. Questo si evince dalla traduzione nel ciclo while, nella quale l'intero codice corrispondente al ciclo for è racchiuso in un blocco.

*Esempio:*

```
for (int i = 0; i < 10; i++) {
    printf("%d\n", i*i);
}

printf("valore di i = %d\n",i);
// ERRORE! i non e' visibile
```

Ciascuna delle tre parti del for (*inizializzazione*, *condizione* e *incremento*) può anche mancare. In questo caso i ";" vanno messi lo stesso. Se manca *condizione*, viene assunta pari a true.

*Esempio:*

```
for (int i = 0; ; ) { ... }
```

La sintassi del for permette che le tre parti siano delle espressioni qualsiasi, purché *inizializzazione;* e *incremento;* siano delle istruzioni (in particolare, facciano side-effect).

Nell'uso del ciclo for è però buona norma:

- usare le tre parti del for in base al significato sopra descritto, con riferimento ad una *variabile di controllo*;
- non modificare la variabile di controllo nel corpo del ciclo.

In generale, *inizializzazione* e/o *incremento* possono essere una sequenza di espressioni con side-effect separate da ",". Questo permette di inizializzare e/o incrementare più variabili contemporaneamente.

*Esempio:* calcola e stampa le prime 10 potenze di 2.

```
int i, potDi2;
for (i=0, potDi2=1; i < 10; i++, potDi2 *= 2)
    printf("2 alla %d  = %d\n", i, potDi2);
```

#### 1.9.9.2. Ciclo for: esempi

Il ciclo for è usato principalmente per realizzare *cicli definiti*.

```
for (int i = 1; i <= 10; i++)
// valori: 1, 2, 3, ..., 10
```

```
for (int i = 10; i >= 1; i--)
// valori: 10, 9, 8, ..., 2, 1
```

```
for (int i = -4; i <= 4; i = i+2)
// valori: -4, -2, 0, 2, 4
```

```
for (int i = 0; i >= -10; i = i-3)
// valori: 0, -3, -6, -9
```

### 1.9.10. Ciclo do

Nel ciclo while la condizione di fine ciclo viene controllata all'inizio di ogni iterazione. Il ciclo do è simile al ciclo while, con la sola differenza che *la condizione di fine ciclo viene controllata alla fine di ogni iterazione*.

**Istruzione do**

*Sintassi:*

```
do
  istruzione
while (condizione);
```

- *condizione* è un'espressione booleana
- *istruzione* è una singola istruzione (detta anche il *corpo* del ciclo)

*Semantica:* è equivalente a

```
istruzione;
while (condizione)
  istruzione
```

Quindi:

- viene prima eseguita *istruzione*
- poi viene valutata prima la *condizione*, e se è vera, si torna ad eseguire istruzione *istruzione*, procedendo così fino a quando *condizione* diventa falsa
- a questo punto si passa ad eseguire l'istruzione che segue il ciclo do

*Esempio:* stampa di 100 asterischi

```
int i = 0;
do {
    printf("*");
    i++;
} while (i < 100);
```

#### 1.9.10.1. Osservazioni sul ciclo do

Dal momento che la condizione di fine ciclo viene valutata solo dopo aver eseguito il corpo del ciclo, ne segue che:

- *Il corpo del ciclo viene eseguito almeno una volta.* Quindi, il ciclo do consente la ripetizione di istruzioni, quando si vuole che queste istruzioni vengano in ogni caso eseguite almeno una volta.
- Non è in generale necessario inizializzare le variabili che compaiono nella condizione prima di iniziare l'esecuzione del ciclo. È sufficiente che tali variabili vengano inizializzate nel corpo del ciclo stesso.

*Esempio:* somma gli interi letti da input fino a quando non viene immesso 0.

```
int i;
int somma = 0;
do {
    printf("inserisci intero (0 per terminare): ");
    scanf("%d", &i);
    somma = somma + i;
} while (i != 0);
printf("\n Totale = %d\n", somma);
```

Si noti che la sintassi del ciclo do richiede che ci sia un ";" dopo while *(condizione)*. Per aumentare la leggibilità del programma, in particolare per evitare di confondere la parte while *(condizione)*; di un ciclo do, con un'istruzione while con corpo vuoto, conviene in ogni caso racchiudere il corpo del ciclo do in un blocco, ed indentare il codice come mostrato nell'esempio di sopra.

#### 1.9.10.2. Esempio di ciclo do: validazione dell'input

Spesso è necessario effettuare una validazione di un dato di input immesso dall'utente, e *nel caso l'utente abbia immesso un dato non valido,*

*ripetere la richiesta del dato stesso*. Questo può essere fatto utilizzando un ciclo do.

*Esempio:* scrivere un frammento di programma che legge da input un intero, ripetendo la lettura fino a quando non viene immesso un intero positivo, e restituisce l'intero positivo letto.

```
...
int i;
do {
    printf("inserisci un intero positivo: ");
    scanf("%d", &i);
} while (i <= 0);
// all'uscita dal ciclo i e' un intero positivo
...
```

Si noti che il precedente codice non è in grado di gestire correttamente tutte le situazioni di input errato, ad esempio il caso in cui l'utente immette un carattere alfabetico invece di un intero. Una trattazione completa delle tecniche di validazione dell'input esula dagli scopi del presente testo.

#### 1.9.10.3. Equivalenza tra ciclo while e ciclo do

Come segue dalla semantica, ogni ciclo do può essere sostituito da un ciclo while *equivalente*. Questo comporta però la necessità di duplicare il corpo del ciclo do.

*Esempio:*

```
do {
    printf("inserisci un intero positivo: ");
    scanf("%d", &i);
} while (i <= 0);
```

equivale a

```
printf("inserisci un intero positivo: ");
scanf("%d", &i);
while (i <= 0) {
    printf("inserisci un intero positivo: ");
    scanf("%d", &i);
}
```

### 1.9.11. Insieme completo di istruzioni di controllo

Due programmi si dicono **equivalenti** se, sottoposti agli stessi dati di ingresso,

- entrambi non terminano, oppure
- entrambi terminano producendo gli stessi risultati in uscita.

Un insieme di istruzioni di controllo del flusso si dice **completo** se per ogni programma nel linguaggio ne esiste uno equivalente scritto solo con le strutture di controllo contenute nell'insieme.

**Teorema di Böhm e Jacopini**

> La *sequenza*, l'*istruzione if-else* e l'*istruzione while* formano un insieme di istruzioni completo.

### 1.9.12. Esempio: calcolo del massimo comun divisore (MCD)

*Specifica:*

Vogliamo realizzare un programma che, dati due interi positivi $x$ ed $y$, calcoli e restituisca il massimo comun divisore $\text{mcd}(x, y)$.

Il massimo comun divisore di due interi $x$ ed $y$ è il più grande intero che divide sia $x$ che $y$ senza resto.

*Es.:* $\text{mcd}(12, 8) = 4$
$\text{mcd}(12, 6) = 6$
$\text{mcd}(12, 7) = 1$

#### 1.9.12.1. MCD: sfruttando direttamente la definizione

- cerchiamo il massimo tra i divisori comuni di $x$ ed $y$
- osservazione: $1 \leq \text{mcd}(x, y) \leq \min(x, y)$
  Quindi, si provano i numeri compresi tra 1 e $\min(x, y)$.
- Conviene iniziare da $\min(\text{x}, \text{y})$ e scendere verso 1. Appena si è trovato un divisore comune di $x$ ed $y$, lo si restituisce.

*Primo raffinamento dell'algoritmo:*

```
massimoComunDivisore di int x ed int y {
  int mcd;
  inizializza mcd al minimo tra x ed y
  while ((mcd > 1) && (divisore comune non trovato))
```

```
    if (mcd divide sia x che y)
      si è trovato un divisore comune
    else
      mcd--;
}
```

*Osservazioni:*

- Il ciclo termina sempre perchè ad ogni iterazione
  - o si è trovato un divisore,
  - o si decrementa mcd di 1 (al più si arriva ad 1).
- Per verificare se si è trovato il mcd si utilizza una variabile booleana (usata nella condizione del ciclo).
- Per verificare se x (o y) divide mcd si usa l'operatore %.

#### 1.9.12.2. MCD: osservazioni

Quante volte viene eseguito il ciclo nel precedente algoritmo?

- *caso migliore:* 1 volta, quando $x$ divide $y$ o viceversa
  Es. mcd(500, 1000)
- *caso peggiore:* min($x$,$y$) volte, quando mcd($x$,$y$) = 1
  Es. mcd(500, 1001)

Quindi, il precedente algoritmo si comporta male se $x$ e $y$ sono grandi e $\mathrm{mcd}(x, y)$ è piccolo.

#### 1.9.12.3. MCD: usando il metodo di Euclide

Il **metodo di Euclide** permette di ridursi più velocemente a numeri più piccoli, sfruttando la seguente proprietà:

$$\mathrm{mcd}(x, y) = \begin{cases} x \,(\text{oppure } y) & \text{se } x = y \\ \mathrm{mcd}(x - y, y) & \text{se } x > y \\ \mathrm{mcd}(x, y - x) & \text{se } x < y \end{cases}$$

Questa proprietà si dimostra facilmente, mostrando che i divisori comuni di $x$ e $y$ sono anche divisori di $x - y$ (nel caso in cui $x > y$).

*Es.* $\text{mcd}(12,8) = \text{mcd}(12-8,8) = \text{mcd}(4,8-4) = 4$ Per ottenere un algoritmo si applica ripetutamente il procedimento fino a che non si ottiene che $x = y$. Ad esempio:

| $x$ | $y$ | maggiore − minore |
|---|---|---|
| 210 | 63 | 147 |
| 147 | 63 | 84 |
| 84 | 63 | 21 |
| 21 | 63 | 42 |
| 21 | 42 | 21 |
| 21 | 21 | $\Longrightarrow$ mcd(21,21) = mcd(21,42) = $\cdots$ = mcd(210,63) |

L'algoritmo può essere realizzato in C al seguente modo:

```
int main ()
{
  int x = 210;
  int y = 63;
  while (x != y) {
    if (x > y)
      x = x - y;
    else                    // significa che y > x
      y = y - x;
  }
  printf("mcd = %d\n", x);
  return 0;
}
```

#### 1.9.12.4. MCD: ammissibilità dei valori degli argomenti

Cosa succede nel precedente algoritmo nei seguenti casi:

- Se $x = y = 0$?
  Il risultato è 0.
- Se $x = 0$ e $y > 0$ (o viceversa)?
  Il risultato dovrebbe essere $y$, ma l'algoritmo entra in *un ciclo infinito*.
- Se $x < 0$ e $y$ è qualunque (o viceversa)?
  L'algoritmo entra in *un ciclo infinito*.

Quindi, se si vuole tenere conto del fatto che il programma possa essere eseguito con valori qualsiasi dei parametri, è necessario inserire una opportuna verifica.

```c
int main ()
{
  int x = 210;
  int y = 63;
  if ((x > 0) && (y > 0)) {
    while (x != y)
      if (x > y)
        x = x - y;
      else                // significa che y > x
        y = y - x;
     printf("mcd = %d\n", x);
  }
  else printf("dati errati");
 return 0;
}
```

#### 1.9.12.5. MCD: osservazioni sul metodo di Euclide

Cosa succede nel precedente algoritmo se $x$ è molto maggiore di $y$ (o viceversa)?

*Esempio:*

| mcd(1000, 2) | | mcd(1001, 500) | |
|---|---|---|---|
| 1000 | 2 | 1001 | 500 |
| 998 | 2 | 501 | 500 |
| 996 | 2 | 1 | 500 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 2 | 2 | 1 | 1 |

Per comprimere questa lunga sequenza di sottrazioni è sufficiente osservare che quello che in fondo si calcola è il resto della divisione intera.

#### 1.9.12.6. MCD: metodo di Euclide con i resti

**Metodo di Euclide:** sia $x = y \cdot k + r$ (con $0 \leq r < y$)

$$\mathrm{mcd}(x,y) = \begin{cases} y, & \text{se } r = 0 \ (x \text{ multiplo di } y) \\ \mathrm{mcd}(r,y), & \text{se } r \neq 0 \end{cases}$$

L'algoritmo può essere realizzato in C nel seguente modo:

```c
int main ()
{
  int x = ...;
  int y = ...;
  while ((x != 0) && (y != 0)) {
    if (x > y)
      x = x % y;
    else
      y = y % x;
  }
  if (x != 0) printf("mcd = %d\n", x);
  else printf("mcd = %d\n", y);
  return 0;
}
```

### 1.9.13. Cicli annidati (o doppi cicli)

Il corpo di un ciclo può contenere a sua volta un ciclo, chiamato **ciclo annidato**. È possibile annidare un qualunque numero di cicli.

*Esempio:* stampa della tavola Pitagorica

```c
int main () {
    int N = 10; // dimensione della tavola
    int riga, colonna;
    for (riga = 1; riga <= N; riga++) {
        for (colonna = 1; colonna <= N; colonna++)
            printf(" %3d  ", riga * colonna);
        printf("\n");
    }
    return 0;
}
```

*Output prodotto:*

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

#### 1.9.13.1. Esempio di doppio ciclo: stampa di una piramide di asterischi

```c
int altezza;
printf("Altezza = ");
scanf("%d", &altezza);
for (int riga = 1; riga <= altezza; riga++) {
    // 1 iterazione per ogni riga della piramide
    for (int i = 1; i <= altezza - riga; i++)
        printf(" ");  // stampa spazi bianchi iniziali
    for (int i = 1; i <= riga * 2 - 1; i++)
        printf("*"); // stampa sequenza di asterischi
    printf("\n");     // va a capo: fine riga
}
```

Nell'esempio della stampa della tavola pitagorica, il numero di iterazioni del ciclo più interno era fisso. In generale, il numero di iterazioni di un ciclo più interno può dipendere dall'iterazione del ciclo più esterno.

*Esempio:* stampa di una piramide di asterischi

```
Piramide di altezza 4
   *
  ***
 *****
*******
```

| riga | blank | * |
|---|---|---|
| 1 | 3 | 1 |
| 2 | 2 | 3 |
| 3 | 1 | 5 |
| 4 | 0 | 7 |

Per la stampa della generica riga $r$: stampa $(\text{altezza} - r)$ blank e $(2 \cdot r - 1)$ asterischi.

#### 1.9.13.2. Esempio: potenza con un doppio ciclo

```
int base = ...;
int esponente= ...;
int risultato = 1;
while (esponente > 0) {
   esponente--;
   // risultato = risultato * base
   int moltiplicando = risultato;
   int moltiplicatore = base;
   int prodotto = 0;
   while (moltiplicatore > 0) {
        moltiplicatore--;
        prodotto = prodotto + moltiplicando;
    }
    risultato = prodotto;
}
printf("%d", risultato);
```

#### 1.9.13.3. Esempio: numeri di Fibonacci

Al nome del matematico Fibonacci è legata una famosa serie numerica, così definita:

- Fib(0)=1;
- Fib(1)=1;
- Fib(n)=Fib(n-1)+Fib(n-2)

Scrivere un programma che legge un intero positivo (altrimenti reitera la richiesta), e calcola il relativo numero di Fibonacci. li

#### 1.9.13.4. Numeri di Fibonacci: soluzione

**fibo.c**

```
#include <stdio.h>  /* funzioni di I/O */

int main ()
{
  int Fib,N,n1,n2;
  printf("Calcolo il numero di Fibonacci\n");
  do {
    printf("Inserisci un numero (>=0): ");
    scanf("%d",&N);
  } while (N<0);
```

```
    if ((0 == N) || (1 == N))
      Fib=1;
    else {
      n1=1; n2=1;
      for (int i = 2; i <= N; i++) {
        Fib= n2 + n1;
        n2 = n1;
        n1 = Fib;
      }
    }
    printf("\nNumero di Fibonacci di %d = %d\n", N, Fib)
      ;
    return 0;
}
```

## 1.10. Istruzioni di controllo del flusso

Le istruzioni di controllo del flusso determinano la successiva istruzione da eseguire. In questo senso, le istruzioni if-else, if, switch, while, for, do sono istruzioni di controllo del flusso. Esse non permettono però di stabilire in modo arbitrario la prossima istruzione da eseguire, ma forniscono una *strutturazione del programma* che determina il flusso di esecuzione.

C, come altri linguaggi, permette però anche di usare (seppur limitatatamente) *istruzioni di salto*, che sono istruzioni di controllo del flusso che causano l'interruzione del flusso di esecuzione ed il salto ad una istruzione diversa dalla successiva nella sequenza di istruzioni del programma.

**Istruzioni di salto:**

- break (salto all'istruzione successiva al ciclo o allo switch corrente)
- continue (salto alla condizione del ciclo)
- goto (salto all'istruzione etichettata)

*Note:*

- L'uso di istruzioni di salto va in generale evitato. Esse vanno usate solo in casi particolari. In questo corso ne faremo solo un accenno.
- Anche l'istruzione return può essere usata per modificare il flusso di esecuzione, nelle funzioni definite (vedi Unità 5).

### 1.10.1. Istruzione break per uscire da un ciclo

Abbiamo già visto nell'Unità 3 che l'istruzione break permette di uscire da un'istruzione switch. In generale, *break permette di uscire prematuramente da un'istruzione switch, while, for o do.*

*Esempio:* ciclo che calcola la radice quadrata di 10 reali letti da input. Si vuole interrompere il ciclo non appena viene immesso un reale negativo.

```
double a;
for (int i = 0; i < 10; i++) {
  printf("Immetti un reale nonnegativo ");
  scanf("%d", &a);
  if (a >= 0)
    printf("%f",sqrt(a));
  else {
    printf("Errore");
    break;
  }
}
```

*Nota:* nel caso di cicli annidati o di switch annidati dentro un ciclo, l'esecuzione di un break fa uscire di *un solo livello*.

#### 1.10.1.1. Eliminazione del break

L'esecuzione di un'istruzione break *altera il flusso di controllo*. Quindi, quando viene usata nei cicli:

- si perde la strutturazione del programma
- si guadagna in efficienza rispetto ad implementare lo stesso comportamento senza fare uso del break.

In generale, è sempre possibile **eliminare un'istruzione break**. Ad esempio, l'istruzione

```
while (condizione) {
  istruzioni-1
  if (condizione-break) break;
  istruzioni-2
}
```

equivale a

```
finito = 0;
while (condizione && !finito) {
  istruzioni-1
  if (condizione-break) finito = 1;
  else { istruzioni-2 }
}
```

La scelta se eliminare o meno un'istruzione break deve essere fatta valutando da una parte il guadagno in efficienza del programma con break rispetto a quello senza, e d'altra parte la perdita in leggibilità dovuta alla presenza del break.

#### 1.10.1.2. Esempio di eliminazione del break

Con riferimento all'esempio precedente, il codice senza break è il seguente:

```
double a;
int errore = 0;

for (int i = 0; (i < 10) && !errore; i++) {
  printf("Immetti un reale nonnegativo ");
  scanf("%lf", &a);
  if (a > 0)
    printf("%f",sqrt(a));
  else {
    printf("Errore");
    errore = 1;
  }
}
```

### 1.10.2. Istruzione continue

L'istruzione continue si applica solo ai cicli e *comporta il passaggio alla successiva iterazione del ciclo*, saltando le eventuali istruzioni che seguono nel corpo del ciclo.

*Esempio:*

```
for (int i = 0; i < n; i++) {
  if (i % 2 == 0)
    continue;
  printf("%d",i);
}
```

stampa i valori dispari tra 0 e n.

Si osservi che l'istruzione continue all'interno di un ciclo for fa comunque eseguire le istruzioni-incremento del ciclo.

*Nota:* Un possibile uso di continue è all'interno di un ciclo di lettura, nel quale vogliamo effettuare un'elaborazione sui dati letti solo se è verificata una determinata condizione. Bisogna però assicurarsi che ad ogni iterazione del ciclo venga in ogni caso letto il prossimo valore. Altrimenti il ciclo non termina.

```
leggi il prossimo dato;
while (condizione) {
  if (condizione-sul-dato-corrente)
    continue; // ERRORE! viene saltata la lettura del
dato
  elabora il dato;
  leggi il prossimo dato;
}
```

### 1.10.3. Istruzioni di salto goto

L'istruzione di salto goto deriva dal linguaggio macchina dove ha un ruolo fondamentale per consentire la realizzazione dei cicli. Il citato teorema di Böhm e Jacopini ha mostrato, tra l'altro, che essa non è necessaria ai fini della completezza del linguaggio.

L'istruzione di salto comporta una interruzione del flusso dell'esecuzione del programma, che prosegue con l'istruzione specificata nel goto.

Per consentire il salto, le istruzioni possono avere delle **etichette**:

```
etichetta: istruzione-ciclo;
```

Le etichette devono essere degli identificatori.

L'istruzione goto ha quindi la seguente sintassi:

```
goto etichetta;
```

*etichetta* è un identificatore che deve individuare una istruzione del programma.

L'esecuzione dell'istruzione goto imposta il programma a proseguire con l'istruzione specificata dall'etichetta.

*Esempio:* interruzione di un ciclo

```
int i = 0; float x;
while (i < 100) {
  printf("inserisci numero positivo: ");
  scanf("%f", &x);
  if (x<=0) goto errore;
  i++;
}
errore: printf("errore: programma interrotto\n");
```

L'istruzione goto errore interrompe il ciclo.

*Nota:* l'uso di etichette e di istruzioni goto è considerata una cattiva pratica di programmazione e va riservato a casi particolari. In questo corso *non ne faremo uso*!

# 2. Tipi di dato primitivi

## 2.1. Tipi di dato e allocazione di memoria

I tipi di dato vengono usati nelle dichiarazioni di variabili e costanti per determinare quali valori esse possono assumere e quali operazioni possono essere effettuate su di esse.

Ad esempio, l'istruzione int i; dichiara la variabile i, indicando che essa può assumere (solo) valori interi.

Come si è visto, quando si dichiara una variabile, viene allocata memoria per memorizzarne il valore assegnato. La quantità di memoria che deve essere allocata dipende dal tipo associato alla variabile. Ad esempio, nel caso di variabili di tipo int, vengono tipicamente allocati 32 bit (4 byte).

Com'è noto, la dimensione della memoria influisce sull'intervallo di valori rappresentabili.

Per trattare le informazioni di tipo numerico, il C definisce tre tipi primitivi:

- int, usati per rappresentare numeri interi;
- float, usati per la rappresentazione di reali in virgola mobile;
- double, usati per la rappresentazione di reali in virgola mobile con doppia precisione.

Per trattare caratteri alfanumerici e simboli speciali, il C offre il tipo di dato:

- char

Il C fornisce inoltre i qualificatori short, long, unsigned che consentono di modificare la dimensione (in bit) del formato di alcuni tipi e la rispettiva interpretazione.

Il C non mette a disposizione un tipo primitivo booleano, ma sfrutta gli interi, interpretando il valore 0 come false e qualunque valore non nullo come true. Inoltre, alcune versioni del C mettono a disposizione un tipo booleano (ad es., bool o _Bool in C99). Tale tipo, tuttavia non è altro che una ridefinizione del tipo intero.

### 2.1.1. Il tipo di dato primitivo int

Rappresentiamo le caratteristiche più rilevanti dei tipi di dato primitivi riportando in una tabella le seguenti informazioni:

- dimensione della rappresentazione[1] , ovvero numero di bit allocati per memorizzare il contenuto della variabile;
- dominio del tipo, ovvero l'insieme dei valori rappresentati dal tipo;
- operazioni comuni disponibili sugli elementi del dominio;
- letterali, ovvero gli elementi sintattici del C che denotano valori del dominio (ad esempio 23).

| Tipo | int | |
|---|---|---|
| Dimensione | 32 bit (4 byte) | |
| Dominio | numeri interi in $[-2^{31}, +2^{31}-1]$ (oltre 4 miliardi di valori) | |
| Operazioni | + | somma |
| | - | sottrazione |
| | * | prodotto |
| | / | divisione intera |
| | % | resto della divisione intera |
| Letterali | sequenze di cifre che denotano valori del dominio (es. 275930) | |

[1] In alcuni tipi, questa grandezza dipende dalla macchina e/o dal compilatore (vedi funzione sizeof più avanti). Noi faremo riferimento alle combinazioni più comuni nei calcolatori moderni.

*Esempio:*

```
int a,b,c;  // Dichiarazione di variabile di tipo int
a = 1;      // Uso di letterali
b = 2;
c = a + b;  // Espressione aritmetica che coinvolge
            // operatori del linguaggio
```

I valori limite del tipo int sono definiti nelle costanti INT_MIN e INT_MAX nel file di sistema limits.h

#### 2.1.1.1. Overflow numerico

Poiché l'insieme dei valori interi rappresentabili mediante un tipo primitivo è limitato ad un dato intervallo (ad esempio, [-$2^{31}$,$2^{31}$-1] per il tipo int), applicando operatori aritmetici a valori di tipo intero si può verificare il cosiddetto **overflow numerico** (trabocco). Cioò avviene quando il risultato dell'operazione non può essere rappresentato con il numero di bit messi a disposizione dal tipo.

*Esempio:*

**overflow.c**

```
int x = INT_MAX;
printf("%d\n",x); // Stampa 2147483647
       // (massimo valore rappresentabile con int)
x++; // Incrementa x di 1
printf("%d\n",x); // Stampa -2147483648
       // (minimo valore rappresentabile con int)
```

L'overflow nell'esempio è dovuto al fatto che il valore 2147483647 equivale, in rappresentazione binaria, a $0\underbrace{1\ldots1}_{31}$. Incrementando tale valore di 1, si ottiene $1\underbrace{0\ldots0}_{31}$ (si noti il bit di segno pari ad 1) che, in complemento a due, rappresenta appunto -2147483648. Considerazioni analoghe valgono nel caso in cui il valore INT_MIN sia decrementato.

Si noti che, in generale, il comportamento conseguente ad un overflow dipende dal compilatore. Inoltre, linguaggi diversi possono trattare gli overflow in maniera diversa, o addirittura prevedere meccanismi che lo impediscono. Ad esempio, Python sfrutta una rappresentazione degli interi più flessibile che permette di codificare valori arbitrari.

### 2.1.2. I qualificatori short, long e unsigned

I qualificatori short, long e unsigned, permettono di modificare la dimensione del formato di alcuni tipi e la relativa interpretazione, specificando di fatto nuovi tipi. Ciascuno di essi può essere usato in combinazione con int (ma non esclusivamente).

Il qualificatore short, associato al tipo int, viene usato per indicare il tipo intero in un intervallo di valori ridotto.

| Tipo | short int |
|---|---|
| Dimensione | 16 bit (2 byte) |
| Dominio | numeri interi in $[-2^{15}, +2^{15}-1] =$ [-32768,+32767] |

*Esempio:*

```
short int a,b; // Dichiarazione di variabile di tipo
    short int
a = 22700; // Uso di letterali
```

Il qualificatore long, associato al tipo int, viene usato per indicare il tipo intero in un intervallo di valori esteso.

| Tipo | long int |
|---|---|
| Dimensione | 64 bit (8 byte) |
| Dominio | numeri interi in $[-2^{63}, +2^{63}-1]$ |

*Esempio:*

```
long a,b;          // Dichiarazione di variabili di tipo
     long int
a = 9000000000L; // Uso di letterali
b = a + 2L;       // Espressione aritm. che coinvolge
                  // operatori del linguaggio
```

Il tipo long int può essere a 32 o a 64 bit a seconda del compilatore usato e dell'architettura del calcolatore. Nel caso di 32 bit è equivalente al tipo int.

Il qualificatore unsigned indica che il primo bit del vettore non deve essere considerato come bit di segno.

| Tipo | unsigned int |
|---|---|
| Dimensione | 32 bit (4 byte) |
| Dominio | numeri interi in $[0, +2^{32} - 1]$ |

*Esempio:* L'istruzione unsigned int a; dichiara la variabile a di tipo *intero senza segno*.

Si noti che, non dovendo usare il primo bit per rappresentare il segno, il valore massimo rappresentabile con un unsigned int è maggiore (doppio) rispetto a quello rappresentabile con un int.

Il qualificatore unsigned può essere combinato con short e long (in qualunque ordine), quando questi siano applicati al tipo int. Il numero di valori rappresentabili cambia di conseguenza, per effetto dell'eliminazione del segno.

*Esempio:* L'istruzione unsigned short int x; definisce la variabile x di tipo short int senza segno. I valori che la variabile può assumere sono nell'intervallo $[0, 2^{16} - 1]([0, 65534])$.

Quando i qualificatori appena visti sono applicati al tipo int, quest'ultimo può essere omesso (e tipicamente lo è). Ad esempio, la dichiarazione unsigned int x è equivalente ad unsigned x e unsigned long int x è equivalente ad unsigned long x.

Precisiamo infine che unsigned può essere associato anche al tipo char e long al tipo double (v. seguito).

### 2.1.3. Tipi di dato primitivi reali

Come anticipato, il C mette a disposizione due tipi di dato primitivi, float e double, per la rappresentazione dei numeri reali in virgola mobile.

#### 2.1.3.1. Il tipo di dato float

Il tipo float identifica i numeri reali, rappresentati in virgola mobile, a precisione singola.

| Tipo | float | | |
|---|---|---|---|
| Dimensione | 32 bit (4 byte) | | |
| Dominio | $2^{32}$ reali pos. e neg. | min | $1.4012985 \cdot 10^{-38}$ |
| | | max | $3.4028235 \cdot 10^{+38}$ |
| | | Precisione | $\sim$ 7 cifre decimali |
| Operazioni | + | somma | |
| | - | sottrazione | |
| | * | prodotto | |
| | / | divisione | |
| Letterali | sequenze di cifre con punto decimale terminate con una f che denotano valori del dominio (es. `3.14f`) | | |
| | rappresentazione in notazione scientifica (es. `314E-2f`) | | |

*Esempio:*

```
float a;   // Dichiarazione di variabile di tipo float
a = 3.14f; // Uso di letterali
a = a*2f;  // Espressione aritmetica che coinvolge
           // operatori del linguaggio
```

#### 2.1.3.2. Il tipo di dato double

Il tipo double identifica i numeri reali, rappresentati in virgola mobile, a precisione doppia.

| Tipo | double | | |
|---|---|---|---|
| Dimensione | 64 bit (8 byte) | | |
| Dominio | $2^{64}$ reali pos. e neg. | min | $1.7976931348623157 \cdot 10^{-308}$ |
| | | max | $2.250738585072014 \cdot 10^{+308}$ |
| | | Precisione | $\sim$ 15 cifre decimali |
| Operazioni | + | somma | |
| | - | sottrazione | |
| | * | prodotto | |
| | / | divisione | |
| Letterali | sequenze di cifre con punto decimale, opzionalmente terminate con `d`, che denotano valori del dominio (es. `3.14`) | | |
| | rappresentazione in notazione scientifica (es. `314E-2`) | | |

*Esempio:*

```
double a; // Dichiarazione di variabile di tipo double
a = 628E-2;  // Uso di letterali
a = a+1.2;   // Espressione aritmetica che coinvolge
             // operatori del linguaggio
```

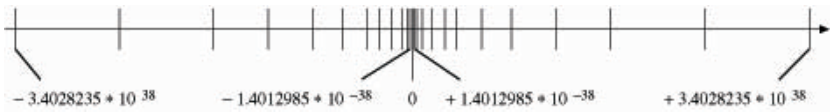
Il qualificatore long può anche essere applicato al tipo double, per identificare reali con doppia precisione estesa.

#### 2.1.3.3. Precisione nella rappresentazione: errori di arrotondamento

In C, i numeri reali vengono rappresentati facendo uso della rappresentazione in virgola mobile (*floating point*, da cui il nome float), ovvero riservando un bit per il segno $s$, un certo numero di bit per l'esponente $e$ ed i restanti bit per la mantissa $m$ (il numero effettivo dei bit riservati per ogni componente dipende dal compilatore usato). Il valore del numero rappresentato è pari a $(-1)^s \cdot m \cdot 2^e$.

Come visto, il tipo float permette di rappresentare valori nell'intervallo $[-3.4028235 \cdot 10^{38}, +3.4028235 \cdot 10^{38}]$. Tuttavia, diversamente dagli interi, i float non permettono di rappresentare tutti i numeri compresi tra gli estremi dell'intervallo (considerazioni analoghe valgono anche per i double). Ad esempio il numero più vicino a 1222333440.0 rappresentabile in float è 1222333568.0, mentre non è possibile rappresentare il numero 122333441.0. Ciò è dovuto al fatto che, essendo il numero di bit per la rappresentazione della mantissa finito, la precisione dei numeri rappresentabili è necessariamente finita.

Questo aspetto è illustrato nella figura seguente: i numeri rappresentati in virgola mobile (float e double) sono tanto più vicini tra loro quanto più sono prossimi allo zero e, viceversa, tanto più lontani fra loro quanto più vicini al massimo valore rappresentabile (in valore assoluto).




Il fatto che si debba lavorare con precisione finita dà luogo ad approssimazioni nei calcoli dovute ad errori di arrotondamento.

*Esempio:* Si consideri il seguente frammento di codice

**approssimazione.c**

```
float x = 1222333444.0;
printf("x = %f\n", x);
x += 1.0;
printf("x + 1.0 = %f\n", x);
```

Quando eseguito, produce il seguente output:

```
x = 1222333440.000000
x + 1.0 = 1222333440.000000
```

Si noti che l'operazione di incremento non produce alcun effetto sul valore di x.

Nel seguente frammento, invece, il valore della variabile x cambia a seguito dell'incremento.

**approssimazione2.c**

```
float x = 1222333440.0;
printf("x = %f\n", x);
x+=65;
printf("x + 65 = %f\n", x);
```

L'output prodotto è infatti il seguente:

```
x = 1222333440.000000
x + 65 = 1222333568.000000
```

Si noti, tuttavia, che il risultato dell'incremento è diverso dal valore atteso, ovvero 1222333505.000000. Infatti, non essendo il valore atteso rappresentabile come float, esso viene approssimato al valore rappresentabile ad esso più prossimo. Osserviamo infine che se avessimo incrementato la variabile x di 64, l'operazione non avrebbe prodotto alcun cambiamento sul valore della variabile.

#### 2.1.3.4. Precisione nelle misure

La precisione del risultato di un'operazione dipende dalla precisione con cui si conoscono i dati.

Ad esempio, se conosciamo le dimensioni di un rettangolo con una precisione di una sola cifra decimale, l'area non potrà avere una precisione superiore, quindi non ha senso considerare come significativa la sua seconda cifra decimale:

9.2 * 5.3 = 48.76 (la seconda cifra decimale non è significativa)
9.25 * 5.35 = 49.48 (qui lo è)

Questo non è causato dalla rappresentazione dei numeri nel linguaggi di programmazione, ma dai limiti sulla conoscenza dei valori di input di un problema.

### 2.1.4. Il tipo di dato primitivo char

Una variabile di tipo char può contenere un carattere alfanumerico o un simbolo. Il dominio del tipo char è normalmente costituito dai caratteri dello standard ASCII [2], contenente 128 caratteri. Il formato del tipo char è a 8 bit. Lo standard ASCII stabilisce una corrispondenza tra numeri e simboli alfabetici, numerici o simboli speciali (come il ritorno a capo o tab).

Ad esempio, il carattere 'A' corrisponde al codice numerico 65, il carattere 'B' al codice numerico 66, ecc.

Per maggiori dettagli sullo standard ASCII, si veda ad esempio il sito web: `www.asciitable.com`.

I letterali del tipo char sono denotati in diversi modi; il più semplice è tramite apici singoli (').

*Esempio:*

```
char c = 'A';
char d = '1';
```

La variabile c viene dichiarata di tipo char ed inizializzata al valore (carattere) `A`. La variabile d viene dichiarata di tipo char ed inizializzata al valore (carattere) 1.

Nel codice dello standard ASCII, le lettere alfabetiche e le cifre hanno codici consecutivi e ordinati (ad esempio, il codice del carattere 'b' è uguale a: codice del carattere 'a' + 1, il codice del carattere 'B' è uguale a: codice del carattere 'A' + 1, il codice del carattere '1' è uguale a: codice del carattere '0' + 1).

---

[2] American Standard Code for Information Interchange.

### 2.1.5. Il tipo char come tipo intero

Il tipo char rappresenta in realtà un tipo intero a 8 bit (i cui valori vengono interpretati come caratteri del codice ASCII). Esso permette di rappresentare i valori interi nell'intervallo $[-128, 127]$.

*Esempio:*

```
char c = 'A';
int i = 65;
printf("%c\n",i); // Stampa: A
printf("%d\n",c); // Stampa: 65
```

Le operazioni disponibili per i char sono le stesse disponibili per i tipi interi.

Analogamente agli interi, è possibile applicare il qualificatore unsigned anche al tipo char. In questo modo si possono rappresentare gli interi nell'intervallo $[0, 255]$.

### 2.1.6. Tipo Booleano

Nella versione standard del linguaggio C non esiste un tipo di dato primitivo Booleano, ma vengono usati i valori numerici interi con il seguente significato: 0 corrisponde a falso, $\neq 0$ corrisponde a vero.

#### 2.1.6.1. Operatori Booleani

Esistono tuttavia gli operatori Booleani che possono essere applicati a tipi di dato interi con il consueto significato.

| Operatore | Significato |
|---|---|
| && | AND: congiunzione |
| \|\| | OR: disgiunzione |
| ! | NOT: negazione |

### 2.1.7. Altri tipi di dato primitivo

Nello standard C99 (adottato dal compilatore g++) sono stati introdotti due nuovi tipi di dato primitivi:

- bool (definito in stdbool.h) per i valori Booleani (letterali: true e false)

- complex (definito in complex.h) per valori complessi (letterali 1+2*I)

Gli operatori Booleani si applicano anche al tipo bool e gli operatori aritmetici si applicano anche al tipo complex.

## 2.2. Costanti e numeri magici

Un *numero magico* è un letterale numerico che compare nel codice senza spiegazioni. La presenza di numeri magici diminuisce la leggibilità e la modificabilità dei programmi.

*Esempio:*

```
int compenso = 20 * orelavoro;
// cosa rappresenta 20?
```

Per evitare inconvenienti dovuti all'impossibilità di interpretare i numeri magici, di regola si usano *nomi simbolici* che siano autoesplicativi. Potremmo usare una variabile, ad esempio COMPENSO_ORARIO a cui assegnare il valore del numero magico. In tal modo, tuttavia, potremmo accidentalmente modificarne il valore. Per impedire che ciò accada definiamo COMPENSO_ORARIO come una *costante*, ovvero come una variabile il cui valore non cambia:

```
const int COMPENSO_ORARIO = 20;
int compenso = COMPENSO_ORARIO * orelavoro;
```

COMPENSO_ORARIO è una *costante*, cioè una variabile il cui contenuto non può variare durante l'esecuzione del programma. Questo effetto è ottenuto usando il qualificatore const nella dichiarazione della variabile, che indica che il valore non può essere modificato (cioè rimane costante).

I principali vantaggi nell'usare le costanti sono:

- *Leggibilità del programma:* un identificatore di costante con un nome significativo è molto più leggibile di un numero (COMPENSO_ORARIO è autoesplicativo, 20 non lo è);
- *Modificabilità del programma:* per modificare un valore costante usato nel programma basta semplicemente cambiare la definizione

della costante (es. const int COMPENSO_ORARIO = 35), usando i numeri magici dovremmo modificare tutte le occorrenze del valore nel programma (es. sostituendo dappertutto 20 con 35).

Oltre alla definizione di costanti tramite la parola chiave const, è possibile definire costanti anche tramite la direttiva #define.

*Esempio:*

```
#define COMPENSO_ORARIO 20
...
int compenso = COMPENSO_ORARIO * orelavoro;
```

Questa è una direttiva che viene processata dal pre-compilatore, sostituendo ad ogni occorrenza del nome definito (nell'esempio `COMPENSO_ORARIO`) il valore corrispondente (nell'esempio `20`).

Si osservi che nel caso della direttiva #define non viene effettuato nessun controllo sui tipi.

*Esempio:*

```
#define COMPENSO_ORARIO 20.50f
```

In questo caso, poiché COMPENSO_ORARIO non è una costante del programma, essa non ha un tipo associato. Pertanto il compilatore non verifica alcuna corrispondenza tra tipi quando il valore viene sostituito all'etichetta COMPENSO_ORARIO (se, ad esempio, si assegnasse il valore `20.50f` ad una costante intera, il compilatore ci informerebbe della possibile perdita di informazione).

## 2.3. Espressioni numeriche

Analogamente al linguaggio Python, variabili, costanti e letterali di tipo numerico (int, float, double, char) possono essere combinati usando *operatori aritmetici* (ma non esclusivamente) per creare *espressioni numeriche*, ovvero formule che rappresentano valori numerici.

Gli operatori aritmetici più comuni in C sono:

- +, - (unari), applicabili a tutti i tipi numerici;
- +, -, *, / (binari), applicabili a tutti i tipi numerici;
- % (resto della divisione, binario), applicabile solo agli interi.

Quando un valore di tipo char occorre in un'espressione aritmetica, esso viene semplicemente interpretato come intero.

*Esempio:*

```
char a = 'A';
char b = 'B';
printf("%d\n",a); // Stampa 65
printf("%d\n",b); // Stampa 66
printf("%d\n",a+b); // Stampa 131
```

Nell'ultima istruzione, il valore delle variabili a e b, rispettivamente 65 e 66, viene interpretato come intero. L'espressione a+b restituisce pertanto il valore 131.

Le regole di composizione delle espressioni in cui compaiono gli operatori aritmetici sono essenzialmente quelle dell'aritmetica. Ad esempio, 3 + a è un'espressione (se a è una variabile di tipo numerico) ma 3 + a - * non lo è.

Anche le regole di precedenza e le associatività rispettano quelle dell'aritmetica. Gli operatori a precedenza più alta sono + e - unari, *, / e %, mentre quelli a precedenza più bassa sono + e - binari. Gli operatori unari sono associativi a destra, quelli binari a sinistra. Le parentesi tonde ( e ) possono essere utilizzate per esplicitare la struttura dell'espressione (cioè l'ordine di esecuzione delle operazioni).

Un'espressione viene valutata nella maniera usuale a partire dagli elementi e dagli operatori di cui le espressioni sono composte. Variabili e costanti (di tipo numerico), che, insieme ai letterali, rappresentano i casi più semplici di espressione, hanno valore pari al valore ad esse assegnato.

Il valore di un'espressione può essere intero o reale, a seconda degli elementi che la compongono. Elementi di tipo intero o reale possono essere combinati in una stessa espressione. In tal caso, il valore ottenuto è di tipo reale.

*Esempio:*

```
123 // vale 123 (intero)
int x = 12;
x // vale 12 (intero)
float y = 1.2;
x+y // vale 13.2 (reale)
10/3+10%3 // vale 4 (intero)
2*-3+4 // vale  -2 (intero)
int a = 5, b =7;
2*a+b // vale  17 (intero)
2*(a+b) // vale  24 (intero)
```

Si noti l'impatto delle parentesi e della precedenza degli operatori nelle ultime due espressioni.

Come in Python, Il valore di un'espressione può essere assegnato ad una variabile (o costante).

L'assegnazione avviene valutando prima l'espressione a destra dell'operatore = e successivamente assegnando il valore ottenuto alla variabile a sinistra.

*Esempio:*

```
int x = 3;
int y = x + 4;
```

Se il tipo dell'espressione è diverso da quello della variabile a cui viene assegnata, il valore dell'espressione viene *convertito automaticamente* prima che l'assegnazione abbia effetto. Questo meccanismo, detto *cast*, che verrà analizzato in dettaglio più avanti.

## 2.4. Espressioni con side-effect ed istruzioni

Il termine *espressione* in C indica due diverse nozioni:

- le espressioni che hanno come effetto solamente il calcolo di un valore, come quelle viste in precedenza;
- le espressioni che, oltre a calcolare un valore, effettuano operazioni sulla memoria, ad esempio, come si vedrà a breve, un'assegnazione. Per queste ultime useremo il termine *espressioni con side-effect*: il termine side-effect si usa per indicare una modifica nella memoria del programma. Espressioni di questo tipo possono essere trasformate in istruzioni se terminate da " ; " .

### 2.4.1. Espressioni con assegnazione

L'operatore di assegnazione può essere usato anche per costruire espressioni.

*Esempio:* x = 7 è un'espressione (con side-effect) che può essere utilizzata come un'espressione qualunque:

```
int x = 0;
printf("%d",x=7);
printf("%d",x);
```

Si noti che terminando l'espressione con il carattere ;, si ottiene l'istruzione di assegnazione x = 7;.

Il valore di un'espressione del tipo variabile = espressione è pari al valore dell'espressione a destra dell'operatore di assegnazione. Pertanto, l'espressione dell'esempio precedente ha valore 7, che è infatti il valore stampato quando la seconda riga viene eseguita.

Quando un'espressione con side-effect viene valutata, vengono anche eseguite le operazioni in memoria corrispondenti. Nella seconda riga dell'esempio precedente, viene assegnato il valore 7 alla variabile x, che è anche il valore stampato quando l'ultima riga viene eseguita.

Nelle espressioni del tipo variabile = espressione, l'espressione a destra può a sua volta essere un'espressione con side-effect, in particolare contenente l'operatore =. Anche in tal caso, assegnazione e valutazione vengono effettuate considerando prima l'espressione a destra.

*Esempio:*

```
int x = 0, y = 0;
printf("%d", y = 3 + (x = 7)); // stampa 10
printf("%d", x); // stampa 7
```

Per valutare l'espressione y = 3 + (x = 7) della seconda riga viene prima valutata l'espressione a destra 3 + (x = 7), per la quale occorre prima valutare x=7. Quest'ultima effettua side-effect assegnando il valore 7 ad x, quindi restituisce tale valore. Conseguentemente, l'espressione 3 + (x = 7) ha valore 10, che è quello che viene assegnato, per effetto dell'operatore = ad y.

Occorre prestare particolare attenzione all'uso degli operatori di uguaglianza (==) e assegnazione (=). Uno degli errori più comuni in C

è infatti quello di usare = al posto di == all'interno delle condizioni, ad esempio nell'istruzione if.

*Esempio:*

```
int x = 0;
if (x == 0)
  printf("x vale 0"); // stampa: x vale 0
if (x=0)
  printf("x vale 0"); // non stampa nulla
```

Nella seconda riga, l'espressione x == 0 restituisce un valore non nullo (quindi vero), in quanto x ha effettivamente valore 0, quindi la condizione è soddisfatta ed il ramo *then* viene eseguito.

Nella quarta riga, invece, l'espressione (con side-effect) x=0 assegna alla variabile x il valore 0, che è anche il valore restituito in quanto espressione (quindi falso). Conseguentemente, la condizione dell'if non è soddisfatta, ed il ramo *then* non viene eseguito.

Sebbene il linguaggio C consenta un uso indifferenziato dei due tipi di espressioni, noi useremo le espressioni con side-effect per formare delle istruzioni, evitandone sempre l'uso all'interno di espressioni matematiche e, in particolare, nelle condizioni.

*Esempio:* L'istruzione

```
x = 5 * (y = 7);
```

dovrebbe essere scritta come

```
y = 7;
x = 5 * y;
```

### 2.4.2. Operatori di assegnazione composta

Il seguente frammento di programma:

```
somma = somma + addendo;
salario = salario * aumento;
```

si può abbreviare in:

```
somma += addendo;
salario *= aumento;
```

In generale l'assegnazione:

```
variabile = variabile operatore espressione
```

si può abbreviare in:

```
variabile operatore= espressione
```

Per ciascun operatore +, -, *, /, %, esiste il corrispondente operatore +=, -=, *=, /=, %=.

### 2.4.3. Operatori di incremento e decremento

Per incrementare o decrementare una variabile intera di 1, alcune tra le più comuni operazioni presenti in un programma, il C mette a disposizione gli operatori ++ e - -, che possono essere usati come operatori prefissi (ad esempio nell'istruzione ++x;) o postfissi (ad esempio x++;).

*Esempio:*

```
int x = 1;
x++; // incrementa x di 1
printf("%d",x); // stampa 2
--x; // decrementa x di 1
printf("%d",x); // stampa 1
```

Dal punto di vista degli effetti sul valore della variabile a cui sono applicati, non c'è nessuna differenza tra l'uso prefisso o postfisso dello stesso operatore: l'esecuzione delle istruzioni x++; o ++x; comporta lo stesso effetto sulla variabile x, ovvero ne viene incrementato il valore di 1.

Tuttavia, tali operatori possono essere usati all'interno di espressioni. In tal caso, le due forme, pur comportando lo stesso effetto sulla variabile, danno luogo a valori diversi delle espressioni. In particolare:

- il valore dell'espressione ++x è pari al valore iniziale della variabile x incrementato di 1;

- il valore dell'espressione x++ è pari al valore iniziale della variabile x;

Un comportamento analogo si ha con l'operatore - -.

*Esempio:* Il seguente frammento di codice mostra il comportamento degli operatori ++ e - - usati nelle forme prefissa e postfissa all'interno di espressioni.

```
int x = 0;
printf("%d",x); // stampa 0
printf("%d",x++); // stampa 0
printf("%d",x); // stampa 1
printf("%d",x--); // stampa 1
printf("%d",x); // stampa 0
printf("%d",++x); // stampa 1
printf("%d",--x); // stampa 0
printf("%d",x); // stampa 0
```

Quando ++ è usato in forma prefissa, viene chiamato operatore di *pre-incremento* (in quanto incrementa la variabile prima di restituirne il valore). Se usato in forma postfissa è detto di *post-incremento*. Analogamente, l'operatore - - usato in forma prefissa è detto di *pre-decremento* e di *post-decremento* se usato in forma postfissa.

Si noti che le espressioni contenenti ++ e - - sono espressioni con side-effect che è preferibile non utilizzare, per evitare di introdurre errori difficili da rilevare e rendere più chiari i programmi.

## 2.5. Lettura e scrittura di espressioni numeriche

Lettura (da standard input) e scrittura (su standard output) di espressioni numeriche si effettuano, rispettivamente, mediante le funzioni scanf e printf, definite nel file di sistema stdio.h.

Le funzioni scanf e printf accettano come primo parametro una stringa di caratteri e come parametri successivi le espressioni di cui si vuole stampare il valore. Non c'è limite al numero di espressioni che si possono passare alle funzioni.

### 2.5.1. La funzione scanf

La funzione scanf può essere invocata usando la sintassi seguente:

```
scanf(stringa,par1,par2,...)
```

Il primo parametro, detto *specifica di formato*, è una stringa di caratteri che rappresenta il tipo di dato letto (intero, reale, ecc.)[3]. I parametri successivi sono espressioni della forma &nome-variabile che identificano le variabili in cui i dati letti devono essere memorizzati (il significato dell'operatore & sarà discusso più avanti).

*Esempio:* Il seguente frammento di codice memorizza nella variabile i il valore immesso da standard input (ovvero tastiera, se non diversamente specificato).

```
int i;
printf("Inserisci un valore intero\n");
scanf("%d",&i);
```

La stringa %d indica alla funzione scanf che il valore letto deve essere interpretato come un intero. Il parametro &i indica che tale valore deve essere memorizzato nella variabile i.

La specifica di formato inizia con il carattere % ed è seguita da una combinazione di caratteri che dipendono dal tipo di dato da memorizzare. La seguente tabella mostra le specifiche per i tipi più comuni.

| Tipo | Specifica di formato |
|---|---|
| int | d |
| unsigned int | u |
| short int | hd |
| long int | ld |
| float | f |
| double | lf |
| char | c |

È possibile leggere e memorizzare più valori con una sola invocazione di scanf.

*Esempio:*

I seguenti frammenti, in cui si usa la funzione scanf per leggere un valore intero ed uno reale, sono equivalenti per quanto riguarda i valori assegnati alle variabili.

[3] Questa stringa può in realtà essere più complessa, in quanto scanf non si limita a leggere semplici valori. Tuttavia questi aspetti avanzati non saranno considerati in questo corso.

**scanf.c (1)**

```
int i;
float f;
scanf("%d%f", &i, &f);
```

**scanf.c (2)**

```
int i;
float f;
scanf("%d", &i);
scanf("%f", &f);
```

In questo caso, l'associazione tra parametri e specifiche di formato avviene su base posizionale, ovvero al secondo parametro (&i) è associata la prima specifica (%d), al terzo parametro (&f) la seconda (%f) e così via.

### 2.5.2. La funzione printf

La funzione printf può essere invocata con una sintassi simile a quella della scanf:

```
printf(stringa,par1,par2,...)
```

Ci sono tuttavia due differenze principali rispetto alla scanf:

- La stringa di caratteri può contenere anche i caratteri che si desidera stampare.
- I parametri sono le espressioni di cui si vuole stampare il valore (non si deve anteporre il carattere &).

*Esempio:* Il seguente frammento mostra un esempio in cui i valori appena letti vengono stampati:

```
int i;
float f;
scanf("%d%f",&i,&f);
printf("i=%d\nf=%f\n",i,f);
```

La funzione printf stampa la stringa di caratteri passata come primo parametro, sostituendo alla prima specifica di formato (%d) il valore del

secondo parametro (ovvero dell'espressione i), alla seconda specifica di formato (%f), il valore del terzo parametro (f) e così via.

*Esempio:* L'esecuzione dell'esempio precedente con i parametri di input: 5 e 12.23 produce il seguente output:

```
i=5
f=12.230000
```

Se la specifica di formato ed il valore letto/stampato non sono conformi, il comportamento delle funzioni risulta non corretto (vedi esempio successivo).

*Esempio:* Si consideri il seguente frammento di codice:

```
int i;
scanf("%d",&i);
printf("%f",i);
```

L'esecuzione del codice produce in output un valore diverso da quello inserito, a causa dell'indicazione errata della specifica di formato. Si noti, comunque, che il compilatore ci informa, tramite un messaggio di *warning*, dell'uso scorretto del formato.

## 2.6. Lettura e scrittura di char

La lettura e la scrittura di valori di tipo char possono avvenire in maniera analoga a quanto visto per i tipi numerici, tramite le funzioni scanf e printf, usando la specifica di formato c, come mostrato nel seguente esempio.

*Esempio:*

```
char c;
printf("Inserisci un carattere\n");
scanf("%c",&c);
printf("Il carattere inserito e' %c\n", c);
```

### 2.6.1. Le funzioni getchar e putchar

Per la lettura e la scrittura di un singolo carattere, il C fornisce inoltre le funzioni getchar e putchar, rispettivamente. La funzione putchar stampa su schermo (o su file, se lo standard output è stato redirezionato con l'operatore >) il carattere passato come argomento. La funzione getchar legge un carattere da tastiera (o da file, se lo standard input è stato redirezionato con l'operatore <) e ne restituisce il rispettivo codice come int, che può successivamente essere assegnato ad un char. Si noti che sebbene il valore restituito sia di tipo int, poiché getchar restituisce il codice del carattere, esso può essere assegnato ad una variabile di tipo char senza che si verifichi perdita di informazione.

*Esempio:*

```
printf("inserisci un carattere\n");
char c = getchar(); // legge il carattere inserito
putchar(c); // stampa il carattere letto
```

Si osservi che nell'assegnazione della seconda riga si sta eseguendo una conversione del valore restituito da getchar, dal tipo int al tipo char della variabile c.

### 2.6.2. Sequenze di escape

Oltre ai caratteri stampabili comuni (`'a','b','c'`, etc.), il codice ASCII definisce anche un insieme di caratteri speciali, rappresentati da sequenze. Esse identificano sia caratteri utili nella formattazione dell'output, come ritorno a capo o tab, o caratteri che non potrebbero essere stampati se usati nella forma comune, ad esempio l'apice singolo `'`. Nella seguente tabella sono riportate le sequenze di escape più comuni in C.

| Nome | Sequenza | Significato |
|---|---|---|
| Alert | \a | emette il suono di alert |
| Backspace | \b | cancella l'ultimo carattere |
| New line | \n | torna a capo |
| Carriage return | \r | torna all'inizio della riga corrente |
| Tab orizzontale | \t | inserisce degli spazi (in genere 8) |
| Backslash | \\ | stampa il carattere \ |
| Apice singolo | \' | stampa il carattere ' |
| Apice doppio | \" | stampa il carattere " |

## 2.7. Esercizio: teorema di Pitagora

Scrivere un programma che riceva in input da tastiera il valore di due cateti di un triangolo rettangolo e restituisca il valore dell'ipotenusa.

Esempio d'uso:

```
Inserisci il valore del primo cateto
3
Inserisci il valore del secondo cateto
4
Il valore dell'ipotenusa e' 5.0
```

Soluzione:

**pitagora.c**

```c
#include<math.h> // Per funzioni sqrt() e pow()
#include<stdio.h> // Per scanf() e printf()

int main(){
    float c1, c2, ip;
    printf("Inserisci il valore del primo cateto\n");
    scanf("%f",&c1);
    printf("Inserisci il valore del secondo cateto\n")
    ;
    scanf("%f",&c2);
    ip = sqrt(pow(c1,2)+pow(c2,2));
    printf("Il valore dell'ipotenusa e'%f\n",ip);
}
```

## 2.8. Conversione di tipo

Il C permette di specificare espressioni aritmetiche e logiche che coinvolgono tipi primitivi diversi (incluso il tipo char).

Per valutare tali espressioni è necessario *convertire* i valori in maniera tale da eseguire le operazioni tra valori dello stesso tipo.

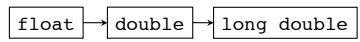
Il C esegue queste conversioni senza l'intervento del programmatore nei seguenti casi:

1. I tipi coinvolti in un'espressione logico-aritmetica sono diversi (ad esempio int e float);
2. Il tipo dell'espressione nel lato destro di un'assegnazione non coincide con quello del lato sinistro (ad esempio int x = 3.2;)
3. Il tipo del parametro attuale passato ad una funzione non coincide con il tipo del parametro formale corrispondente (v. seguito);
4. Il tipo dell'espressione di una return non coincide con il tipo di ritorno della funzione (v. seguito).

Normalmente, la conversione avviene in maniera tale da non perdere informazione, ovvero verso i tipi il cui dominio copre l'intervallo più grande.

Ad esempio nell'espressione 1 + 3.0, il valore 1 viene convertito da int a float prima di valutare l'espressione.

Per il caso 1, si consideri un'espressione del tipo *v1 op v2*, dove *op* denota un operatore aritmetico. In questo caso, se uno dei due operandi è di tipo floating point, tutti gli operandi vengono convertiti nel tipo tipo floating point a precisione maggiore.




*Esempio:*

**cast-implicito.c**

```
double d = 3.23199;
float f = 56.3419;
int i = 100;

printf("%lf\n", (d + i));
printf("%lf\n", (f + d));
```

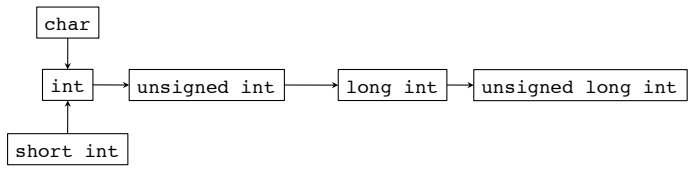
```
printf("%lf\n", (f + d + i));
```

L'espressione (d+i) nella quarta riga viene valutata convertendo prima il risultato dell'espressione intera i in double e successivamente sommando il valore ottenuto (100.000000) a quello dell'espressione double d, ottenendo come risultato il valore 103.231990.

Nell'espressione (f+d) è invece f ad essere convertita in double prima di eseguire la somma.

Nell'espressione complessa (f + d + i), viene convertito il valore di i in double, quindi viene valutata l'espressione (d + i) (di tipo double), successivamente viene convertito il valore di f in double e infine viene calcolato il valore dell'espressione completa (f+(d+i)).

Se i tipi coinvolti sono solo interi, eventualmente con rappresentazioni diverse, ad esempio char e unsigned, gli short ed i char, se presenti, vengono convertiti in int e successivamente viene applicata la conversione dell'operando con rappresentazione più piccola nel tipo di quello con rappresentazione più grande, secondo lo schema seguente:




*Esempio:*

**cast-implicito-int.c**

```
long int i = 100;
short int j = 8;
printf("%ld\n", (j-i));
```

L'espressione (i-j) viene valutata convertendo il valore (short) di i in int, poi da int in long int, e successivamente eseguendo la differenza. Nel caso in cui *op* sia un operatore logico (ad esempio &&), gli operandi vengono convertiti in int. Si noti che questa operazione può comportare perdita di precisione.

La seguente tabella descrive il tipo risultante di una espressione della forma a + b per ciascuna coppia di tipi possibili per a e b:

| a + b | char | short | int | long | float | double |
|---|---|---|---|---|---|---|
| char | int | int | int | long | float | double |
| short | int | int | int | long | float | double |
| int | int | int | int | long | float | double |
| long | long | long | long | long | float | double |
| float | float | float | float | float | float | double |
| double | double | double | double | double | double | double |

Per il caso 2, il valore dell'espressione nel lato destro dell'assegnazione viene convertito nel tipo della variabile a sinistra, e successivamente viene effettuata l'assegnazione.

Si noti che quando non si ha perdita di informazione, la conversione non può generare errori derivanti dal valore dell'espressione a destra dell'operatore di assegnazione.

*Esempio:*

```
int i = 10;
long int j = i;
float f = 100.23f;
double d = f;
```

Se si effettua una conversione da un tipo in virgola mobile ad uno intero, si ha il troncamento della parte decimale.

*Esempio:*

```
double d = 102.945;
long int i = d; // i vale 102
```

Negli altri casi (ad esempio da int a short), affinché la conversione non dia luogo a perdita d'informazione, occorre assicurarsi che il valore da convertire sia incluso nell'insieme dei valori rappresentabili nel tipo di destinazione.

*Esempio:*

**conversione-down.c**

```
int i = 32767;
short j = i; //OK: 32767 rappresentabile come short
printf("%d\n",j); // 32767
j = i+1; // NO: 32768 fuori intervallo short
printf("%d\n",j); // -32768 !!!
```

Per i casi 3 e 4, valgono essenzialmente le stesse regole del caso 2, ma l'espressione convertita viene assegnata al parametro della funzione o restituita dalla funzione.

## 2.9. Casting

Il C permette di convertire dati da un tipo ad un altro in maniera esplicita. Ad esempio, è possibile convertire il float 3.2 in int, ottenendo 3. Questa operazione è detta *cast* ed ha la seguente sintassi.

**Cast**

*Sintassi:*

```
(tipo) espressione;
```

- *tipo* è il nome di un tipo di dato;
- *espressione* è l'espressione di cui si vuole convertire il tipo in *tipo*

*Semantica:*
Converte il tipo di *espressione* in *tipo*.

L'operatore di cast ha precedenza più alta rispetto agli operatori aritmetici.

*Esempio:* Nel seguente esempio, a seconda dell'uso dell'operatore di cast, l'operatore di divisione / può identificare la divisione tra reali o tra interi.

**cast.c**

```
#include<stdio.h>

int main() {
```

```
    float x = 7.0f;
    float y = 2.8f;
    int a = x / y; // Divisione reale, risultato = 2.5
    printf("%d\n",a); // Stampa 2
    a = (int) x / (int) y; // Divisione intera,
    risultato = 3
    printf("%d\n",a); // Stampa 3
}
```

Nella terza riga, l'espressione x /y restituisce il valore reale 2.5, che viene poi troncato a 2 per completare l'assegnazione ad a. Nella quinta riga, invece, il casting viene effettuato *prima* che sia applicata la divisione, convertendo i valori delle espressioni x ed y rispettivamente in 7 e 2. Poichè gli operandi sono interi, la divisione viene interpretata come intera, e viene restituito pertanto il valore 3.

Si noti che, in questo esempio, la conversione comporta *perdita di informazione*, dovuta al troncamento.

ATTENZIONE: le variabili x e y rimangono di tipo float! È solo il valore restituito dalle espressioni ad essere convertito.

In generale è possible eseguire il casting da e verso qualunque tipo. Le regole del casting sono le stesse discusse per le conversioni in presenza di assegnazione.

*Esempio:*

```
double d; float f; long l; int i; short s;

/* Le seguenti assegnazioni sono corrette
 (e non comportano perdita di informazione)*/

d=(double)f; l=(long)i; i=(int)s;

/* Le seguenti assegnazioni sono corrette
 (ma possono comportare perdita di informazione) */

f=(float)d; l=(long)f; i=(int)l;
s=(short)i; f=(float)l;
```

Nonostante vi siano molte occasioni in cui la conversione di tipo avviene in maniera implicita e non vi sia necessità di eseguire il casting, è sempre opportuno farlo per migliorare la leggibilità del codice.

*Esempio:*

```
float f = 23.45f;
int a = f;        // NO
int a = (int) f; // SI
```

## 2.10. L'operatore sizeof

L'operatore sizeof, applicato ad un tipo, restituisce la quantità di memoria, in byte, necessaria a memorizzare un valore del tipo specificato come parametro.

*Esempio:*

```
int i = sizeof(char); // i vale 1 (byte)
int j = sizeof(int);  // j vale 4
```

Il valore restituito da sizeof è di tipo intero senza segno.

Come già evidenziato, la dimensione dei tipi di dato primitivi dipende dal compilatore e dal sistema operativo usato. L'operatore sizeof consente di verificare la dimensione dei tipi primitivi durante l'esecuzione del programma e di comportarsi conseguentemente.

sizeof può anche essere applicato ad espressioni (incluse costanti e variabili). In tal caso restituisce la dimensione associata al tipo del valore dell'espressione.

*Esempio:*

```
int i = 10;
int j = sizeof(i); // j vale 4
j = sizeof(i+2);   // j vale 4
j = sizeof(i+2.0); // j vale 8 (i+2.0 e' double)
```

## 2.11. Definizione di nuovi tipi

In C è possibile definire nuovi tipi mediante l'istruzione typedef.

*Esempio:*

```
typedef int TipoA;
typedef int TipoB;
typedef int Bool;

TipoA a = 0;
TipoB b = 1;
Bool c = a + b;
```

I tipi definiti sono sempre compatibili con il tipo base, quindi nell'esempio precedente l'istruzione c=a+b; è valida e ritorna un risultato di tipo int.

Il vantaggio principale nell'uso di typedef consiste in una migliore manutenibiltà del programma.

*Esempio:*

```
typedef int valuta;
valuta stipendio = 20;
```

Se la valuta cambia da lira ad euro (quindi da intero a reale), è sufficiente rimpiazzare la prima riga con la seguente definizione di tipo:

```
typedef float valuta;
```

# 3. Puntatori

## 3.1. Memoria, indirizzi e puntatori

La memoria di un elaboratore è organizzata in celle contigue, tipicamente da 1 byte, ciascuna con un proprio indirizzo [1].

| indirizzo | contenuto |
|---|---|
| 0xF0000000 | 00100010 |
| 0xF0000001 | 00010000 |
| 0xF0000002 | 11111110 |
| 0xF0000003 | 11011010 |
| ... | ... |

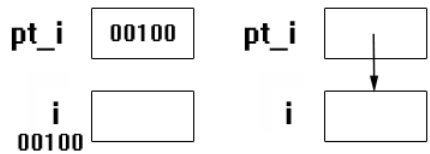
Una variabile identifica un certo numero di celle contigue, dipendente dal tipo di dato che memorizza (ad es., 1 cella per char, 4 celle per int). Accedendo ad una variabile, accediamo al contenuto delle celle identificate dalla variabile. In C il programmatore ha la possibilità di gestire gli indirizzi attraverso delle variabili che vengono definite di tipo *puntatore*.

I valori delle variabili di tipo puntatore sono *indirizzi di memoria*, cioè valori numerici che fanno riferimento a specifiche locazioni di memoria. Normalmente non interessa conoscere lo specifico indirizzo contenuto in una variabile di tipo puntatore, mentre è molto importante conoscere a quali variabili il puntatore fa riferimento ed il valore in essa contenuto.

---

[1] Gli indirizzi di una macchina a 32 bit vengono rappresentati per brevità con 8 cifre esadecimali, ciascuna rappresentativa del gruppo di 4 bit corrispondente alla sua rappresentazione in sistema binario. Il prefisso 0x indica che l'indirizzo riportato è rappresentato, appunto, nel sistema esadecimale.

È quindi sufficiente rappresentare graficamente l'indirizzamento usando una freccia.




### 3.1.1. Operatore *indirizzo-di*

Vediamo innanzitutto come ottenere dei valori di tipo puntatore, cioè degli indirizzi.

**stampa-puntatore.c**

```
int i = 15;
printf("L'indirizzo di i e' %p\n", &i);
printf("mentre il valore di i e' %d\n", i);
```

stampa

```
L'indirizzo di i e' 0xbffff47c
mentre il valore di i e' 15
```

L'operatore & si chiama operatore *indirizzo-di* e restituisce l'indirizzo della prima cella di memoria del gruppo identificato dalla variabile a cui viene applicato.

NOTA: Per la specifica di formato dei puntatori si usa il carattere p, indipendentemente dal tipo di dato a cui il puntatore fa riferimento.

### 3.1.2. Operatore di indirizzamento indiretto

L'operatore di indirizzamento indiretto * (si può chiamare anche operatore di *dereferenziamento* o *contenuto-di*) permette di accedere al valore contenuto nella locazione di memoria identificata da un certo indirizzo.

*Esempio:* Si consideri il seguente frammento di codice

```
int j = 1;
int i = *&j;
```

L'espressione &j restituisce l'indirizzo della locazione puntata da j. L'operatore * restituisce, invece, il valore contenuto nella locazione di memoria puntata dal suo argomento (cioè il risultato dell'espressione &j). Pertanto, l'istruzione i = *&j; equivale all'istruzione i = j;.

L'operatore di indirizzamento indiretto * si può usare anche nella parte sinistra dell'istruzione di assegnazione e in questo caso consente l'assegnazione del risultato dell'espressione a destra dell'operatore = nella zona di memoria puntata dal puntatore.
NOTA: l'operatore (unario) di indirizzamento indiretto * non deve essere confuso con l'operatore di moltiplicazione (binario).

### 3.1.3. Variabili di tipo puntatore

Per memorizzare gli indirizzi occorre dichiarare delle variabili di tipo *puntatore*. Nella dichiarazione è necessario specificare che tipo di dato è contenuto nella locazione puntata.

*Esempio:* L'istruzione

```
int *p1;
```

dichiara p1 come variabile di tipo puntatore, specificando che il tipo contenuto nella locazione puntata è intero.
ATTENZIONE: La dichiarazione di una variabile puntatore non alloca memoria per la variabile puntata.

*Esempio:*

```
int *p1;
*p1 = 10; // ERRORE: l'indirizzo contenuto in p1
          // non corrisponde a memoria allocata
```

Per accedere ad una locazione di memoria, è necessario che essa sia allocata. In caso contrario viene generato un errore a tempo di esecuzione.

Come visto, una dichiarazione di variabile alloca memoria.

*Esempio:*

```
int i;  // Alloca memoria per un intero
int *p; // Alloca memoria per la variabile puntatore
p = &i; // assegna a p l'indirizzo della locazione
        // associata ad i
*p = 10; // OK
printf("i = %d\n", i); // Stampa 10.
```

ATTENZIONE alle dichiarazioni multiple!

*Esempio:*

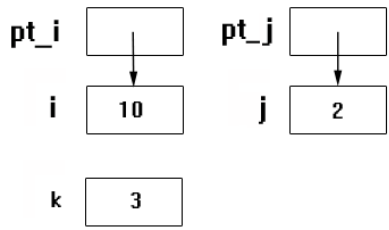
```
int *p1, p2;
```

p1 è di tipo puntatore a intero, mentre p2 è di tipo intero!

### 3.1.4. Esempio: uso di variabili puntatore

```
int i,j,k;
int *pt_i, *pt_j, *pt_k;
pt_i = &i;
pt_j = &j;
pt_k = &k;
*pt_i = 1;
*pt_j = 2;
*pt_k = *pt_i + *pt_j;
*pt_i = 10;
printf("i = %d\n", *pt_i);
printf("j = %d\n", *pt_j);
printf("k = %d\n", *pt_k);
printf("i = %d\n", i);
printf("j = %d\n", j);
printf("k = %d\n", k);
```

In questo frammento di programma, le variabili di tipo int i,j,k, vengono utilizzate tramite puntatori alle locazioni di memoria ad esse associate al momento della dichiarazione.




Il programma stampa:

```
i = 10
j = 2
k = 3
i = 10
j = 2
k = 3
```

### 3.1.5. Condivisione di memoria

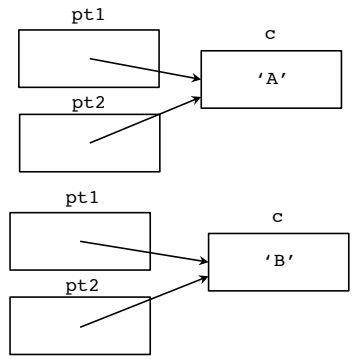
Un'area di memoria a cui fanno riferimento due o più puntatori è detta *condivisa*. In questo caso, le modifiche effettuate tramite un puntatore sono visibili tramite l'altro (e viceversa).

*Esempio:*

**condivisione.c**

```
char c = 'A';
char* pt1 = &c;
char* pt2 = pt1;
printf("*pt1 = %c\n",*pt1);
printf("*pt2 = %c\n",*pt2);
*pt2 = 'B'; // Modifica tramite pt2
printf("%c\n",c);// Ovviamente, c e' cambiata!
printf("*pt1 = %c\n",*pt1); // anche pt1 vede la
    modifica
```

Le figure seguenti mostrano lo stato della memoria prima e dopo la modifica effettuata tramite pt2.




### 3.1.6. Assegnazione

A variabili di tipo puntatore possono essere assegnati valori corrispondenti ad indirizzi di memoria.

*Esempio:*

**assegnazione-puntatore.c**

```
int* p;
int q = 10;
char c = 'x';
p = &q; // OK
p = &c; // WARNING! p punta a int mentre c e' char
```

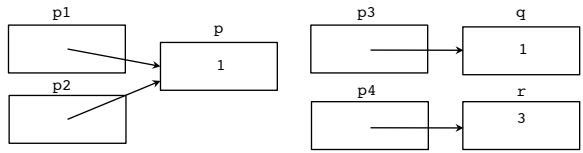
NOTA: L'assegnazione tra variabili puntatori e indirizzi di variabili di tipo non compatibile (come nell'esempio) provoca tipicamente solo un warning con il compilatore C, mentre è considerato un errore dal compilatore C++.

### 3.1.7. Uguaglianza tra puntatori

Quando si usa l'operatore di uguaglianza (==) tra puntatori occorre prestare particolare attenzione. L'operatore, infatti, verifica l'uguaglianza tra gli indirizzi contenuti nelle variabili puntatore, non tra il contenuto delle variabili puntate.

*Esempio:*

```
int p=1, q=1, r=3;
int *p1=&p, *p2=&p, *p3=&q, *p4=&r;
if(p1 == p2)  // TRUE: p1 e p2 puntano alla stessa
    variabile
if(*p1 == *p2) // TRUE: variabili puntate da p1 e p2
    hanno stesso valore
if(p2 == p3)  // FALSE: p2 e p3 puntano a variabili
    diverse
if(*p2 == *p3)  // TRUE: variabili puntate da p2 e p3
    hanno stesso valore
if(p3 == p4)  // FALSE: p3 e p4 puntano a variabili
    diverse
if(*p3 == *p4)  // FALSE: variabili puntate da p3 e p4
     hanno valori diversi
```




## 3.2. Aritmetica dei puntatori

Alle variabili di tipo puntatore è possibile aggiungere o sottrarre valori interi. Tali operazioni hanno una *semantica diversa* rispetto al caso degli interi. Inoltre, queste due operazioni, insieme alla differenza tra puntatori (discussa più avanti) rappresentano le uniche operazioni aritmetiche disponibile sui puntatori.

### 3.2.1. Somma di un valore intero ad un puntatore

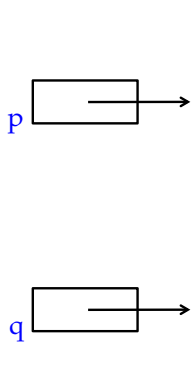
*Esempio:* Si consideri il seguente esempio

```
int i = 10;
int* p = &i;
int* q = p + 2;  // indirizzo di p + 2 * sizeof(int)
```

Al termine dell'esecuzione, a q viene assegnato l'indirizzo della locazione di memoria ottenuta valutando l'espressione p + 2 come segue:

- si considera l'indirizzo a cui p punta (&i);
- si considera la dimensione $N$ in byte del tipo a cui la variabile p punta (4 byte per int);

- si moltiplica il valore sommato a p, cioè 2, per $N$;
- si restituisce il valore dell'indirizzo puntato da p, incrementato del valore ottenuto sopra.




| indirizzo | contenuto | |
|---|---|---|
| 0x00AF0000 | 00100010 | |
| 0x00AF0001 | 00001010 | Celle allocate per i (4 byte) |
| 0x00AF0002 | 00000000 | |
| 0x00AF0003 | 00000000 | |
| 0x00AF0004 | 00000000 | |
| ... | ... | |
| 0x00AF0009 | 11011111 | |
| ... | ... | |

Si noti che $N$ dipende dal tipo di p, non di q.

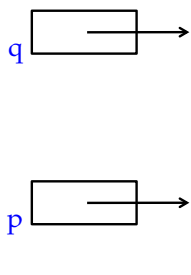
ATTENZIONE: la locazione a cui punta q dopo l'assegnazione potrebbe non essere valida (cioè non allocata).

### 3.2.2. Sottrazione di un valore intero da un puntatore

Il caso della sottrazione di un valore intero da un puntatore è duale rispetto a quanto visto per la somma, con l'indirizzo di partenza decrementato del valore intero moltiplicato per $N$.

*Esempio:*

```
char c = 'd';
char *p = &c;
char *q = p - 4; // indirizzo di p - 4 * sizeof(char)
```




| indirizzo | contenuto | |
|---|---|---|
| 0x1BA01004 | 00101110 | |
| 0x1BA01005 | 00000000 | |
| 0x1BA01006 | 10100010 | |
| 0x1BA01007 | 00011010 | |
| 0x1BA01008 | 00001010 | |
| 0x1BA01009 | 01000000 | 1 byte allocato per c |
| ... | ... | |

## 3.3. Puntatori a costanti

La specifica const può essere applicata anche a variabili di tipo puntatore e specifica che il contenuto della *locazione puntata* è costante (cioè non può essere modificato).

*Esempio:*

```
double pi = 3.5;
const double *pt = &pi;
(*pt)++; // ERRORE *pt e' costante
pt++;    // OK: pt non e' costante
```

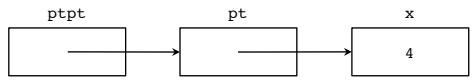
## 3.4. Puntatori a puntatori

Come per ogni altro tipo si può definire un puntatore ad una variabile di tipo puntatore.

*Esempio:* Il seguente frammento di codice

```
int x;
int *pt;
int **ptpt;

x = 4;
pt = &x;
ptpt = &pt;
printf("%d\n", **ptpt);
```

stampa il valore di x, cioè 4.




## 3.5. Il valore NULL

Le variabili di tipo puntatore possono assumere anche il valore speciale NULL. Tale valore specifica che la variabile non punta ad alcuna locazione di memoria valida. Si noti la differenza tra una variabile (puntatore) il cui valore è NULL, che risulta pertanto inizializzata, ed un variabile non inizializzata, il cui valore non è noto. Il confronto con NULL può essere usato per verificare se una variabile di tipo puntatore punta ad una locazione di memoria valida o meno.

*Esempio:*

```
int *pt = NULL;
...
if (pt!=NULL)
    *pt=10;
```

In questo caso il ramo if non viene eseguito. L'istruzione *pt=10; eseguita al momento in cui pt vale NULL genera un errore a tempo di esecuzione, in quanto non è possibile accedere ad un'area di memoria valida se il valore del puntatore è NULL.

## 3.6. Il tipo void*

Poiché la dimensione della memoria allocata per i puntatori è fissa (rappresenta un indirizzo) si può omettere la specifica del tipo della variabile puntata, dichiarando un puntatore di tipo void*.

*Esempio:*

```
void* pt;
int i;
pt = &i;
```

Il tipo void* è compatibile con qualsiasi altro tipo puntatore.

I puntatori di tipo void sono utilizzati quando il tipo dei valori da trattare non è noto a priori. Un caso tipico si ha nell'uso delle istruzioni per l'allocazione dinamica di memoria (v. seguito). Si osservi che il contenuto della locazione puntata da un puntatore void* non può essere assegnato direttamente ad un'altra variabile (di qualsiasi tipo).

*Esempio:*

```
void* pt;
...
int j = *pt; // ERRORE!
```

Per poter effettuare questa assegnazione, è necessario prima eseguire una conversione.

## 3.7. Conversione di puntatori

Anche le variabili di tipo puntatore possono essere convertite (automaticamente o mediante casting).

*Esempio:*

```
void* pt;
...
int* pti = pt; // Conversione automatica
int* pti2 = (int*) pt; // Casting esplicito
```

Dopo la conversione è possibile assegnare il contenuto della locazione puntata ad una variabile (di tipo opportuno).

```
int j = *pti2; // pti2 e' un puntatore a int
```

... o più brevemente:

```
int j = *((int *)pt);// ((int *)pt) e' un puntatore a
    int
```

Si noti che l'espressione ((int *)pt) restituisce l'indirizzo della locazione puntata da pt, convertita in riferimento a valore di tipo intero. L'uso dell'operatore di dereferenziamento permette quindi di accedere a tale valore.

## 3.8. Allocazione dinamica della memoria

L'*allocazione dinamica* della memoria è realizzata da opportune funzioni che gestiscono un'area di memoria chiamata *heap*. In quest'area, infatti,(a differenza dello stack (v.seguito), la allocazione/deallocazione della memoria ha luogo solo su esplicita richiesta del programma.

### 3.8.1. Funzione malloc

La funzione malloc, definita nella libreria stdlib.h, permette al programmatore di allocare dinamicamente spazio di memoria.

**Invocazione malloc**

*Sintassi:*

```
malloc(n);
```

- *n* è la dimensione in byte della memoria da allocare;

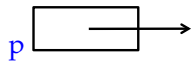
*Semantica:*
L'invocazione malloc(n):

- alloca *n* byte contigui di memoria;
- restituisce un puntatore di tipo void*, contenente l'indirizzo della prima cella allocata.

NOTA: Per accedere correttamente alla memoria allocata, il puntatore restituito deve essere convertito nel tipo opportuno.

*Esempio:*

```
#include<stdlib.h>
...
char* p = (char*) malloc(sizeof(char));
...
```




| indirizzo | contenuto | |
|---|---|---|
| 0x014F2100 | 00100010 | |
| 0x014F2101 | 00010000 | allocata da malloc (1 byte) |
| 0x014F2102 | 11001110 | |
| 0x014F2103 | 11010010 | |
| 0x014F2104 | 11010010 | |
| ... | ... | |

Tipicamente la dimensione della memoria da allocare dinamicamente è calcolata con un espressione del tipo

```
n * sizeof(tipo)
```

specificando quindi il numero e il tipo delle informazioni da allocare.

*Esempio:*

```
int *a = (int *)malloc(10*sizeof(int));
```

alloca memoria per 10 interi.

### 3.8.2. Recupero della memoria

In generale, esistono diversi meccanismi che consentono di recuperare la memoria allocata dinamicamente quando questa non è più necessaria al programma:

*Garbage collection:* la memoria viene recuperata, ovvero rilasciata, da una speciale funzione (*garbage collector*) che, eseguita periodicamente senza l'intervento del programmatore, si occupa di identificare le aree di memoria allocate per le quali non sia disponibile un riferimento (puntatore) all'interno nel programma. Tale meccanismo non è disponibile in C (ma lo è, ad esempio, in Python o in Java).

*Deallocazione esplicita:* la memoria deve essere rilasciata esplicitamente dal programma tramite l'invocazione di una funzione specifica.

Si noti che l'uso del garbage collector riduce le responasabilità del programmatore nella gestione della memoria al costo di una minore efficienza nell'esecuzione dei programmi.

#### 3.8.2.1. Deallocazione

Il C delega al programmatore il compito di rilasciare la memoria inutilizzata. *R*ilasciare o *d*eallocare un'area di memoria significa renderla disponibile per usi futuri, in particolare, ad esempio, per l'allocazione dinamica di nuove variabili.

La funzione preposta a questo scopo ha la seguente intestazione:

```
void free(void* p)
```

Quando invocata con parametro un puntatore ad un'area di memoria allocata dinamicamente, free si occupa di rilasciare l'area precedentemente allocata. È necessario che il puntatore passato alla funzione free sia stato restituito da una funzione di allocazione dinamica (malloc,

realloc o calloc –v. seguito) oppure sia il puntatore NULL, caso in cui la funzione non ha effetto. In tutti gli altri casi, free ha un comportamento indefinito.

*Esempio:* Si consideri il seguente frammento di codice:

**free.c**

```
int *p = (int *) malloc(sizeof(int));
*p = 0;
int *q = (int *) malloc(sizeof(int));
* q = 10;
free (p);
int* r = (int *) malloc(sizeof(int));
```

Notiamo che l'indirizzo assegnato a q è sicuramente diverso da quello memorizzato in p, in quanto la memoria a cui p fa riferimento è già allocata e non può esserlo nuovamente. Dopo l'esecuzione di free(p), invece, la memoria a cui p faceva riferimento è diventata libera e può quindi essere riutilizzata per nuove allocazioni; ad esempio, potrebbe (ma non deve necessariamente) essere riusata nell'ultima invocazione di malloc, caso in cui il puntatore r finirebbe per contenere lo stesso valore di p.

Si osservi che dopo l'esecuzione di free(p) il valore di p rimane inalterato. In altre parole, p continua a puntare alla stessa locazione di memoria cui puntava inizialmente, sebbene questa sia stata rilasciata (e potenzialmente riallocata).

Chiaramente, l'accesso ad un'area di memoria deallocata deve essere evitato, in quanto tale area non fornisce nessuna garanzia sui dati che essa contiene.

*Esempio:* Si completi l'esempio precedente con il seguente frammento di codice:

```
*r = 100;
printf("%d\n",*p);
```

Poiché l'indirizzo contenuto in p è rimasto inalterato, p continua a puntare allo stesso indirizzo cui puntava prima dell'invocazione a free. Pertanto, se la successiva invocazione a malloc non ha riutilizzato la memoria rilasciata, la stampa di *p produce ancora 0. Tuttavia, l'indirizzo puntato da p è rimasto disponibile per nuove allocazioni. In particolare esso potrebbe essere stato usato per allocare la variabile puntata da r

che, a seguito dell'assegnazione *r = 100 contiene ora il valore 100. In tal caso, ovviamente, la stampa di *p produrrebbe 100.

Nell'esempio precedente, il valore assunto dalla variabile puntata da p è essenzialmente arbitrario, in quanto dipendente dalle scelte effettuate dal sistema e non dal programmatore. Nella pratica esistono situazioni in cui l'accesso ad una stessa variabile tramite puntatori diversi corrisponde ad una precisa scelta. In questi casi é necessario indicare esplicitamente questa volontà (e non sperare che il sistema si comporti come desiderato). Nel caso in esame, avremmo potuto esplicitamente assegnare a p il valore di r, tramite l'istruzione: p = r.

#### 3.8.2.2. Puntatori *appesi*

Il problema dei *puntatori appesi* (*dangling pointers*) si verifica quando un puntatore si trova a fare riferimento ad un'area di memoria non allocata. Ciò può avvenire in maniera diretta, ovvero quando la memoria puntata da un puntatore viene deallocata tramite il puntatore stesso e l'indirizzo cui il puntatore fa riferimento non viene cambiato, oppure in maniera indiretta, per effetto della deallocazione di memoria a partire da un altro puntatore. Mostriamo di seguito un esempio del secondo caso.

*Esempio:*

**dangling-pointer.c**

```
int* q = (int *) malloc(sizeof(int));
int* p = q;
*q = 10;
free (p);
printf("%d\n",*q);
```

Per effetto del rilascio della memoria puntata dalla variabile p, la variabile q fa riferimento ad un'area di memoria non allocata, quindi suscettibile di modifiche arbitrarie.

Situazioni di questo genere devono essere evitate, in quanto fonti di errori difficili da individuare. Un approccio possibile, mostrato nel seguente esempio, consiste nell'assegnare il valore NULL ad un puntatore ogni volta che la memoria cui esso fa riferimento viene rilasciata (direttamente o indirettamente) e nel verificare che un puntatore faccia riferimento ad un indirizzo non NULL prima accedere alla locazione da esso puntata.

*Esempio:*

**fixed-dangling-pointer.c**

```
int* q = (int *) malloc(sizeof(int));
int* p = q;
*q = 10;
free (p);
p = NULL;
q = NULL;
// ...
if (q != NULL)
    printf("%d\n",*q);
//...
```

Omettere il rilascio della memoria non più utilizzata può comportare la saturazione della memoria disponibile, come mostrato nell'esempio seguente.

*Esempio:*

**finisci-memoria.c**

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
  long int s = 0;
  long * p;
  while (1) {
    p = (long *) malloc(1000000); // 1 MiB
    *p = 10l;
    s++;
    printf("Allocati %d MiB\n", s);
  }
}
```

### 3.8.3. Tempo di vita delle variabili allocate dinamicamente

Il *tempo di vita* delle variabili allocate dinamicamente corrisponde al periodo che intercorre tra l'allocazione e la deallocazione della variabile. Se una variabile allocata dinamicamente non viene deallocata esplicitamente, il suo tempo di vita termina con il programma. Pertanto, il tempo di vita di una variabile allocata dinamicamente è noto solo a tempo di esecuzione.

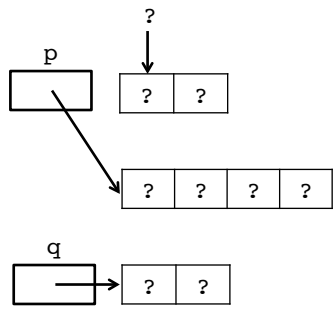
Nonostante una variabile possa rimanere in vita per l'intera durata di un programma, si può verificare il caso in cui la corrispondente memoria

non sia più accessibile. Questo accade quando non ci sono variabili di tipo puntatore che fanno riferimento a quell'area di memoria. Quando ciò si verifica, non vi è alcun modo di recuperare il riferimento ed accedere nudamente all'area di memoria: essa rimarrà inaccessibile per tutta la durata del programma ma, poiché ancora allocata, inutilizzabile per allocare nuova memoria. Tali situazioni sono ovviamente da evitare.

*Esempio:* Nel seguente frammento di codice, dopo la seconda linea, viene perso il riferimento alla prima area allocata, che rimane inutilmente allocata per tutta la durata dell'esecuzione del programma.

**riferimento-perso.c**

```
void* p = malloc(2);
p = malloc(4);
void* q = malloc(2);
```




Si osservi che oltre ad essere non più utilizzabile, la prima area di memoria, poichè già allocata, non può essere nemmeno sfruttata per allocare una nuova area.

## 3.9. Lettura tramite puntatori

I parametri di scanf, a partire dal secondo, denotano indirizzi di memoria in cui memorizzare i valori letti. Come visto, data una variabile x, il rispettivo indirizzo di memoria può essere ottenuto tramite l'operatore &.

*Esempio:*

```
int x;
scanf("%d",&x);// &x indirizzo della locazione
               // contenente il valore inserito
```

Quando il riferimento è memorizzato in una variabile puntatore, può essere usato direttamente.

*Esempio:*

**scanf.c**

```
int* p = (int*) malloc(sizeof(int));
printf("Inserisci un valore intero\n");
scanf("%d",p);
printf("Il valore inserito e': %d\n", *p);
```

# 4. Funzioni

## 4.1. Astrazione sulle operazioni: funzioni

L'astrazione sulle operazioni è supportata da tutti i linguaggi di programmazione attuali (Java, C#, C++, C, Pascal, Fortran, Lisp, ecc.). In C l'astrazione sulle operazioni si realizza attraverso la nozione di *funzione*. Una funzione può essere vista come una scatola nera che prende dei parametri in ingresso e restituisce dei risultati o compie delle azioni.

## 4.2. Definizione di funzioni

**Definizione di una funzione**

*Sintassi:*

```
intestazione
{
    istruzioni
}
```

- *intestazione* ha la seguente forma:

  ```
  tipoRisultato nomeFunzione(parametriFormali)
  ```

  dove

  - *tipoRisultato* è il tipo del risultato restituito dalla funzione oppure void se non viene restituito alcun risultato
  - *nomeFunzione* è il nome della funzione
  - *parametriFormali* è una lista di dichiarazioni di parametri (tipo e nome) separate da virgola; ogni parametro è una variabile; la lista di parametri può anche essere vuota.

- *istruzioni* sono le istruzioni che saranno eseguite all'invocazione della funzione stessa.

*Semantica:*
Definisce una funzione specificandone intestazione e corpo.

- L'intestazione indica il nome della funzione, il numero e il tipo dei parametri formali e il tipo dell'eventuale valore restituito.
- Il corpo della funzione specifica le istruzioni che devono essere eseguite quando essa viene attivata.
- I parametri formali servono a passare informazioni da elaborare nel corpo della funzione. I parametri formali vengono utilizzati nel corpo della funzione esattamente come variabili già inizializzate (l'inizializzazione avviene all'atto dell'invocazione della funzione, assegnando ai parametri formali il valore dei parametri attuali - vedi dopo).
- L'eventuale risultato restituito è il valore dell'invocazione della funzione. Se la funzione non restituisce risultati, allora essa viene utilizzata per effettuare operazioni (ed il tipo del risultato è void).

*Esempio:*

La funzione main ha la forma

```
int main () {
  ...
}
```

### 4.2.1. Risultato di una funzione: l'istruzione return

**Istruzione return**

*Sintassi:*

```
return espressione;
```

dove

- *espressione* è un'espressione il cui valore è compatibile con il tipo del risultato dichiarato nell'intestazione della funzione.

*Semantica:*
L'istruzione return eseguita all'interno di una funzione termina l'esecuzione della stessa e restituisce il risultato alla parte di programma dove è stata invocata la funzione.

*Esempio:*

```
int main() {
    return 0;  // terminazione normale del programma
}
```

Se il tipo del risultato della funzione è void, l'istruzione return si può omettere, oppure può essere usata semplicemente per interrompere l'esecuzione della funzione stessa. Dato che non si deve restituire un risultato, la sintassi dell'istruzione è in questo caso:

```
return;
```

L'esecuzione dell'istruzione return fa sempre terminare la funzione, anche se ci sono altre istruzioni che seguono.

### 4.2.2. Esempi di definizione di funzioni

*Esempio:* Funzione di stampa

```
void stampaSaluto() {
  printf("Buon giorno!\n");
}
```

La funzione stampaSaluto non ha parametri formali e non restituisce un risultato. Il suo corpo è costituito da una sola istruzione che stampa sullo schermo la stringa "Buon giorno!"

*Esempio:* Funzione per calcolo matematico

```
double distanza(double x1, double y1, double x2,
    double y2)
{
    return sqrt(pow(x2-x1,2)+pow(y2-y1,2));
}
```

La funzione distanza ha 4 parametri formali di tipo double e restituisce un valore dello stesso tipo. In particolare, calcola e restituisce la distanza tra i punti (x1,y1) e (x2,y2).

*Esempio:* Potenza tramite definizione di due funzioni

```
int moltiplicazione(int moltiplicando,
                    int moltiplicatore) {
  int prodotto = 0;
  while (moltiplicatore > 0) {
    moltiplicatore--;
    prodotto = prodotto + moltiplicando;
  }
  return prodotto;
}
```

```
int potenza(int base, int esponente) {
  int risultato = 1;
  while (esponente > 0) {
    esponente--;
    risultato = moltiplicazione(risultato, base);
  }
  return risultato;
}
```

Si noti che in questo caso il ciclo più interno del doppio ciclo è racchiuso nell'invocazione della funzione moltiplicazione.

### 4.2.3. Funzioni: istruzioni o espressioni?

```
double i = distanza(2,3,3,2);
distanza(2,3,3,2);
stampasaluto();
```

In C le espressioni possono essere considerate a tutti gli effetti istruzioni. Ma è buona pratica di programmazione mantenere la distinzione per aumentare la leggibilità del programma.

### 4.2.4. Segnatura di una funzione

La **segnatura** di una funzione consiste nel nome della funzione e nella descrizione (tipo, numero e posizione) dei suoi parametri.

*Esempi:*

- sqrt(double x)
- pow(double b, int e)

*Nota:* il nome del parametro non è significativo nella segnatura.

Il C non consente l'*overloading* (sovraccarico), in quanto non si possono definire funzioni con lo stesso nome e diversa segnatura.

*Esempio:*

```
int somma(int x, int y) {
    return x+y;
}

int somma(double x, double y) {
    return (int)(x+y);
}

int somma(int x, int y, int z) {
    return x+y+z;
}
```

Le tre definizioni non sono compatibili in C.

Se si usa il compilatore C++ (g++) vengono invece compilate!

### 4.2.5. Dichiarazione delle funzioni

Abbiamo già visto che in C si applica la regola di *dichiarazione prima dell'uso*. Per le funzioni il compilatore C, tenta di attribuire un tipo agli argomenti di una funzione se questa non è stata precedentemente dichiarata. Tuttavia, è buona prassi far precedere l'intestazione della funzione al suo uso.

**promo-arg.c**

```
#include <stdio.h>

int somma(int x, int y);

int main(){
    int i = 1;
    double d = 1.0;
    printf("int somma(int x, int y) %d", somma(i,i));
}

int somma(int x, int y) {
    return x+y;
}
```

Se si omette la dichiarazione della intestazione della funzione vengono generati dei messaggi di **warning**, ma il programma viene compilato e può essere eseguito. Tuttavia in alcune situazioni, si possono generare degli errori dovute alle conversioni di tipo che vengono fatte in modo automatico e si raccomanda di inserire sempre la dichiarazione della funzione.

#### 4.2.5.1. Organizzazione del codice

L'applicazione delle regole per la dichiarazione di funzioni suggerisce una strutturazione del codice nelle seguenti sezioni:

- direttive del compilatore (comandi che iniziano con il carattere #, per il momento #include);
- dichiarazioni di tipi;
- dichiarazioni di variabili;
- intestazioni di funzioni;
- programma principale main;
- definizione delle funzioni;

## 4.3. Passaggio dei parametri

Come abbiamo detto, la definizione di una funzione presenta nell'intestazione una lista di **parametri formali**. Questi parametri sono utilizzati come variabili all'interno del corpo della funzione.

L'invocazione di una funzione contiene i parametri da utilizzare come argomenti della funzione stessa. Questi parametri sono detti **parametri attuali**, per distinguerli dai parametri formali presenti nell'intestazione della definizione della funzione.

Quando attraverso una invocazione **attiviamo** una funzione, dobbiamo **legare** i parametri attuali ai parametri formali. In generale esistono diversi modi possibili di effettuare questo legame.

In C/C++ occorre distinguere tre casi di passaggio dei parametri:

- passaggio per valore
- passaggio mediante puntatori
- passaggio per riferimento (solo in C++)

### 4.3.1. Passaggio di parametri per valore

Sia pa un parametro attuale presente nell'invocazione della funzione e pf il corrispondente parametro formale nella definizione della funzione: legare pa a pf per valore significa, al momento della attivazione della funzione:

1. valutare il parametro attuale pa (che in generale è un'espressione)
2. associare una locazione di memoria al parametro formale pf
3. inizializzare tale locazione con il valore di pa.

In altre parole, il parametro formale pf si comporta esattamente come una variabile allocata al momento della attivazione della funzione e inizializzata con il valore del parametro attuale pa.

Alla fine dell'esecuzione del corpo della funzione la memoria riservata per il parametro formale viene rilasciata e il suo valore si perde.

*Nota:* i valori delle variabili che compaiono nell'espressione pa non vengono quindi alterati dall'esecuzione della funzione.

*Esempio:* Consideriamo la definizione della seguente funzione

```
int raddoppia(int x) {
    x = x * 2;
    return x;
}

int main() {
    int a,b;
    a=5;
    b=raddoppia(a+1);
    printf("a = %d\n",a);
    printf("b = %d\n",b);
}
```

Analizziamo cosa succede quando viene eseguito il programma:

1. *vengono definite* due variabili intere a,b e a viene inizializzata al valore 5;

2. *vengono valutati i parametri attuali*: nel nostro caso il parametro attuale è l'espressione a+1 che ha come valore 6;

3. *viene individuata la funzione da eseguire* in base al numero e tipo dei parametri attuali, cercando la definizione di una funzione la cui segnatura sia conforme alla invocazione: il nome della funzione deve essere lo stesso e i parametri attuali devono corrispondere in numero e tipo ai parametri formali: nel nostro caso la funzione che cerchiamo deve avere la segnatura raddoppia(int);

4. *viene sospesa l'esecuzione dell'unità di programma chiamante*: nel nostro caso la funzione main;

5. *viene allocata la memoria* per i parametri formali (considerati come variabili) e variabili definite nella funzione (vedi dopo): nel nostro caso viene allocata la memoria per il parametro formale x;

6. *viene assegnato il valore dei parametri attuali ai parametri formali*: nel nostro caso il parametro formale x viene inizializzato con il valore 6;

7. *vengono eseguite le istruzioni del corpo della funzione invocata* (a partire dalla prima): nel nostro caso il valore di x viene moltiplicato per 2 e diventa 12;

8. *l'esecuzione della funzione invocata termina* (o per l'esecuzione dell'istruzione return o perché non ci sono altre istruzioni da eseguire): nel nostro caso incontriamo l'istruzione return x;

9. *viene deallocata la memoria* utilizzata per i parametri formali e le variabili della funzione, perdendo qualsiasi informazione ivi contenuta: nel nostro caso viene rilasciata la locazione di memoria corrispondente al parametro formale x;

10. *se la funzione restituisce un risultato*, tale risultato diviene il valore dell'espressione costituita dall'invocazione nell'unità di programma chiamante: nel nostro caso il risultato è 12;

11. *si riprende l'esecuzione dell'unità di programma chiamante* dal punto in cui era stata interrotta dall'invocazione: il valore 12 viene assegnato alla variabile b.

Nel passaggio di parametri per valore il contenuto delle variabili usate come parametri attuali non viene mai modificato. Il programma illustrato stampa quindi `5 12`.

### 4.3.2. Passaggio di parametri tramite puntatori

L'uso di variabili di tipo puntatore consente diverse modalità di passaggio dei parametri.

Abbiamo inizialmente considerato il passaggio di parametri per *valore*, in genere utilizzato per i dati di tipo primitivo, in cui il parametro formale può essere considerato come una variabile locale che viene inizializzata al momento della chiamata della funzione con il valore corrispondente al parametro attuale.

Vediamo innanzitutto che il passaggio di parametri per valore può essere effettuato anche con il tipo puntatore, anche se viene a cadere una proprietà cruciale del passaggio di parametri per valore, cioè la garanzia che la funzione non abbia effetti sul contenuto delle variabili del programma chiamante.

Con il passaggio di puntatori diventa possibile utilizzare il puntatore per restituire al programma chiamante il risultato di una funzione.

Il passaggio di parametri di tipo puntatore risulta necessario anche quando i dati che devono essere scambiati tra funzione chiamata e programma chiamante sono voluminosi ed il passaggio del puntatore

risulta molto più efficiente sia in termini di occupazione di memoria che di tempo di calcolo (non occorre la copia del parametro).

*Esempio:* La seguente funzione ha lo scopo di scambiare il valore di due variabili.

```
void swap (int *a, int *b) {
     int temp = *b;
     *b = *a;
     *a = temp;
}
```

```
int main (){
     int j,i;
     i = 1;  j = 2;
     printf("i = %d\n",i);
     printf("j = %d\n",j);
     swap(&i, &j);
     printf("dopo swap\n");
     printf("i = %d\n",i);
     printf("j = %d\n",j);
}
```

Si noti che in questo caso, nonostante il passaggio di parametri sia per valore, il risultato dell'esecuzione della funzione swap consiste proprio nel modificare il valore di due variabili del programma principale.

In pratica, vengono passati alla funzione i puntatori a due variabili e pertanto è possibile modificare il valore delle variabili puntate, anche se il valore delle variabili puntatore rimane inalterato.

Il meccanismo di passaggio tramite puntatore può essere sfruttato in una funzione per restituire più valori.

*Esempio:* La funzione puntoMedio calcola le coordinate del punto medio dei punti (x1,y1) e (x2,y2) e ne memorizza le coordinate nelle variabili a cui fanno riferimento i puntatori passati come ultimi due parametri.

**punto-medio.c**

```
#include <math.h>
#include <stdio.h>

void puntoMedio(double x1, double y1,
     double x2, double y2, double* xm, double* ym){
    *xm = (x1 + x2)/2;
```

```
    *ym = (y1 + y2)/2;
}

int main(){
    double x1 = 1, y1 = 1, x2 = 3, y2 = 5, xm, ym;
    puntoMedio(x1,y1,x2,y2,&xm,&ym);
    printf("punto medio = (%lf,%lf)\n", xm, ym);
}
```

Si noti come al termine dell'esecuzione della funzione puntoMedio, il risultato sia visibile nelle variabili xm e ym, i cui riferimenti sono stati forniti in input tramite puntatori (ultimi due parametri) alla funzione.

Il passaggio di parametri tramite puntatori consente di accedere all'area di memoria associata ad una variabile e può essere pertanto usato anche per analizzarne il suo contenuto.

*Esempio:* La funzione printhex definita nel programma seguente stampa il contenuto di una zona di memoria individuata dal puntatore x e di dimensione sz, in notazione desadecimale, a partire dal byte più significativo (indirizzo di memoria più alto) fino al byte meno significativo (indirizzo di memoria più basso).

**printhex.c**

```
#include<stdio.h>

void printhex(char c, void *x, int sz) {
  unsigned char *p = (unsigned char *)x;
  printf("%c (%2d) [", c, sz*8);
  for (int i=sz-1; i>0; i--)
    printf("%02X|", *(p+i));
  printf("%02X]\n", *p);
}


int main() {

  double d = 195905.94;
  float f = (float)d;
  int i = (int)d;
  short s = (short)d;
  unsigned char c = (unsigned char)d;

  printf("d %f\n", d);
  printf("f %f\n", f);
  printf("i %d\n", i);
  printf("s %d\n", s);
  printf("c %d %c\n", c, c);
  printf("\n");
```

```
    printhex('d',&d,sizeof(d));
    printhex('f',&f,sizeof(f));
    printhex('i',&i,sizeof(i));
    printhex('s',&s,sizeof(s));
    printhex('c',&c,sizeof(c));

}
```

La funzione printhex è usata nel programma illustrato per mostrare il comportamento delle conversioni di tipo, producendo il seguente output.

```
d 195905.940000
f 195905.937500
i 195905
s -703
c 65 A

d (64) [41|07|EA|0F|85|1E|B8|52]
f (32) [48|3F|50|7C]
i (32) [00|02|FD|41]
s (16) [FD|41]
c ( 8) [41
```

Si noti che, come già visto nel capitolo precedente, le conversioni di tipo da double a float o int comportano approssimazioni, mentre le conversioni verso tipi a dimensione ridotta (short e char) comportano perdita di informazione (in questo caso, i valori dei byte più significativi).

#### 4.3.2.1. Parametri puntatori a costanti

Quando il parametro passato per riferimento non deve essere modificato dalla funzione, si può utilizzare la specifica const

```
void f (const double *x) {
    *x=2.0; // ERRORE di compilazione
            // *x non puo' essere modificato
    ...
}
```

In questo caso il compilatore segnala che c'è un'assegnazione non permessa. In generale con i puntatori a costanti si possono impedire effetti collaterali indesiderati.

### 4.3.3. Passaggio di parametri per riferimento

Il passaggio dei parametri per riferimento viene illustrato nel seguito per completare l'analisi delle modalità di passaggio dei parametri. Occorre tuttavia ricordare che il passaggio di parametri per riferimento non è consentito in C (ma solo in C++).

Nel passaggio dei parametri per riferimento, il parametro attuale fornito alla funzione è il riferimento (cioè l'indirizzo di memoria) di una variabile. Tuttavia, nella chiamata e nel corpo della funzione si usa la notazione delle variabili normale, cioè senza puntatore.

La dichiarazione di un parametro passato per riferimento avviene usando il carattere & davanti al nome del parametro formale.

*Esempio:*

```
void swapRef (int &a, int &b) {
    int temp;
    temp = b;
    b = a;
    a = temp;
    return;
}
```

```
int main (){
    int i = 1;
    int j = 2;
    printf("i = %d\n",i);
    printf("j = %d\n",j);
    swapRef(i, j);
    printf("dopo swapRef\n");
    printf("i = %d\n",i);
    printf("j = %d\n",j);
    return 0;
}
```

In questo caso, la funzione C++ che scambia il valore delle variabili, si comporta esattamente come la precedente, ma l'uso del passaggio di parametro per riferimento ne rende più chiara la scrittura ed esplicita la possibilità di modificare i contenuti delle variabili del modulo di programma chiamante, senza usare variabili puntatore esplicite.

### 4.3.4. Valori restituiti di tipo puntatore

Le funzioni C possono restituire valori di tipo puntatore o di tipo riferimento. Cioè, il risultato di una funzione può essere di tipo puntatore.

```
double *puntatore (double a) {
    double *r = (double *) malloc(sizeof(double));
    *r = a;
    return r;
}

int main () {
    double *pd = puntatore(5.4);
    printf("pd = %p\n",pd);
    printf("*pd = %f\n",*pd);
    return 0;
}
```

In questo esempio la funzione puntatore crea un puntatore ad una variabile di tipo double e la inizializza con il valore passato come argomento.

Si noti che la memoria allocata dinamicamente con la funzione malloc non viene rilasciata al termine dell'esecuzione della funzione. Nell'esempio precedente la variabile puntatore r (variabile locale allocata staticamente) viene deallocata al termine della funzione, mentre la locazione di memoria puntata da r (allocata dinamicamente) rimane allocata.

Attenzione: la variabile a cui punta il risultato della funzione deve essere allocata dinamicamente.

```
double *puntatore (double a) {
    double d = a;
    double *r = &d;  // ERRORE: d allocata
    staticamente
    return r;
}

int main () {
    double *pd = puntatore(5.4);
    printf("pd = %p\n",pd);
    printf("*pd = %f\n",*pd);
    return 0;
}
```

In questo esempio la funzione puntatore restituisce il puntatore ad una variabile che viene rilasciata al termine dell'esecuzione della funzio-

ne! Il suo indirizzo di memoria potrebbe essere usato successivamente (in quanto è ritornato dalla funzione), ma la zona di memoria corrispondente non è più allocata. Si potrebbero verificare quindi diversi tipi di errori di esecuzione.

### 4.3.5. Parametri di tipo puntatore a funzione

Un caso particolare di passaggio di parametri di tipo puntatore, è quello in cui il puntatore passato alla funzione punta ad una funzione.

Consideriamo ad esempio una funzione che calcola il valore di una funzione nell'intervallo [a,b] della funzione in ingresso. La sua intestazione sarà:

```
void calcola(double (*f)(double), double primo,
             double ultimo, double inc)
```

oppure semplicemente

```
double calcola(double f(double), double primo,
               double ultimo, double inc)
```

Programma chiamante:

```
risultato = calcola(sin, 0.0, PI / 2, 0.1);
```

All'interno del corpo della funzione calcola possiamo chiamare la funzione alla quale punta f:

```
y = (*f)(x);
```

Il programma completo è il seguente:

**pfun.c**

```
#include <math.h>
#include <stdio.h>

void calcola(double (*f)(double), double primo,
             double ultimo, double inc);
```

```
int main(void) {
    double ini, fin, inc;
    printf("immetti valori (iniziale, incremento,
    finale): ");
    scanf("%lf %lf %lf", &ini, &fin, &inc);
    printf("\n       x         cos(x)"
           "\n    -------     -------\n");
    calcola(cos, ini, fin, inc);
    printf("\n       x         sin(x)"
           "\n    -------     -------\n");
    calcola(sin, ini, fin, inc);
  return 0;
}


void calcola(double (*f)(double), double primo,
             double ultimo, double inc)
{
    double x;
    int i, num_intervalli;
    num_intervalli = ceil((ultimo - primo) / inc);
    for (i = 0; i <= num_intervalli; i++) {
      x = primo + i * inc;
      printf("%10.5f %10.5f\n", x, (*f)(x));
    }
}
```

## 4.4. Variabili locali di una funzione

Il corpo di una funzione può contenere dichiarazioni di variabili. Tali variabili vengono dette **variabili locali**. Abbiamo già visto la distinzione tra variabili locali e variabili globali. Dato che le funzioni consentono di definire variabili locali, riconsideriamo due aspetti fondamentali ad esse relativi:

- **campo d'azione** (è una nozione statica, che dipende dal testo del programma)
- **tempo di vita** (è una nozione dinamica, che dipende dall'esecuzione del programma)

### 4.4.1. Campo d'azione delle variabili locali

Come abbiamo visto, il **campo d'azione** (o **scope**) di una variabile è *l'insieme delle unità di programma in cui la variabile è visibile* (cioè accessibile ed utilizzabile).

Nel caso delle definizioni di funzione, il campo di azione di una variabile locale è il corpo della funzione in cui essa è dichiarata. Cioè la variabile è visibile nel corpo della funzione in cui compare la sua dichiarazione, mentre non è visibile all'esterno.

Questa considerazione deriva dalla regola generale sul campo d'azione delle variabili: una variabile dichiarata in un qualsiasi blocco (istruzione {...}) è visibile in quel blocco (inclusi eventuali blocchi interni), ma non è visibile all'esterno del blocco stesso (vedi Unità 1). Naturalmente, come abbiamo specificato nell'Unità 1, una variabile non può essere utilizzata nel corpo della funzione prima di essere dichiarata.

*Nota:* il campo d'azione di una variabile è una nozione completamente statica. Infatti esso può essere stabilito analizzando la struttura del programma, senza considerare come il programma si comporta in esecuzione. La maggior parte dei linguaggi di programmazione attualmente in uso adotta questa nozione detta *campo di azione (o scope) statico*. Pertanto il campo d'azione è un concetto rilevante a tempo di compilazione.

#### 4.4.1.1. Esempio: campo di azione di variabili locali

Consideriamo il seguente programma.

```
int raddoppia (int x) {
    return x*2;
}
void stampa() {
    printf("a = %d\n",a);    //ERRORE a non e' definito
}
int main() {
    int a = 5;
    a = raddoppia(a);
    stampa();
    printf("a = %d\n",a);
}
```

Durante la compilazione del programma sarà evidenziato un errore: nella funzione stampa la variabile a non è visibile (perché definita nella funzione main).

### 4.4.2. Variabili locali definite static

In C esiste un meccanismo per consentire di definire variabili locali ad un blocco che mantengono il valore tra due esecuzioni successive

del blocco stesso. Per ottenere questo effetto occorre usare la parola riservata static nella definizione della variabile.

*Esempio:* Il programma seguente

**static.c**

```
#include <stdio.h>

int ricorda() {
    int static c;
    c++;
    return c;
}

int main() {
    printf("ricorda() = %d\n",ricorda());
    printf("ricorda() = %d\n",ricorda());
    printf("ricorda() = %d\n",ricorda());
}
```

stampa

```
ricorda() = 1
ricorda() = 2
ricorda() = 3
```

Si osservi che la variabile statica è differente da una variabile globale (vedi dopo) in quanto essa non è comunque visibile all'esterno del blocco. Inoltre le variabili statiche (a differenza delle variabili locali o globali) vengono sempre inizializzate a zero.

La parola static si può trovare associata anche alle definizioni di funzione. In questo caso il suo significato è di specificare che la definizione della funzione è valida soltanto all'interno del file in cui si trova (vedi successivamente l'alternativa definizione extern).

## 4.5. Variabili globali

In C le variabili *globali*, cioè definite al di fuori di una definizione di funzione sono visibili in tutte le funzioni. Quindi in ogni funzione sono visibili le variabili dichiarate localmente e le variabili globali.

*Esempio:*

```c
int a = 1; // dichiarazione globale di a
void f1 () {
   printf("In f1\n");
   printf("a = %d\n",a);
// printf("b = %d\n",b); errore in compilazione
   return;
}
int main () {
   int b = 1;
   f1();
   printf("In main\n");
   printf("a = %d\n",a);
   printf("b = %d\n",b);
   return 0;
}
```

## 4.6. Tempo di vita delle variabili

Il **tempo di vita** di una variabile è il *tempo in cui la variabile rimane effettivamente accessibile in memoria durante l'esecuzione*.

Si distinguono tre casi: variabili locali, variabili globali, variabili statiche.

Una variabile definita all'interno di un blocco di istruzioni ha un tempo di vita pari al tempo di esecuzione del blocco stesso. In particolare, le variabili locali ad una funzione vengono allocate al momento dell'attivazione della funzione (come i parametri formali) e vengono deallocate al momento dell'uscita dall'attivazione. Quindi il tempo di vita di queste variabili corrisponde al tempo in cui viene eseguita la funzione.

Le variabili globali e le variabili statiche invece hanno un tempo di vita pari a tutta la durata dell'esecuzione del programma.

*Esempio*: tempo di vita delle variabili locali

Consideriamo il seguente programma.

```
int raddoppia (int x) {
    int temp = x*2;
    return temp;
}
void stampa(int b) {
    printf("b = %d\n",b);
}

int main() {
    int a = 5;
    a = raddoppia(a);
    a = raddoppia(a);
    stampa(a);
    printf("a = %d\n",a);
}
```

Durante l'esecuzione del programma la variabile a ha un tempo di vita pari al tempo di esecuzione della funzione main.

Il parametro x della funzione raddoppia (che corrisponde ad una variabile locale) e la variabile locale temp hanno un tempo di vita che corrisponde al tempo di esecuzione della funzione raddoppia. Si noti che ad ogni esecuzione di raddoppia corrispondono in esecuzione diverse istanze di variabili.

Analogamente il parametro x della funzione stampa ha un tempo di vita pari all'esecuzione di stampa.

Si noti, infine, che durante le esecuzioni di raddoppia e stampa la variabile a, rimane in vita anche se non è visibile.

## 4.7. Modello run-time

Con il termine *modello run-time* si indicano i meccanismi che consentono l'esecuzione dei programmi. In particolare, di seguito viene esaminata la parte del modello run-time che riguarda la gestione delle chiamate di funzione.

A tempo di esecuzione, il sistema operativo deve gestire diverse zone di memoria per l'esecuzione di un programma:

- zona che contiene il codice eseguibile del programma
  - determinata a tempo di esecuzione al momento del caricamento del programma
  - dimensione fissata per ogni funzione a tempo di compilazione

- **heap**: zona di memoria che contiene la memoria allocata dinamicamente
  - cresce e decresce dinamicamente durante l'esecuzione
  - ogni area di memoria viene allocata e deallocata indipendentemente dalle altre
- **pila dei record di attivazione (o stack)**: zona di memoria per i dati locali alle funzioni (variabili e parametri)
  - cresce e decresce dinamicamente durante l'esecuzione
  - viene gestita con un meccanismo a *pila*

### 4.7.1. Record di attivazione

Quando viene attivata una funzione, viene allocato uno spazio in memoria, detto *record di attivazione*.

Il record di attivazione contiene le seguenti informazioni:

- locazioni di memoria per i parametri formali;
- locazioni di memoria per le variabili locali (se presenti);
- locazione di memoria per memorizzare il valore di ritorno dell'invocazione della funzione (se la funzione ha tipo di ritorno diverso da void);
- locazione di memoria per l'indirizzo di ritorno, ovvero l'indirizzo della prossima istruzione da eseguire nella funzione chiamante.

Tale record viene utilizzato durante l'esecuzione della funzione e viene poi deallocato alla fine dell'esecuzione stessa. Quando il record di attivazione viene deallocato, le locazioni di memoria per le variabili locali e i parametri formali vengono quindi deallocate e il loro contenuto non è più accessibile.

Ad una nuova attivazione della funzione corrisponde una nuova allocazione delle variabili (che non ha nulla in comune con quella delle attivazioni precedenti). Ne segue che ad ogni attivazione della funzione vengono allocate locazioni di memoria diverse per le variabili locali e i parametri formali.

### 4.7.2. Pila dei record di attivazione

Una *pila* (o *stack*) è una struttura dati con accesso LIFO: Last In First Out = ultimo entrato è il primo a uscire (Es.: pila di piatti).

A run-time il sistema operativo gestisce la **pila dei record di attivazione** (RDA):

- per *ogni attivazione di funzione* viene creato un nuovo RDA in cima alla pila;
- al termine dell'attivazione della funzione il RDA viene rimosso dalla pila.

### 4.7.3. Esempio di evoluzione della pila dei record di attivazione

Consideriamo le seguenti tre funzioni main, A e B e vediamo cosa avviene durante l'esecuzione della funzione main.

```
int B(int pb) {
/* b0 */ printf("In B. Parametro pb = %d\n", pb);
/* b1 */ return pb+1;
}

int A(int pa) {
/* a0 */ printf("In A. Parametro pa = %d\n", pa);
/* a1 */ printf("Chiamata di B(%d).\n", pa * 2);
/* a2 */ int va = B(pa * 2);
/* a3 */ printf("Di nuovo in A. va = %d\n", va);
/* a4 */ return va + pa;
}
```

```
int main() {
/* m0 */ printf("In main.\n");
/* m1 */ int vm = 22;
/* m2 */ printf("Chiamata di A(%d)\n",vm);
/* m3 */ vm = A(vm);
/* m4 */ printf("Di nuovo in main. vm = %d\n", vm);
/* m5 */ return 0;
}
```

Per semplicità, assumiamo che ad ogni istruzione del codice sorgente C corrisponda una singola locazione di memoria.

| | main |
|---|---|
| | ... |
| 100 | m0 |
| 101 | m1 |
| 102 | m2 |
| 103 | m3 |
| 104 | m4 |
| 105 | return |
| 106 | ... |

⇒ A(vm)

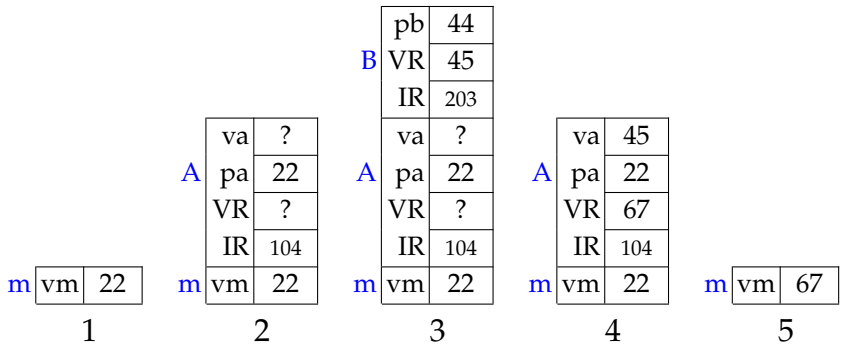
| | A |
|---|---|
| | ... |
| 200 | a0 |
| 201 | a1 |
| 202 | a2 |
| 203 | a3 |
| 204 | return |
| 205 | ... |

⇒ B(va*2)

| | B |
|---|---|
| | ... |
| 300 | b0 |
| 301 | return |
| 302 | ... |

```
In main.
Chiamata di A(22).
In A. Parametro pa = 22
Chiamata di B(44).
In B. Parametro pb = 44
Di nuovo in A. va = 45
Di nuovo in main. vm = 67
```

Evoluzione della pila dei RDA (m usato per indicare la funzione main):




Per comprendere cosa avviene durante l'esecuzione del codice, è necessario fare riferimento, oltre che alla pila dei RDA, al **program counter** (PC), il cui valore è l'indirizzo della prossima istruzione da eseguire.

Analizziamo in dettaglio cosa avviene al momento dell'**attivazione** di A(vm) dalla funzione main. Prima dell'attivazione, la pila dei RDA è come mostrato in 1 nella figura di sopra:

1. *vengono valutati i parametri attuali*: nel nostro caso il parametro attuale è l'espressione vm che ha come valore l'intero 22;

2. *viene individuata la funzione da eseguire* in base al numero e tipo dei parametri attuali, cercando la definizione di una funzione la cui segnatura sia conforme alla invocazione (il nome della funzione deve essere lo stesso e i parametri attuali devono corrispondere in numero e tipo ai parametri formali): nel nostro caso la funzione da eseguire deve avere la segnatura A(int);

3. *viene sospesa l'esecuzione della funzione chiamante*: nel nostro caso la funzione main;

4. *viene creato il RDA* relativo all'attivazione corrente della funzione chiamata: nel nostro caso viene creato il RDA relativo all'attivazione corrente di A; il RDA contiene:

   - le locazioni di memoria per i parametri formali: nel nostro caso, il parametro pa di tipo int;
   - le locazioni di memoria per le variabili locali: nel nostro caso, la variabile va di tipo int;
   - una locazione di memoria per il valore di ritorno: nel nostro caso indicata con VR;
   - una locazione di memoria per l'indirizzo di ritorno: nel nostro caso indicata con IR;

5. *viene assegnato il valore dei parametri attuali ai parametri formali*: nel nostro caso, il parametro formale pa viene inizializzato con il valore 22;

6. *l'indirizzo di ritorno nel RDA viene impostato all'indirizzo della prossima istruzione che deve essere eseguita nella funzione chiamante al termine dell'invocazione*: nel nostro caso, l'indirizzo di ritorno nel RDA relativo all'attivazione di A viene impostato al valore 104, che è l'indirizzo dell'istruzione di main corrispondente all'istruzione m4, da eseguire quando l'attivazione di A sarà terminata; a questo punto, la pila dei RDA è come mostrato in 2 nella figura di sopra;

7. *al program counter viene assegnato l'indirizzo della prima istruzione della funzione invocata*: nel nostro caso, al program counter viene assegnato l'indirizzo 200, che è l'indirizzo della prima istruzione di A;

8. *si passa ad eseguire la prossima istruzione indicata dal program counter, che sarà la prima istruzione della funzione invocata*: nel nostro caso l'istruzione di indirizzo 200, ovvero la prima istruzione di A.

Dopo questi passi, le istruzioni della funzione chiamata, nel nostro caso di A, vengono eseguite in sequenza. In particolare, avverrà l'attivazione, l'esecuzione e la terminazione di eventuali funzioni a loro volta invocate nella funzione chiamata. Nel nostro caso, avverrà l'attivazione, l'esecuzione e la terminazione della funzione B, con un meccanismo analogo a quello adottato per A; la pila dei RDA passerà attraverso gli stati 3 e 4.

Analizziamo ora in dettaglio cosa avviene al momento della **terminazione dell'attivazione** di A, ovvero quando viene eseguita l'istruzione return va+pa;. Prima dell'esecuzione, la pila dei RDA è come mostrato in 4 nella figura di sopra, (in realtà, la zona di memoria predisposta a contenere il valore di ritorno, indicata con VR nella figura, viene inizializzata contestualmente all'esecuzione dell'istruzione return, e non prima):

1. *al program counter viene assegnato il valore memorizzato nella locazione di memoria riservata all'indirizzo di ritorno nel RDA corrente*: nel nostro caso, tale valore è pari a 104, che è proprio l'indirizzo, memorizzato in IR, della prossima istruzione di main che dovrà essere eseguita;

2. *nel caso la funzione invocata preveda la restituzione di un valore di ritorno, tale valore viene memorizzato in un'apposita locazione di memoria del RDA corrente*: nel nostro caso, il valore 67, risultato della valutazione dell'espressione va+pa viene assegnato alla locazione di memoria indicata con VR, predisposta per contenere il valore di ritorno;

3. *viene eliminato dalla pila dei RDA il RDA relativo all'attivazione corrente, e il RDA corrente diviene quello immediatamente precedente nella pila; contestualmente all'eliminazione del RDA dalla pila dei RDA, un eventuale valore di ritorno viene copiato in una locazione di memoria del RDA del chiamante*: nel nostro caso, viene eliminato il RDA relativo all'attivazione di A e il RDA corrente diviene quello relativo all'attivazione di main; inoltre, il valore 67, memorizzato nella locazione di memoria VR viene assegnato alla variabile vm nel RDA di main; la pila dei RDA è come mostrato in 5 nella figura di sopra;

4. *si passa ad eseguire la prossima istruzione indicata dal program counter*, ovvero quella appena impostata al passo 1: nel nostro caso, si passa ad eseguire l'istruzione di indirizzo 104, che fa riprendere l'esecuzione di main.

# 5. Tipi di dato indicizzati

## 5.1. Array

Un **array** è una struttura contenente una collezione di elementi dello stesso tipo, ciascuno indicizzato da un valore intero. Una **variabile di tipo array** è un riferimento alla collezione di elementi che costituisce l'array.

Per usare un array in C occorre:

1. dichiarare una variabile di tipo array specificandone la dimensione (numero di elementi contenuti);
2. accedere mediante la variabile agli elementi dell'array per assegnare o leggerne i valori (trattando ciascun elemento come se fosse una variabile).

### 5.1.1. Dichiarazione di variabili di tipo array

Per usare un array bisogna prima dichiarare una variabile di tipo array.

**Dichiarazione di variabili di tipo array**

*Sintassi:*

```
tipo nomeArray[n];
```

dove:

- *tipo* è il tipo degli elementi contenuti nell'array;
- *nomeArray* è il nome della variabile (riferimento ad) array

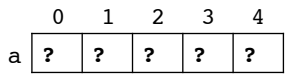
che si sta dichiarando

- *n* è un'espressione costante che rappresenta il numero di elementi dell'array (C99/C++ ammette anche espressioni variabili).

*Semantica:*
Alloca un array di *n* elementi di tipo *tipo* e crea la variabile *nomeArray* di tipo array.

*Esempio:*

```
int a[5]; // a e' una variabile di tipo
          // array di 5 interi
```




La creazione di un array corrisponde alla allocazione di n blocchi di memoria *contigui*, ciascuno di dimensione opportuna (ad es., blocchi da 32 bit se gli elementi sono di tipo int). Una dichiarazione di variabile di tipo array comporta un'allocazione dell'array di tipo *statico*. Ciò significa che lo spazio di memoria contenente l'array è fissato al momento della dichiarazione e non varia durante l'esecuzione del programma (contrariamente al suo contenuto, che è ovviamente modificabile). In particolare, la dimensione dell'array rimane invariata per tutto il tempo di vita. Osserviamo inoltre che lo spazio di memoria corrispondente ad una variabile di tipo array viene allocato nello stack, fatto che implica la fine del tempo di vita della variabile (ovvero il rilascio della memoria) quando il blocco in cui è stato dichiarato termina.

Possiamo conoscere la dimensione di un array statico tramite la funzione sizeof.

*Esempio:*

```
int a[5];
printf("%d byte\n", sizeof(a));  // 20 byte
printf("%d elementi\n", sizeof(a)/sizeof(int));  // 5
    elementi
```

La prima invocazione della `printf` stampa: `20 byte`, ovvero la quantità di memoria necessaria a contenere 5 int (ciascuno di 4 byte). La seconda stampa invece: `5 elementi` (20/4).

### 5.1.2. Accesso agli elementi di un array

Si può accedere ai singoli elementi di un array tramite l'operatore di *subscripting* (o *indicizzazione*) [].

**Accesso agli elementi di un array**

*Sintassi:*

```
nomeArray[indice]
```

dove

- *nomeArray* è l'identificatore della variabile array che contiene un riferimento all'array a cui si vuole accedere
- *indice* è un'espressione di tipo int non negativa che specifica l'indice dell'elemento a cui si vuole accedere.

*Semantica:*
Accede all'elemento di indice *indice* dell'array *nomeArray* per leggerlo o modificarlo.

Se l'array *nomeArray* contiene *n* elementi, la valutazione dell'espressione *indice* deve fornire un numero intero nell'intervallo [0, *n*-1].

*Esempio:*

```
int a[5];  // a e' una variabile di tipo array di
    interi
           // viene creato un array con 5 elementi int
a[0] = 23; // assegnazione al primo elemento dell'
    array
a[4] = 92; // assegnazione all'ultimo elemento dell'
    array
a[5] = 16; // ERRORE: l'indice 5 non e' nell'
    intervallo [0,4]
```

È molto importante ricordare che, se l'array contiene $N$ elementi ($N = 5$ nell'esempio), gli unici indici validi sono gli interi nell'intervallo $[0, N-1]$. Tentare di accedere ad un elemento con indice al di fuori di esso può generare un errore a tempo di esecuzione. Nell'esempio precedente, si ha un errore quando viene eseguita l'istruzione a[5]=16;. Mentre esistono dei linguaggi di programmazione che sono in grado di segnalare questo tipo di errore (ad esempio il Python informa il programmatore che l'indice è al di fuori dell'intervallo), il C in genere non aiuta il programmatore. Errori di questo tipo possono essere non rilevati nel momento in cui si verificano ed in genere hanno effetti imprevedibili, in quanto corrispondenti alla scrittura/lettura di locazioni di memoria esterne all'array.

Si osservi inoltre che dichiarando una variabile di tipo array, viene contestualmente creato l'array a cui essa si riferisce (cioè viene allocata memoria).

### 5.1.3. Inizializzazione di array tramite espressioni

In C è possibile inizializzare gli elementi di un array usando espressioni numeriche.

**Inizializzazione di array tramite espressioni**

*Sintassi:*

```
tipo nomeArray[] = { espr_0, ..., espr_n-1 }.
```

dove:

- *tipo* è il tipo degli elementi dell'array;
- *nomeArray* è l'identificatore dell'array;

- *espr_i* è un'espressione di tipo *tipo*.

*Semantica:*
*nomeArray* viene inizializzato ad un array di *n* elementi di tipo *tipo*, dove l'elemento di indice *i* ha valore pari al valore restituito dall'espressione *espr_i* (opportunamente convertita, laddove necessario).

*Esempio:*

```
int v[] = { 4, 6, 3, 1 };
// oppure int v[4] = { 4, 6, 3, 1 };
```

è equivalente a:

```
int v[4];
v[0] = 4; v[1] = 6; v[2] = 3; v[3] = 1;
```

L'assegnazione ad un array tramite espressioni può avvenire *solo all'interno di una dichiarazione di array*.

*Esempio:*

```
int v[4];
v = { 4, 6, 3, 1 }; // errato
```

Il seguente programma memorizza in un array 10 valori interi letti da input e ne restituisce la somma.

*Esempio:* Somma degli elementi di un array di interi

**somma-array.c**

```
int a[10];
int n_elementi = sizeof(a)/sizeof(int);
for (int i = 0; i < n_elementi; i++){
    printf("Inserisci il valore di a[%d]: ",i);
    scanf("%d",&a[i]);
}
int somma = 0;
for (int i = 0; i < n_elementi; i++){
    somma += a[i];
}
```

```
printf("La somma degli elementi e': %d\n",somma);
```

Si noti come l'uso della variabile n_elementi permetta di lasciare il programma essenzialmente invariato nel caso la dimensione dell'array venisse cambiata.

### 5.1.4. Variabili array e puntatori

#### 5.1.4.1. Accesso ad array tramite puntatori

L'identificatore di una variabile di tipo array denota un puntatore alla prima locazione di memoria dell'array (ovvero al primo elemento).

*Esempio:* Nel seguente frammento, l'identificatore a viene usato come un puntatore di tipo char*.

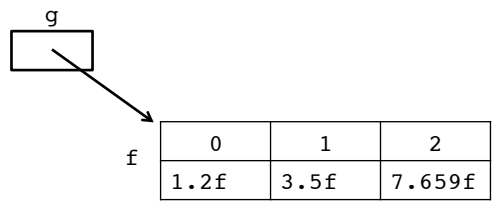
```
char a[3] = {'a','b','c'};
printf("%c\n",*a); // stampa a
```

Viceversa, un puntatore allo stesso tipo degli elementi di un array può essere usato per accedere all'array.

*Esempio:* In questo esempio, il puntatore g viene usato per accedere all'array f.

```
float f[3] = {1.2f, 3.5f, 7.659f};
float* g = f;
printf("%f\n",*(g+2));// stampa il valore in f[2]
```

Come mostrato nella figura seguente, f e g condividono lo stesso frammento di memoria.




Le modifiche apportate all'array tramite uno qualsiasi dei riferimenti sono pertanto visibili accedendo all'array tramite l'altro.

```
*(g+2)=0;
printf("f[2]=%f\n",f[2]); // stampa f[2]=0.000000
```

Nonostante l'identificatore di una variabile di tipo array denoti un puntatore, a tali variabili non possono essere assegnati valori.

*Esempio:*

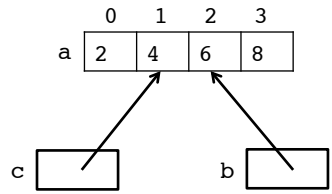
```
f = g; // ERRORE: f non e' assegnabile
```

Poiché un array identifica un insieme di blocchi di memoria contigui, avendo a disposizione il puntatore ad uno dei suoi elementi, è possibile accedere agli altri elementi applicando le operazioni di somma e sottrazione.

*Esempio:*

```
int a[4] = {2,4,6,8};
int* b = a;
b += 2;
printf("*b=%d\n",*b);// stampa: *b=6
int* c = &a[3];
c -= 2;
printf("*c=%d\n",*c);// stampa: *c=4
```

La figura seguente mostra lo stato della memoria al termine del frammento di codice riportato sopra.




#### 5.1.4.2. Operatore di subscripting e puntatori

L'operatore di subscripting [] può essere applicato ad un puntatore, indipendentemente dal fatto che esso punti ad un array o meno. Il valore restituito da un'espressione della forma *puntatore*`[`*indice*`]` è pari al valore restituito dall'espressione *(*puntatore*+*indice*). Si osser-

vi che l'espressione restituisce il *contenuto* della locazione puntata da *puntatore+indice*, non il valore del puntatore.

*Esempio:* Le seguenti assegnazioni sono equivalenti:

```
int v, i, *p;
...
v = p[i];
v = *(p + i);
```

Nel caso in cui il puntatore fa riferimento ad un array, l'uso dell'operatore di subscripting è particolarmente utile, in quanto permette di accedere all'array tramite puntatore con le stesse modalità viste per variabili di tipo array.

*Esempio:*

```
char v[3]={'a','b','c'};
char *p = v;
printf("%c\n",v[2]); // stampa c
printf("%c\n",p[2]); // stampa c
```

#### 5.1.4.3. Sottrazione di puntatori

La differenza tra puntatori che puntano ad elementi di uno stesso array è pari alla differenza tra gli indici degli elementi puntati.

*Esempio:*

```
int a[4] = {2,4,6,8};
int* b = &a[2];
int* c = &a[1];

printf("c-b= %ld\n",c-b); // stampa -1
printf("b-c= %ld\n",b-c); // stampa 1
```

La sottrazione tra puntatori che non fanno riferimento ad elementi di uno stesso array produce un comportamento indefinito.

### 5.1.5. Passaggio di parametri di tipo array

Anche gli array possono essere usati come parametri di funzione.

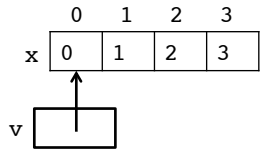
*Esempio:* La seguente funzione prende in input un array di interi e la sua dimensione e restituisce la somma degli elementi contenuti nell'array.

```
int sommaValoriArray(int v[], int n) {
    int somma = 0;
    for (int i=0; i < n; i++)
        somma += v[i];
    return somma;
}
```

*Esempio d'uso:*

```
int main() {
    const int n = 4;
    int x[n] = {0,1,2,3};
    printf("somma = %d\n" ,sommaValoriArray(x,n));
}
```

Nell'esempio, la specifica del parametro formale int v[] indica che il tipo del parametro denotato da v è *array di* int. In effetti, il parametro v viene trattato semplicemente come un puntatore ad intero: al momento dell'invocazione della funzione, viene copiato nel parametro formale v il valore dell'espressione x (cioè l'indirizzo della prima locazione dell'array) passato come parametro attuale. Per quanto riguarda l'array, invece, esso non viene copiato, cosicché eventuali modifiche ad esso apportate dalla funzione risulteranno visibili al modulo chiamante. In altre parole, tramite il riferimento, la funzione può effettuare side-effect sull'array passato in input. La figura seguente mostra lo stato della memoria all'atto dell'invocazione di sommaValoriArray nella funzione main dell'esempio precedente.




Si osservi che al momento dell'invocazione di sommaArray, non viene creata una copia dell'array, ma viene solo copiato, nella variabile v il riferimento ad esso. Ciò avviene indipendentemente dal fatto che l'array sia memorizzato nello heap (per effetto di un'allocazione

dinamica) o nello stack (per effetto di una dichiarazione locale avvenuta in precedenza all'interno di un blocco non ancora terminato, ad es. il modulo chiamante).

È anche possibile indicare esplicitamente, nell'intestazione della funzione, il numero di elementi contenuti nell'array di input. *Esempio:* La seguente funzione prende in input solo array con 3 elementi.

```
void f(int v[3]) { ... }
```

Poiché i parametri di tipo array vengono considerati a tutti gli effetti puntatori, è anche possibile usare un parametro di tipo puntatore, nella segnatura di una funzione, in luogo di un parametro di tipo array. Nell'esempio precedente, avremmo potuto usare la specifica int *v invece di int v[], rendendo esplicito l'uso di un riferimento. Si noti tuttavia che in questo modo si perde l'utile indicazione, fornita dalla segnatura della funzione, che il riferimento è ad un array.

Per la stessa ragione, è anche possibile usare un puntatore in luogo di un array nell'invocazione di una funzione.

*Esempio:*

```
void f(char s[]) { // oppure void f(char *s)
...
}

int main(){
  char t[3] = {1,2,3};
  char *p = t;
  f(t); // oppure f(p)
}
```

Notiamo infine che l'invocazione della funzione sizeof su un parametro di tipo array, all'interno di una funzione, non restituisce la dimensione dell'array a cui il parametro fa riferimento.[1] Infatti, essendo il parametro trattato come una variabile di tipo puntatore, l'invocazione restituisce semplicemente la dimensione (in byte) dello spazio contenente il valore del parametro. Pertanto, laddove necessario, occorre

[1] Alcuni compilatori segnalano questa situazione con un warning, ad es.: warning: sizeof on array function parameter will return size of 'int *' instead of 'int []'.

passare il numero di elementi contenuti nell'array come parametro della funzione (o specificarlo esplicitamente nell'intestazione).

*Esempio:*

```
void f(int v[], int n) {
  printf("Dimensione di v: %d byte\n",sizeof(v)); // 4
  printf("Numero di elementi in v[]: %d\n",n);
}
```

#### 5.1.5.1. Esempio: ricerca sequenziale di un elemento in un array

La seguente funzione cercaArray prende come parametri un array di interi, un intero corrispondente alla dimensione dell'array ed un intero e da cercare nell'array e restituisce 1 (true) se il valore e è presente nell'array, 0 (false) altrimenti.

```
int cercaElemArray(int v[], int n, int e) {
    for (int i=0; i<n; i++)
        if (e == v[i])
            return 1;
    return 0;
}
```

*Esempio d'uso:*

```
int main () {
    const int n=4;
    int x[n]= {1,2,3,4};
    if (cercaElemArray(x,n,3)) // cerca 3 nell'array x
        printf("trovato\n");
    else
        printf("non trovato\n");
}
```

#### 5.1.5.2. Esempio: ricerca del valore massimo in un array

La seguente funzione massimoArray prende come parametri un array di long int e la sua dimensione e, assumendo che l'array non sia vuoto, restituisce il valore massimo che esso contiene.

```
long massimoArray(long v[], int n) {
    long max = v[0];
    for (int i=1; i<n; i++)
        if (v[i]>max) max = v[i];
    return max;
}
```

*Esempio d'uso:*

```
int main () {
  const int n = 5;
  long x[n] = { 5, 3, 9, 5, 12 };
  printf("Max = %ld",massimoArray(x,n));
}
```

#### 5.1.5.3. Esempio: gli ultimi saranno... i primi

Vediamo ora un esempio di funzione che effettua side-effect sui parametri di input di tipo array. La funzione rovesciaArray prende in input un array di interi e ne modifica il contenuto riorganizzando gli elementi in ordine inverso, dall'ultimo al primo.

rovesciaArray sfrutta la funzione ausiliaria scambia che effettua side-effect sulle locazioni a cui puntano i suoi argomenti, scambiandone il contenuto.

```
void scambia(int *i, int *j){
    int t=*i;
    *i=*j;
    *j=t;
}
```

```
void rovesciaArray(int v[], int n) {
    int temp;
    for (int i=0; i<n/2; i++)
        scambia(&v[i],&v[n-i-1]);
}
```

*Esempio d'uso:*

```
int main () {
    const int n=5;
    int x[n] = {5, 3, 9, 5, 12};
    for (int i=0; i<n; i++) // stampa 5 3 9 5 12
        printf("%d ", x[i]);
    printf("\n");
    rovesciaArray(x,n);
    for (int i=0; i<n; i++) // stampa 12 5 9 3 5
        printf("%d ", x[i]);
    printf("\n");
}
```

### 5.1.6. Array come risultato di una funzione

Una funzione può restituire un array. In questo caso, la restituzione può avvenire solo tramite puntatore. Si noti che l'array restituito non può essere allocato staticamente dalla funzione, in quanto, al pari di qualunque altra variabile locale, il suo tempo di vita terminerebbe al completamento dell'esecuzione della funzione.

*Esempio:* La funzione seguente crea un array e restituisce un puntatore ad esso.

```
int* creaArray(){
  int risultato[5] = {10,20,30,40,50};
  return risultato;
}
```

Nel seguente frammento di codice, la funzione creaArray viene invocata allo scopo di inizializzare il valore del puntatore a all'indirizzo dell'array da essa creato. Tuttavia, dopo l'assegnazione, la variabile a punta ad una locazione di memoria libera, in quanto al termine dell'esecuzione della funzione creaArray la memoria allocata per l'array risultato viene automaticamente rilasciata (e può essere usata, ad esempio, per allocare variabili locali di altre funzioni).

```
int main(){
  int* a = creaArray(); // a punta ad una locazione
    non allocata!
  ...
}
```

Questo problema, legato alla definizione della funzione creaArray viene evidenziato dal compilatore tramite un messaggio di warning: address of stack memory associated with local variable 'risultato' returned

In alternativa alla creazione del nuovo array nel corpo della funzione, si può allocare l'array (staticamente) nel modulo chiamante e passarlo come argomento alla funzione, che può effettuare side-effect sull'array. Tuttavia, per adottare questo approccio è necessario che la dimensione dell'array sia nota prima dell'esecuzione della funzione.

*Esempio:*

```
void inizializzaArray(int v[], int n){
  for(int i = 0; i < n; i++){
    v[i]=0;
    }
}
```

*Esempio d'uso:*

```
int main(){
  const int n = 10;
  int x[n];
  inizializzaArray(x,n);
}
```

Come soluzione generale, si può sfruttare l'allocazione *dinamica* dell'array, che permette di superare gli inconvenienti mostrati nei casi precedenti. Questo argomento sarà analizzato in dettaglio nel seguito. Mostriamo comunque l'implementazione generale della funzione creaArray dell'esempio precedente.

*Esempio:*

```
int* creaArrayDinamico(int n) {
    int* risultato = (int*) malloc(n*sizeof(int));
    // Alloca n interi contigui
    // Accessibile come se fosse un array

    return risultato;
}
```

Si ricordi che l'invocazione alla funzione malloc alloca uno spazio di memoria di $N$ byte contigui, per $N$ pari al valore del parametro, e restituisce un puntatore alla prima locazione di tale spazio. Per quanto detto circa l'operatore [] applicato ai puntatori, è possibile visitare gli elementi memorizzati in questo spazio come se comparissero all'interno di un array.

### 5.1.7. Riepilogo: come dichiarare un array

Il seguente codice mostra tutte le modalità di dichiarazione di un array viste fino a questo punto.

**crea-array.c**

```
#include <stdio.h>
#include <stdlib.h>
#define dimdef 10
const int dimconst = 10;

int * crearray (int d) {
  return (int *) malloc(sizeof(int)*d);
}

int * crearrayAppeso (int d) {
  int a[d];
  return a;
}

int main(){
  // allocazione statica
  int A[dimdef];
  int AA[dimconst];
  // allocazione stack run-time
  int n = 0;
  printf("dimensione dell'array: ");
  scanf("%d", &n);
  int b[n];
  // allocazione dinamica
  int * bb;
  bb = (int*) malloc(n*sizeof(int));
  // allocazione dinamica tramite funzione
  int * c;
  c = crearray(n);
  // allocazione stack run-time tramite funzione -- NO
  int * cc;
  cc = crearrayAppeso(n);

  for (int i=0; i<n; i++) {
    printf("prossimo elemento: ");
    scanf("%d",&b[i]);
  }
```

```
    printf("array letto\n");
    for (int i=0; i<n; i++) {
      printf("%d ",b[i]);
    }
    printf("\n");
    return 0;
}
```

### 5.1.8. Gestione dinamica della memoria

Il meccanismo di dichiarazione delle variabili visto finora permette di associare ad una variabile una quantità di memoria fissa e nota a tempo di compilazione. Ad esempio, ad eccezione del C99/C++ che permettono la dichiarazione di array con espressioni generiche, la dimensione degli array deve essere ottenuta come espressione costante, ovvero il cui valore sia noto a tempo di compilazione e non modificabile durante l'esecuzione del programma.

Per indicare che uno spazio di memoria viene allocato con una dimensione nota a tempo di compilazione, si usa il termine *allocazione statica*. L'esempio seguente mostra quanto sia importante poter allocare strutture dati di dimensione non nota a tempo di compilazione.

*Esempio:* L'intento del seguente programma è di creare un array di dimensione definita dall'utente e di popolarlo con dei caratteri da esso inseriti.

```
int n;
printf("Inserisci un intero: ");
scanf("%d\n", &n);
char a[n];// ERRORE: n non e' costante
          // (consentito in C99/C++)
for(int i = 0; i < n; i++){
    // Popola l'array con dei caratteri
    printf("Inserisci un carattere: ");
    scanf("%c\n", &a[i]);
}
```

Il programma genera un errore in compilazione dovuto al fatto che la dimensione dell'array a specificata nella dichiarazione è indicata da un'espressione non costante.

#### 5.1.8.1. Allocazione dinamica di array: la funzione calloc

Un modo semplice per allocare un array dinamicamente consiste nell'invocare la funzione malloc passandogli come parametro la dimensione dell'array. *Esempio:* Il seguente frammento di codice alloca un array di 100 interi.

```
int n_elementi = 100;
int* a = (int *)malloc(n_elementi * sizeof(int));
```

L'aritmetica dei puntatori e la possibilità di usare l'operatore di subscripting per accedere ai diversi blocchi di un'area di memoria permettono al programmatore di trattare la variabile a come se fosse di tipo array.

Un'alternativa a malloc è la funzione calloc, che ha la seguente segnatura:

```
void* calloc(size_t n_elementi, size_t size)
```

La funzione alloca un vettore di n_elementi elementi, ciascuno di dimensione size, ne inizializza gli elementi a 0 e restituisce il puntatore al primo elemento. *Esempio:* Il seguente frammento di codice alloca un array di 100 interi con calloc e ne inizializza tutti gli elementi a 0.

```
int n_elementi = 100;
int* a = (int *)calloc(n_elementi, sizeof(int));
         // inizializza gli elementi a 0
```

#### 5.1.8.2. Ridimensionamento di un'area di memoria allocata: la funzione realloc

Un'altra importante funzionalità disponibile solo nel caso di allocazione dinamica è il *ridimensionamento* dello spazio di memoria a tempo di esecuzione. La funzione C che permette di ridimensionare uno spazio di memoria allocato è la seguente:

```
void* realloc(void *p, size_t size)
```

realloc prende in input un puntatore p e un intero senza segno size, alloca size byte di memoria copiandovi il contenuto dello spazio

puntato da p (fin dove possibile, se la memoria è stata ridotta), dealloca la memoria del puntatore in input, restituisce un puntatore al nuovo blocco.

Il puntatore p deve far riferimento al primo blocco di un'area di memoria precedentemente allocata (con malloc, realloc o calloc). Se p è NULL, il comportamento di realloc è analogo a quello di malloc. Un valore di size pari a 0 comporta la deallocazione dello spazio. Infine, la funzione restituisce l'indirizzo NULL se il ridimensionamento non è possibile.

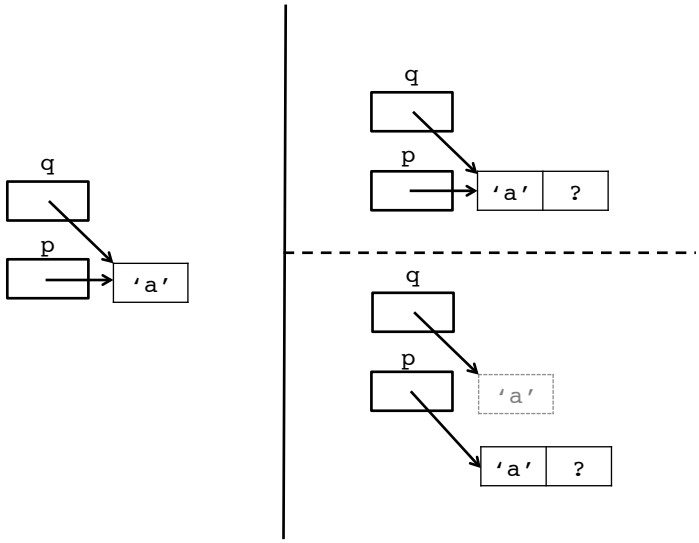
Mentre in generale non è garantito che il primo blocco della nuova area di memoria corrisponda a quello vecchio, ovvero che la vecchia area sia stata estesa ma non spostata, in pratica ciò accade spesso. È tuttavia sempre necessario aggiornare tutti i puntatori alla vecchia area di memoria con il nuovo indirizzo restituito dalla funzione, in quanto, se il blocco viene spostato, la vecchia area di memoria viene deallocata.

*Esempio:* In questo esempio, la dimensione dell'area puntata da p viene raddoppiata.

```
char *p = (char*) malloc(1);
*p = 'a';
char* q = p;
p = realloc(p,2);
// *(q+1) = 'b'; // ERRORE: L'area potrebbe essere
    cambiata!
q = p; // Aggiorno tutti i riferimenti alla vecchia
    area
*(q+1) = 'b'; // OK! Ora si puo' accedere
```

Poiché l'invocazione a realloc potrebbe aver allocato un nuovo spazio di memoria, prima di utilizzare q è necessario assicurarsi che esso punti effettivamente alla nuova area puntata da p. Questo è lo scopo dell'assegnazione q = p dopo l'invocazione a realloc.

La figura seguente mostra lo stato della memoria immediatamente prima dell'invocazione a realloc (sinistra) e le due alternative possibili immediatamente dopo (destra). Come si vede nella parte inferiore della figura destra, se la funzione alloca una nuova area di memoria, il puntatore q diventa pendente, in quanto la vecchia area viene contestualmente deallocata.




La possibilità di ridimensionare un'area di memoria risulta di particolare utilità nel caso in cui l'area contenga un array.

*Esempio:* Il seguente programma legge una serie di caratteri inseriti dall'utente e li memorizza in un array, finché non viene inserito il carattere ;. Quando l'array è pieno, la sua dimensione viene incrementata di n. Al termine dell'inserimento, l'array viene ridimensionato in modo da contenere solo gli elementi effettivamente usati ed il suo contenuto viene stampato.

```
int n = 5, i = 0;
char *p = (char*) calloc(n,sizeof(char));
char prox_char;
do{
    printf("Inserisci un carattere (; per uscire): ");
    scanf(" %c",&prox_char);
    if (i>=n){
        // array pieno: incrementa la dimensione n
        n+=n;
        p = (char*) realloc(p,n*sizeof(char));
    }
    p[i] = prox_char;
    i++;
} while(prox_char != ';');


/* Ridimensiona l'array al numero di caratteri
 effettivamente memorizzati: */
p = (char*) realloc(p,i*sizeof(char));

// Stampa:
for (int j = 0; j < i; j++){
    printf("%c",p[j]);
}
printf("\n");
```

#### 5.1.8.3. Esempio: restituire il puntatore ad un nuovo array

Come già discusso in precedenza, possiamo anche restituire un array allocato dinamicamente durante l'esecuzione di una funzione. Consideriamo la funzione copiaInverso che prende in ingresso un array v insieme alla sua dimensione n e, senza modificarlo, restituisce un *nuovo* array contenente gli stessi elementi di v in ordine inverso.

```
int* copiaInversa(int v[], int n) {
    int* risultato = (int*) calloc(n, sizeof(int));
    for (int i=0; i<n; i++) {
        risultato[n-1-i] = v[i];
    }
    return risultato;
}
```

Osserviamo che la variante, vista in precedenza, in cui viene usata malloc è altrettanto valida.

*Esempio d'uso:*

```
int main () {
    const int n=5;
    int x[n] = { 5, 3, 9, 5, 12 };
    int *y = copiaInversa(x,n);
    for (int i=0; i<n; i++)
        printf("%d ", y[i]);
    printf("\n");
}
```

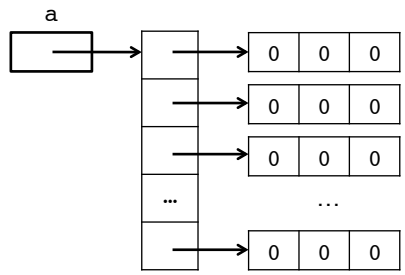
### 5.1.9. Array di puntatori

L'uso combinato di array e puntatori, unito alla possibilità di allocare memoria dinamicamente, permette di costruire strutture dati particolarmente utili come gli *array di puntatori*. Al pari di qualunque altro tipo, infatti, anche i puntatori possono essere presi come tipo base nella costruzione di array.

*Esempio:* Si assuma di dover memorizzare le coordinate di $N$ punti nello spazio cartesiano 3D. Ciascun punto può essere memorizzato in un array di 3 reali (float), mentre l'insieme di punti può essere memorizzato in un array di puntatori, in cui ciascun elemento punta ad uno degli array. Il seguente frammento di codice mostra la costruzione della struttura necessaria a memorizzare i punti.

```
int n;
printf("Quanti punti vuoi memorizzare?\n");
scanf(" %d",&n);
float** a = calloc(n,sizeof(float*));

for(int i = 0; i < n; i++){
    a[i] = calloc(3,sizeof(float));
}
```

La figura seguente illustra la memoria dopo l'esecuzione del frammento di codice.




Nell'esempio, la variabile a è dichiarata come puntatore a puntatore a float. Essa infatti punta ad un array i cui elementi sono di tipo puntatore a float, ovvero float*. Di conseguenza la componente *i*-esima di a, cioè a[i], è un puntatore a float. Tale puntatore viene inizializzato all'indirizzo del primo blocco dell'array restituito dall'invocazione a calloc effettuata all'*i*-esima iterazione del ciclo for (si ricordi che calloc inizializza tutte le componenti del vettore allocato a 0).

L'accesso alle componenti di ciascun array avviene attraverso l'operatore di subscripting. Occorre tuttavia tenere presente che il riferimento all'array *i*-esimo memorizzato in a è ottenuto tramite l'espressione a[i]. Pertanto, la *j*-esima componente dell'*i*-esimo array di a può essere ottenuta tramite l'espressione a[i][j].

*Esempio:* L'esempio precedente può essere modificato come segue per permettere all'utente di inserire le coordinate di ciascun punto, stampare i dati inseriti e, al termine dell'elaborazione, rilasciare la memoria.

```
int n;
printf("Quanti punti vuoi memorizzare?\n");
scanf(" %d",&n);
float** a = calloc(n,sizeof(int*));

for(int i = 0; i < n; i++){
    a[i] = calloc(3,sizeof(float));
    printf("Inserisci coordinate del punto %d: ",i);
    scanf("%f%f%f",&(a[i][0]),&(a[i][1]),&(a[i][2]));
}

for (int i = 0; i < n; i++){
    printf("P%d = (%f,%f,%f)\n",i,a[i][0],a[i][1],a[i
    ][2]);
}

//... elaborazione

// Deallocazione array profondi:
for(int i = 0; i < n; i++)
    free(a[i]);
// Deallocazione a:
free(a);

//... elaborazione
```

Si noti, nel primo ciclo for, l'uso dell'operatore & per ottenere l'indirizzo della componente *j*-esima dell'array *i*-esimo, da fornire in input a scanf.

Si osservi inoltre che il rilascio della memoria deve avvenire in due fasi: una prima in cui viene rilasciata la memoria occupata dagli array più "in profondità" nella struttura ed una seconda in cui viene rilasciata la memoria dell'array superficiale. Se venisse prima deallocata la memoria occupata dall'array a, si perderebbero infatti i riferimenti agli array più profondi, che non potrebbero quindi essere né usati né deallocati.

## 5.2. Stringhe

### 5.2.1. Variabili di tipo stringa in C

Il C non mette a disposizione un tipo speciale per memorizzare stringhe. Semplicemente, una stringa è memorizzata come un array di caratteri terminante con il carattere speciale `'\0'` (codice ASCII = 0), detto *terminatore di stringa*.

*Esempio:* Nel seguente frammento di codice la stringa 'Hello' viene memorizzata nella variabile s di tipo array di caratteri.

```
const int N=256;
char s[N];
s[0]='H'; s[1]='e'; s[2]='l';
s[3]='l'; s[4]='o';
s[5]='\0'; // terminatore stringa
printf("%s\n",s); // stampa Hello
```

Nell'esempio, il contenuto dell'array viene stampato usando la funzione printf. La specifica di formato %s indica alla funzione che l'argomento corrispondente deve essere trattato come una stringa, cioè che esso è un array di caratteri. printf leggerà l'array in sequenza, partendo dal primo elemento e fermandosi *solo* quando incontra il terminatore di stringa. Il fatto che la dimensione dell'array sia maggiore rispetto a quella della stringa non rappresenta un problema: la stringa rappresentata dall'array è costituita dai caratteri inclusi tra la prima posizione e quella contenente il terminatore di stringa.

Il terminatore di stringa è sempre obbligatorio, anche se l'array ha la stessa dimensione della stringa memorizzata. Nell'esempio precedente, se si omettesse di assegnare il terminatore di stringa ad s[5], la funzione printf stamperebbe tutti i caratteri dell'array (quelli successivi al quinto sono indefiniti), producendo `Hello??????????????...`, fino a raggiungere l'ultimo, superarlo e quindi generando un errore a tempo di esecuzione. La funzione, infatti, non incontrando il terminatore di stringa, andrebbe a leggere locazioni di memoria al di fuori dello spazio allocato per l'array.

### 5.2.2. Stringhe e puntatori a char

Essendo le stringhe essenzialmente array, per quanto detto circa la relazione tra puntatori ed array, è sempre possibile accedere ad un vettore contenente una stringa mediante un puntatore di tipo char*.

*Esempio:*

```
char s[10];
. . . // inizializzazione di s
char* p = s; // p punta al primo elemento di s
```

Poiché, come visto, il passaggio di parametri di tipo array avviene essenzialmente tramite puntatori, molte funzioni che prendono in input

stringhe specificano il rispettivo parametro come tipo char* invece di char[]. Inoltre, si può sempre usare un puntatore di tipo char* in luogo di un array di char (char[]) nell'invocazione di una funzione.

### 5.2.3. Dimensione delle stringhe in C

Negli esempi precedenti si noti la differenza tra la dimensione dell'array e la lunghezza della stringa che esso contiene. La prima rappresenta il numero di elementi dell'array ed è determinata staticamente al momento della sua dichiarazione, mentre la seconda rappresenta il numero di caratteri contenuti nella stringa rappresentata. In particolare, la dimensione dell'array è 256, mentre la lunghezza della stringa è 5. Il carattere di terminazione non è considerato appartenente alla stringa.

Essendo il terminatore di stringa obbligatorio, la lunghezza della stringa rappresentata da un array è sempre strettamente minore della dimensione dell'array. In caso contrario possono verificarsi accessi fuori della zona di memoria allocata per l'array, con conseguenti errori.

Il calcolo della lunghezza di una stringa, cioè il conteggio dei caratteri che precedono il terminatore di stringa, può essere effettuato tramite la funzione `strlen` definita nel file string.h (v. dopo). Poiché tale funzione richiede un ciclo di scansione dell'intera stringa, non è raro, per ragioni di efficienza, l'uso di una variable intera per memorizzare la lunghezza di una stringa.

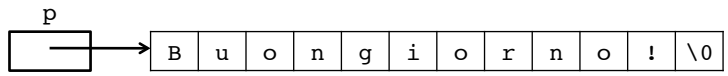
### 5.2.4. Stringhe letterali e inizializzazione di variabili stringa

Un letterale che denota una stringa è una sequenza di caratteri alfanumerici racchiusi tra apici doppi, ad esempio: "Hello world!". In C, quando viene incontrato un letterale di questo tipo, viene allocato un array costante di dimensione pari alla dimensione della stringa più uno (per memorizzare il terminatore) e viene inizializzato con i caratteri della stringa ed il terminatore. Il letterale viene trattato come un puntatore (di tipo const char*) al primo carattere dell'array.

*Esempio:* La seguente dichiarazione alloca un array costante di char di dimensione 12 ed assegna il puntatore al suo primo elemento a p.

```
char* p = "Buongiorno!";
```

Lo stato della memoria dopo l'esecuzione di questa istruzione è riportato nella figura seguente:




Nota: l'assegnazione di una stringa ad un puntatore char provoca un warning in C++.

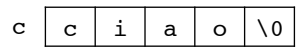
Come visto, una variabile di tipo stringa è in realtà un vettore di caratteri che può, quindi, essere inizializzato tramite espressioni come: char c[5] = {'c','i','a','o','\0'}. Si osservi che per rappresentare una stringa è necessario che l'array contenga il terminatore. In altre parole, con la dichiarazione char c[5] = {'c','i','a','o'} si sta trattando c semplicemente come un array di caratteri.

Tramite l'uso di letterali, il C mette a disposizione una forma semplice d'inizializzazione.

*Esempio:* Nel seguente frammento di codice l'array dichiarato viene popolato con i caratteri della stringa rappresentata dal letterale, con il terminatore di stringa come ultimo elemento.

```
char c[5] = "ciao";
```

Lo stato della memoria dopo l'esecuzione di questa istruzione è riportato nella figura seguente:




La dichiarazione mostrata sopra è equivalente a

```
char c[5] = { 'c', 'i', 'a', 'o', '\0' };
```

#### 5.2.4.1. Esempio: occorrenze di un carattere in una stringa

Realizziamo una funzione che, presi come parametri una stringa (sotto forma di array di caratteri) ed un carattere c, restituisce il numero di occorrenze di c nella stringa.

**conta-carattere.c**

```
int contaCarattere(char s[], char c) {
    int quanti = 0;
    int pos = 0;
    while (s[pos] != '\0') {
        if (s[pos] == c)
            quanti++;
        pos++;
    }
    return quanti;
}
```

Si osservi come, assumendo che la funzione prenda in input un array contenente una stringa, non sia necessario indicare la dimensione dell'array. Infatti, il terminatore di stringa garantisce che il ciclo while termini prima che la variabile pos superi l'indice dell'ultima componente dell'array.

### 5.2.5. Stringa vuota

La stringa vuota rappresenta la sequenza di caratteri di lunghezza 0. Essa è memorizzata come un array di char contenente il terminatore di stringa come primo elemento. La stringa vuota si può denotare con il letterale `""`. Si faccia attenzione a non confondere la stringa vuota con il valore NULL. La prima è un array di caratteri non vuoto (contenente il terminatore come primo carattere) mentre il secondo è il valore di un puntatore.

### 5.2.6. Esempio: codifica di una stringa

Realizziamo una funzione che, presi come parametri due stringhe str e strRis (come array di caratteri) ed un intero d, restituisce in strRis la stringa ottenuta sostituendo ciascun carattere c di str con il carattere il cui codice ASCII è pari al codice di c incrementato di d. La funzione deve inoltre restituire un puntatore di tipo char* alla stringa contenente il risultato.

```
#include <string.h> // per strlen

char* codificaStringa(const char *str, char strRis[],
    int d) {
    int n = strlen(str);
    for (int i = 0; i < n; i++)
        strRis[i] = d + str[i];
    strRis[n]='\0'; // terminatore di stringa
    return strRis;
}
```

*Esempio d'uso:*

```
int main(){
  char s[5]="ciao";
  char t[5];
  printf("%s\n",codificaStringa(s,t,1));// stampa djbp
}
```

Si osservi che il primo parametro della funzione codificaStringa svolge il ruolo di input, ed è quindi dichiarato const per prevenire modifiche al suo contenuto, mentre il parametro strRis non può essere dichiarato const in quanto verrà modificato.

Inoltre, si noti che codificaStringa usa la funzione strlen, la quale restituisce la lunghezza della stringa (non dell'array). Pertanto, il ciclo for termina quando l'ultimo carattere diverso dal terminatore è stato letto ed è quindi necessario, per garantire che strRis rappresenti effettivamente una stringa, inserire il terminatore di stringa.

Infine, notiamo che la restituzione del puntatore all'array contenente il risultato della funzione codificaStringa permette di usare il risultato della funzione direttamente all'interno dell'invocazione di printf, senza dover prima eseguire la funzione e successivamente stampare l'array contenente il risultato (come sarebbe necessario se il tipo restituito dalla funzione fosse void).

### 5.2.7. Esempio: lunghezza della più lunga sottosequenza

Realizzare una funzione che prende in ingresso una stringa `s` (sotto forma di array di caratteri) costituita dai soli caratteri `'0'` e `'1'`, e restituisce la lunghezza della più lunga sottosequenza di `s` costituita da soli `'0'` tutti consecutivi. Ad esempio, se la stringa passata come parametro

è 001000111100, allora la più lunga sottosequenza di soli '0' è quella sottolineata, che ha lunghezza 3.

**sottosequenza.c**

```c
#include <string.h> // per strlen

int sottosequenza(const char * s) {
    char bit;           // l'elemento corrente della
    sequenza
    int cont = 0;       // lunghezza attuale della
    sequenza di zeri;
    int maxlung = 0;    // valore temporaneo della
    massima lunghezza;
    int N = strlen(s);  // lunghezza della stringa
    for (int i = 0; i < N; i++) {
        bit = s[i];
        if (bit == '0') {// e' stato letto un altro
    '0'
            cont++;// aggiorna la lunghezza della
    sequenza corrente
            if (cont > maxlung)  // se necessario, ...
                // ... aggiorna il massimo temporaneo
                maxlung = cont;
        }
        else // e' stato letto un '1'
            cont = 0;// azzera la lunghezza della
    sequenza corrente
    }
    return maxlung;
}
```

### 5.2.8. Stampa e lettura di stringhe in C

#### 5.2.8.1. Stampa

La stampa di stringhe in C può essere effettuata tramite le funzioni printf e puts.

Per usare la prima occorre conoscere la specifica di formato per la formattazione delle stringhe: %s.

Per quanto riguarda la seconda, è sufficiente sapere che essa ha un solo argomento di tipo char *, il puntatore al primo carattere della stringa da stampare, e stampa i caratteri della stringa, seguiti da un ritorno a capo.

*Esempio:*

```
char* s = "Stringa da stampare";
puts(s); // stampa: ''Stringa da stampare'' e torna a
    capo
```

#### 5.2.8.2. Lettura

Per la lettura di stringhe da tastiera (o, più in generale da standard input) si possono usare le funzioni scanf e gets.

La specifica di formato da usare nell'invocazione di scanf è %s, mentre il parametro usato per memorizzare la stringa letta deve essere di tipo char*. Nel caso si usi un array, non è necessario anteporre il carattere &, in quanto l'identificatore dell'array rappresenta già un puntatore di tipo char*.

*Esempio:* Il seguente frammento di codice memorizza nella variabile di tipo stringa s una stringa fornita in input dall'utente.

```
char s[256];
printf("Inserisci una stringa \n");
scanf("%s",s);
```

La funzione scanf considera gli spazi bianchi come caratteri di ritorno a capo. Pertanto, con la stringa "prova stringa" l'esecuzione di scanf nel programma precedente terminerebbe dopo la prima parola, "prova", che sarebbe quindi l'unica stringa memorizzata in s (per memorizzare la parte restante sono necessarie altre invocazioni di scanf). scanf si occupa di inserire il terminatore di stringa dopo l'ultimo carattere della stringa letta.

Per quanto riguarda la funzione gets, essa ha un solo parametro di tipo char*, che rappresenta il puntatore al primo elemento dell'array in cui memorizzare la stringa letta. Diversamente da scanf, la funzione gets considera gli spazi bianchi come dei caratteri qualunque, terminando la lettura solo quando incontra un ritorno a capo. In altre parole, gets legge e memorizza una riga intera. Anche gets inserisce il terminatore di stringa.

*Esempio:*

```
printf("Inserisci una stringa \n");
char s[256];
gets(s);
printf("%s\n",s);
```

Sia scanf che gets memorizzano la stringa letta nell'array passato come parametro. Tali funzioni non effettuano alcun controllo sulla dimensione dell'array, in particolare se essa sia sufficiente a memorizzare l'intera stringa letta. Nel caso in cui ciò non accada, queste funzioni semplicemente scrivono i caratteri eccedenti nelle locazioni successive all'array, tipicamente causando errori a runtime. [2]

### 5.2.9. Funzioni comuni della libreria per le stringhe (string.h)

Il C, come parte della libreria standard, mette a disposizione un insieme di funzioni per la manipolazione di stringhe. Tali funzioni sono dichiarate nel file header string.h quindi, per poterle usare, è necessario inserire la direttiva #include <string.h>. Di seguito descriviamo alcune funzioni di uso comune della libreria.

**size_t strlen(char *str):** restituisce la lunghezza della stringa passata come parametro (size_t è un tipo definito, corrispondente essenzialmente ad un intero senza segno).

**int strcmp(const char *str1, const char *str2):** confronta due stringhe in base all'ordinamento lessicografico (v. sotto), restituendo: un valore negativo se str1 precede str2; 0 se str1 è uguale a str2; un valore positivo se str1 segue str2.

**char *strcpy(char *dest, const char *src):** copia la stringa src in dest (su cui fa side-effect) e restituisce un puntatore a dest.

**char *strcat(char *str1, const char *str2):** concatena str2 alla fine di str1 (su cui fa side-effect) e restituisce un puntatore a str1.

**char *strstr(const char *str1, const char *str2):** restituisce il puntatore all'elemento iniziale della prima occorrenza della stringa str2 in str1, oppure NULL se str2 non compare come sotto-stringa di str1.

[2] Per questa ragione, con alcun compilatori, quando si usa la funzione gets, l'esecuzione del programma produce il messaggio: warning: this program uses gets(), which is unsafe.

#### 5.2.9.1. Ordine lessicografico di stringhe

Per definire l'*ordine lessicografico* delle stringhe, occorre innanzitutto stabilire un ordine tra i caratteri. Questo corrisponde all'ordine indotto dai rispettivi codice ASCII di ciascun carattere. In base ad esso abbiamo che:

- l'ordine lessicografico delle lettere alfabetiche corrisponde a quello alfabetico;
- le cifre precedono le lettere;
- le maiuscole precedono le minuscole.

Possiamo a questo punto definire l'ordine lessicografico tra stringhe. Una stringa $s$ precede una stringa $t$, se:

- $s$ è un prefisso di $t$, oppure
- se $c$ e $d$ sono il primo carattere rispettivamente di $s$ e $t$ in cui $s$ e $t$ differiscono, allora $c$ precede $d$ nell'ordinamento dei caratteri.

*Esempio:*

- `auto` precede `automatico`
- `Automatico` precede `auto`
- `albero` precede `alto`
- `H2O` precede `HOTEL`

### 5.2.10. Esempi d'uso delle funzioni per stringhe

Il seguente frammento di programma usa alcune funzioni della libreria <string.h> per eseguire delle operazioni su stringhe.

**esempi-funzioni-stringhe.c**

```
#include <string.h>
#include <stdio.h>

int main(){
    const int N=256;
    char nome[N];
    printf("Inserisci il tuo nome: ");
    gets(nome);
    printf("Il tuo nome contiene %lu caratteri\n",
    strlen(nome));
```

```
    if (strcmp(nome,"Mario")==0)
        printf("Ciao Mario, come stai?\n");
    else
        if (strcmp(nome,"Mario")<0)
            printf("Il tuo nome precede Mario.\n");
    else
        printf("Il tuo nome segue Mario.\n");

    char cognome[N];
    printf("Inserisci il tuo cognome: ");
    gets(cognome);
    strcat(nome," ");
    strcat(nome,cognome);
    printf("Il tuo nome completo e' %s\n", nome);
    if (strcmp(nome,"Mario Rossi")!=0)
        printf("Tu non sei Mario Rossi!\n");
    char* sottostringa = strstr(nome,"Ro");
    if (sottostringa != NULL){
        printf("Il tuo nome completo contiene \"Ro\":
    %s\n",
                sottostringa);
    }
}
```

### 5.2.11. Passaggio di parametri e risultato di una funzione

Essendo le stringhe un caso speciale di array, il passaggio di parametri e la restituzione di un risultato di tipo stringa da parte di una funzione avvengono secondo gli stessi meccanismi illustrati nel caso degli array.

Si noti, tuttavia, che grazie al terminatore di stringa è possibile conoscere la dimensione della stringa senza doverla passare esplicitamente come parametro, come invece accade per gli array.

### 5.2.12. Parametri passati ad un programma

La funzione main può essere dichiarata anche con una segnatura a due argomenti:

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

Gli argomenti della funzione main indicano un array di stringhe argv (rappresentate mediante array di caratteri) e il numero di elementi dell'array argc. Il primo argomento, cioè argv[0] corrisponde al nome del file eseguibile. Gli argomenti di un programma vengono forniti in input da linea di comando al momento della sua invocazione.

*Esempio:* Il seguente programma stampa gli argomenti passatigli in input al momento della sua invocazione.

**parametri-main.c**

```
#include <stdio.h>

int main(int argc, char** argv){
    for (int i = 0; i < argc; i++){
        printf("Parametro n.%d: %s\n", i,argv[i]);
    }
}
```

Supponendo di aver generato il file eseguibile myprog, un esempio di linea di comando che fornisce gli argomenti in input al programma è la seguente:

```
> ./myprog primo secondo terzo
```

Il programma stampa:

```
Parametro n.0: ./myprog
Parametro n.1: primo
Parametro n.2: secondo
Parametro n.3: terzo
```

## 5.3. Matrici

Una **matrice** è una collezione in forma tabellare di elementi dello stesso tipo, ciascuno indicizzato da una coppia di interi positivi (0 incluso) che ne identificano riga e colonna nella tabella.

*Esempio:* Di seguito è riportata una matrice $M$ di 3 righe e 4 colonne. Gli indici di riga crescono verso il basso e quelli di colonna verso l'alto. L'indice della prima riga e della prima colonna è 0. L'elemento generico con indice di riga $i$ e di colonna $j$ è denotato $M[i, j]$. Pertanto, ad esempio, abbiamo: $M[0, 0] = 1$, $M[1, 3] = 9$ e $M[2, 3] = 19$.

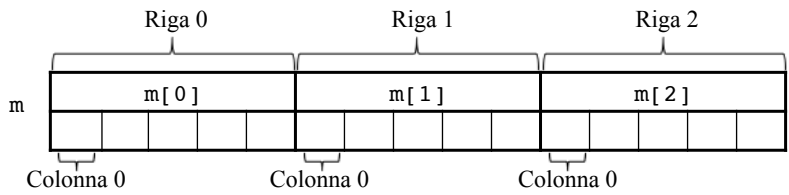
| | | | |
|---|---|---|---|
| 1 | 3 | 34 | 24 |
| 2 | 31 | 39 | 9 |
| 7 | 6 | 8 | 19 |

Una matrice di $N$ righe ed $M$ colonne è memorizzata mediante un array di array dello stesso tipo della matrice, ciascuno contenente gli elementi di una riga.

*Esempio:* Nel seguente frammento di codice viene dichiarata una matrice m di 3 righe e 5 colonne

```
const int N = 3, M = 5;
int m[N][M]; // dichiarazione di una matrice NxM m
```

La figura seguente mostra la rappresentazione interna della matrice m appena dichiarata.



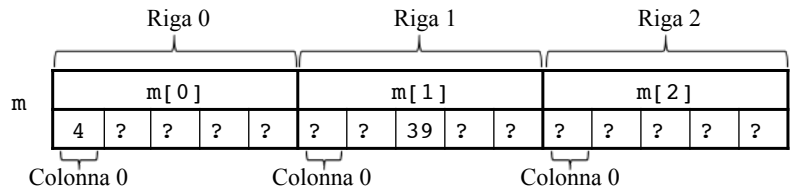

L'array $i$-esimo di m (contenente la riga $i$-esima della matrice) è denotato dall'espressione `m[`$i$`]`. Pertanto, l'elemento con indice di riga $i$ e colonna $j$ è denotato in C dall'espressione m[$i$][$j$]. L'identificatore della matrice rappresenta un puntatore all'array m[0].

La lettura e la scrittura delle componenti di una matrice avvengono in maniera essenzialmente analoga a quanto visto per gli array.

*Esempio:* Con riferimento alla matrice m definita nell'esempio precedente, assegniamo dei valori ad alcune sue componenti.

```
// assegnazione dell'elemento della matrice m
// alla riga 1, colonna 2
m[1][2] = 39;
// assegnazione dell'elemento della matrice m
// alla riga 0, colonna 0
m[0][0] = 4;
printf("%d\n", m[1][2]); // stampa 39
```

La figura seguente mostra lo stato della memoria dopo l'esecuzione del frammento di codice.




### 5.3.1. Inizializzazione di matrici tramite espressioni

Le matrici possono essere inizializzate mediante espressioni, in maniera simile agli array.

*Esempio:* Definiamo ed inizializzamo la matrice m usando espressioni.

```
int m[][2] = {
 {3,5},
 {9,12}
};
```

Nella definizione, solo la prima dimensione della matrice, ovvero il numero di righe, può essere non specificata.

La definizione di matrice sopra mostrata è del tutto equivalente al seguente frammento:

```
int m[2][2];
m[0][0] = 3; m[0][1] = 5;
m[1][0] = 9; m[1][1] = 12;
```

### 5.3.2. Numero di righe e colonne di una matrice

Usando la funzione sizeof è possibile ottenere il numero di elementi contenuti in una matrice ed il suo numero di righe e colonne. *Esempio:*

```
int x[][2]= {{3,5}, {9,12}, {8,10}};
int n_bytes = sizeof(x); // 24 (6 interi x 4 bytes
)
int n_elementi = n_bytes / sizeof(int); // 6
```

Per conoscere il numero di colonne è sufficiente calcolare il numero di elementi di una riga qualsiasi (ovvero del corrispondente array), ad esempio:

```
int n_colonne = sizeof(x[0])/sizeof(int); // 2
```

Infine, il numero di righe può essere ottenuto semplicemente dividendo il numero di elementi per il numero di colonne:

```
int n_righe = n_elementi / n_colonne; // 3
```

#### 5.3.2.1. Esempio: stampa di una matrice per righe e per colonne

La visita degli elementi di una matrice può essere facilmente realizzata facendo uso di due cicli annidati.

Il seguente frammento stampa una matrice x di n_righe righe ed m_colonne colonne, procedendo per righe, e ne stampa il contenuto (una riga per linea di stampa.

```
for (int i=0; i<n_righe; i++) {
    for (int j=0; j<n_colonne; j++)
        printf("%d ", x[i][j]);
    printf("\n");
}
```

Si noti che per accedere a tutti gli elementi della matrice x vengono usati due cicli for annidati: quello esterno legge le righe (i), quello interno legge gli elementi di ogni riga (j). Quando tutti gli elementi di una riga sono stati stampati, viene stampato il carattere di ritorno a capo.

La stampa per colonne può essere effettuata nel modo seguente (una colonna per linea di stampa).

```
for (int i=0; i<n_colonne; i++) {
    for (int j=0; j<n_righe; j++)
        printf("%d ", x[j][i]);
    printf("\n");
}
```

Anche in questo caso vengono usati due cicli for annidati: quello esterno scorre le colonne (i), quello interno scandisce gli elementi di ogni colonna (j).
*Esempio d'uso:*

```
int main () {
    const int n_righe=2;
    const int n_colonne=3;
    int m[][n_colonne]= {{1, 2, 2}, {7, 5, 9}};
    printf( "stampa matrice per righe\n");
    for (int i=0; i<n_righe; i++) {
        for (int j=0; j<n_colonne; j++)
            printf("%d ",m[i][j]);
        printf("\n");
    }
    printf("stampa matrice per colonne\n");
    for (int i=0; i<n_colonne; i++) {
        for (int j=0; j<n_righe; j++)
            printf("%d ",m[j][i]);
        printf("\n");
    }
    return 0;
}
```

Il programma stampa:

```
stampa matrice per righe
1 2 2
7 5 9
stampa matrice per colonne
1 7
2 5
2 9
```

#### 5.3.2.2. Esempio: somma di matrici

Sempre usando di cicli annidati, è possibile calcolare la somma di due matrici A e B aventi stesse dimensioni (numero di righe e stesso numero di colonne), ovvero la matrice C ottenuta sommando gli elementi corrispondenti (stessi indici di riga e colonna) delle due matrici.

```
for (int i = 0; i < n_righe; i++){
    for (int j = 0; j < n_colonne; j++){
        C[i][j] = A[i][j] + B[i][j];
    }
}
```

L'uso dei cicli annidati permette di accedere a tutti gli elementi delle matrici A, B (per effettuarne le somme) e C (per inizializzarne i valori). L'accesso agli elementi di ciascuna matrice segue l'ordine per righe (il ciclo esterno scorre le righe, quello interno gli elementi di ogni riga) ma la stessa operazione avrebbe potuto essere realizzata procedendo per colonne.

#### 5.3.2.3. Esempio: prodotto di matrici

Scriviamo un frammento di codice che definisce i valori di una nuova matrice C ottenuta come prodotto di A e B, assumendo che A e B siano quadrate, ovvero abbiano lo stesso numero N di righe e colonne.

Si ricordi che ogni elemento C[i][j] del prodotto di matrici $A \times B$ è ottenuto come prodotto scalare della riga i di A con la colonna j di B, cioè per ogni coppia di indici i,j, si ha: $\texttt{C[i][j]} = \sum_k (\texttt{A[i][k]} * \texttt{B[k][j]})$.

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++) {
        C[i][j] = 0;
        for (int k=0; k<N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

In questo caso occorrono tre cicli annidati: uno (esterno) per scorrere le righe (i) di C, uno (intermedio) per scorrere le colonne (j) di C ed uno (interno) per calcolare il prodotto scalare, da assegnare a C[i][j], della riga i di A con la colonna j di B.

### 5.3.3. Passaggio di parametri matrice

Per passare dei parametri di tipo matrice è necessario indicare, nell'intestazione della funzione, il numero di colonne della matrice. Il numero di righe è invece opzionale.

*Esempio:* La seguente funzione prende in input una matrice con un numero di righe passato come parametro (n_righe) ed un numero fisso di colonne pari a 3.

```
void stampaMatrice(float A[][3], int n_righe){
    for(int i = 0; i < n_righe; i++){
        for (int j = 0; j < 3; j++){
            printf("%f ",A[i][j]);
        }
        printf("\n");
    }
}
```

Si osservi che la funzione appena definita può trattare solo matrici con 3 colonne! Inoltre, se il numero di righe fosse stato definito nella specifica della matrice, usando la dichiarazione int A[2][3], anche il numero di righe sarebbe stato fissato.

Per quanto riguarda l'uso della funzione sizeof all'interno di una funzione che prende matrici in input, valgono considerazioni analoghe a quanto visto per gli array. In particolare, quando è necessario conoscere il numero di righe della matrice all'interno della funzione (se non specificato nell'intestazione), questo deve essere fornito in input come parametro.

#### 5.3.3.1. Esempio: somma per righe di una matrice

Realizziamo una funzione void sommaRigheMatrice(int A[][3], int N, int V[]) che assegna all'elemento di indice i dell'array V la somma degli elementi della riga i di M.

```
void sommaRigheMatrice (int A[][3], int N, int V[]) {
    for (int i=0; i<N; i++) {
        V[i]=0;
        for (int j=0; j<3; j++)
            V[i] += A[i][j];
    }
}
```

*Esempio d'uso:*

```
const int N=3;
int A [][N] = {     // crea matrice A con dimensione 3
    x3
    { 1, 2, 2 },    // riga 0 di A (array di 3 int)
    { 7, 5, 9 },    // riga 1 di A (array di 3 int)
    { 3, 0, 6 }     // riga 2 di A (array di 3 int)
  }
  int B[3];
  sommaRigheMatrice(A,N,B);
  for (int i=0; i<N; i++)
    printf("%d\n", B[i]); // stampa array
```

Il programma stampa:

```
5
21
9
```

### 5.3.4. Matrici come array di puntatori

La restrizione di dover indicare il numero di colonne rende l'uso delle matrici ristretto a casi particolari, in cui le dimensioni sono note a tempo di compilazione. Quando necessario, è possibile sfruttare gli array di puntatori per creare matrici di dimensioni note a tempo di esecuzione, passarle come parametri e farle restituire da funzioni. *Esempio:*

```
int righe = 10, colonne = 5;
int** m = (int**) calloc(righe,sizeof(int*));
for (int i = 0; i < righe; i++) {
    m[i] = (int*) calloc(colonne,sizeof(int));
}
```

#### 5.3.4.1. Modificare una matrice passata come parametro

Realizziamo una funzione che prenda in input una matrice mat di interi di n righe ed m colonne, rappresentata come array di puntatori, e faccia side-effect su di essa, raddoppiando il valore di ogni suo elemento.

```
void raddoppiaMatrice(int** mat, int n, int m){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            mat[i][j] *= 2;
        }
    }
}
```

```
int main(){
    int righe = 10, colonne = 5;
    int** m = (int**) calloc(righe,sizeof(int*));
    for (int i = 0; i < righe; i++) {
        m[i] = (int*) calloc(colonne,sizeof(int));
        for(int j = 0; j < colonne; j++){
            m[i][j] = i + j;
        }
    }
    raddoppiaMatrice(m,righe,colonne);
}
```

#### 5.3.4.2. Restituire una nuova matrice

Realizziamo una funzione che crea una matrice tramite allocazione dinamica di array di array inizializzata a 0.

```
int **creamatrice(int righe, int colonne) {
    int** m = (int**) calloc(righe,sizeof(int*));
    for (int i = 0; i < righe; i++) {
        m[i] = (int*) calloc(colonne,sizeof(int));
    return m;
}
```

Realizziamo una funzione che, presa in input una matrice mat di interi di n righe ed m colonne rappresentata come array di puntatori, restituisca una nuova matrice di m righe ed n corrispondente alla trasposta della matrice di input.

```
int** trasponiMatrice(int** mat, int n, int m){
    int** risultato = (int **) calloc(m,sizeof(int*));
    for(int i = 0; i < m; i++){
        risultato[i] = (int*) calloc(n,sizeof(int));
        for(int j = 0; j < n; j++){
            risultato[i][j] = mat[j][i];
        }
    }
    return risultato;
}
```

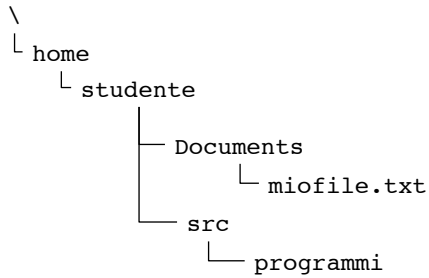
```
int main(){
    int righe = 10, colonne = 5;
    int** m = creaMatrice(righe,colonne);
    int** trasposta = trasponiMatrice(m,righe,colonne)
    ;
}
```

## 5.4. File

I *file* rappresentano la principale struttura per la memorizzazione di dati in maniera persistente (ovvero tali da essere mantenuti anche dopo lo spegnimento della macchina) su memoria di massa. Essi possono contenere caratteri alfanumerici codificati in formato standard (es. ASCII), direttamente leggibili dall'utente (*file di testo*), oppure dati in un formato interpretabile dai programmi (*file binari*). I file di testo sono normalmente organizzati in sequenze di linee ciascuna delle quali contiene una sequenza di caratteri. In questa unità tratteremo solamente file di testo.

Ogni file è caratterizzato dal nome e dalla directory (o cartella) in cui risiede. In C, un file è identificato dal suo percorso (assoluto o relativo).

*Esempio:* Si consideri la seguente organizzazione del filesystem:




Il percorso assoluto del file miofile.txt è:

- /home/studente/Documents/miofile.txt.

Il percorso relativo del file a partire dalla directory studente è:

- Documents/miofile.txt

Il percorso relativo del file a partire dalla directory programmi è:

- ../../Documents/miofile.txt

Il percorso relativo del file a partire dalla directory Documents è:

- ./miofile.txt

Le operazioni fondamentali sui file sono: creazione, lettura, scrittura, rinominazione ed eliminazione. Queste operazioni possono essere effettuate tramite il sistema operativo (ovvero un apposito interprete dei comandi) o mediante istruzioni del linguaggio C.

La libreria C che fornisce funzioni, tipi e costanti per la manipolazione di file è definita nel file stdio.h.

### 5.4.1. Operazioni sui file

Per eseguire operazioni di lettura e scrittura su file è necessario *aprire* il file prima di eseguire le operazioni e *chiudere* il file al termine dell'esecuzione delle operazioni.

- Aprire un file significa indicare al sistema operativo la volontà di eseguire operazioni sui file. Il sistema operativo verifica all'atto dell'apertura del file se tale operazione è possibile (controllando, ad esempio, che non vi siano altri programmi che stiano scrivendo sul file). L'apertura del file può avvenire in diverse modalità, tra cui

*apertura in lettura* e *apertura in scrittura*, che definiscono un comportamento differente nel sistema operativo (ad esempio è possibile che due applicazioni aprano lo stesso file contemporaneamente in lettura, ma non in scrittura).

- Chiudere un file significa indicare al sistema operativo che il file precedentemente aperto non è più usato dal programma. L'operazione di chiusura serve anche ad assicurarsi che i dati vengano effettivamente scritti sul disco.

### 5.4.2. Il tipo FILE

Per permettere la manipolazione di file, il C definisce il tipo FILE (come record). Tutte le operazioni su file avvengono tramite puntatori a strutture di questo tipo.

*Esempio:* Esempio di dichiarazione di un puntatore a FILE

```
FILE* pfile = NULL;
```

### 5.4.3. Apertura di un file di testo

La funzione C che permette l'apertura di un file è la seguente:

- `FILE* fopen(const char* filename, const char* mode)`

Questa funzione apre il file identificato dal percorso filename nella modalità indicata dalla stringa mode e restituisce un puntatore alla struttura di tipo FILE che lo identifica (NULL se non è stato possibile aprire il file). Il percorso può essere sia assoluto che relativo. Nel secondo caso, il percorso deve partire dalla directory di lavoro del programma in esecuzione (tipicamente, la directory da cui il programma è stato lanciato, a meno che non sia modificata all'interno del programma). Per quanto riguarda la modalità di apertura, per i file di testo sono disponibili le seguenti opzioni:

| | |
|---|---|
| r | lettura (default) |
| w | scrittura (sovrascrive, crea se non esiste) |
| a | accodamento (crea se non esiste) |
| r+ | lettura e scrittura da inizio file |
| w+ | lettura e scrittura (sovrascrive, crea se non esiste) |
| a+ | lettura e scrittura (accodamento, crea se non esiste) |

*Esempio:* Il seguente frammento di codice apre il file miofile.txt in lettura e scrittura (creandolo, se non esiste) ed assegna il puntatore al file alla variabile file:

```
FILE* file = fopen("/home/studente/Documents/miofile.
    txt", "w+");
```

### 5.4.4. Chiusura di un file di testo

La funzione C che permette la chiusura di un file è la seguente:

- `int fclose(FILE* file)`

Essa prende in input il puntatore ad una struttura di tipo FILE e chiude il file corrispondente. In caso di successo, la funzione restituisce il valore 0, altrimenti la costante EOF definita in stdio.h.

*Esempio:* Il seguente frammento di codice mostra il classico schema di accesso ad un file. Il file viene aperto, elaborato e al termine dell'elaborazione viene chiuso.

```
FILE* file = fopen("/home/studente/Documents/miofile.
    txt", "w+");
// ...accesso al file
fclose(file);
```

### 5.4.5. Scrittura di file di testo

Per scrivere stringhe in un file di testo occorre:

1. aprire il file e verificarne la corretta apertura
2. scrivere testo nel file
3. chiudere il file e verificarne la corretta chiusura

Ricordando che fopen ed fclose restituiscono rispettivamente il valore NULL e EOF in caso di insuccesso, gli esiti dell'apertura e della chiusura possono essere facilmente verificati tramite il test di una condizione (v. esempio seguente).

Per quanto riguarda l'output, le funzioni più comuni sono:

- int fprintf(FILE* file, const char* formato, ...)
- int fputs(const char* s, FILE* file)

La prima funzione è essenzialmente analoga a printf, ad eccezione del fatto che il suo primo parametro è un puntatore a FILE che rappresenta il file su cui essa andrà a stampare l'output.

*Esempio:* Il seguente frammento di codice stampa 3 righe in un file di testo. Se il file non esiste, lo crea, altrimenti ne sovrascrive il contenuto a partire dal primo carattere.

**scrivi-file.c**

```
FILE* file = fopen("/home/studente/Documents/miofile.
    txt", "w+");

if (file == NULL){
    printf("Errore nell'apertura del file\n");
    exit(1);
}

for(int i = 0; i < 3; i++){
    fprintf(file, "Questa e' la riga numero %d\n", i);
}

int ok = fclose(file);
if (ok != 0){
    printf("Errore nella chiusura del file\n");
    exit(1);
}
```

Si noti come la corretta apertura e chiusura del file siano verificate tramite il test di opportune condizioni.

L'esecuzione di questo frammento produce il seguente output nel file /home/studente/Documents/miofile.txt

```
Questa e' la riga numero 0
Questa e' la riga numero 1
Questa e' la riga numero 2
```

La funzione fputs si comporta in maniera analoga a puts, ovvero stampa la stringa seguita dal carattere di ritorno a capo, ma stampa l'output nel file passato come parametro.

#### 5.4.5.1. Esempio: scrittura su un file di input fornito da utente

Realizziamo un programma che accoda in un file, il cui percorso è preso da input, il testo inserito dall'utente. Se il file non esiste, deve essere creato. Il programma termina quando l'utente inserisce il carattere ;.

**appendi-testo.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv){
    char* nomefile = argv[1];

    FILE* file = fopen(nomefile,"a");
    if (file == NULL){
        printf("Errore nell'apertura del file\n");
        exit(1);
    }

    char stringa[256] = "";

    while(strcmp(stringa,";")!= 0){
        printf("Inserisci una stringa: ");
        gets(stringa);
        fprintf(file,"%s\n",stringa);
    }
    int close = fclose(file);
    if(close == EOF){
        printf("Errore nella chiusura del file\n");
        exit(1);
    }
}
```

Si notino l'uso del ciclo while, la cui condizione di terminazione corrisponde all'immissione da parte dell'utente del carattere ;, ed il controllo della corretta apertura e chiusura del file. Si osservi che in caso di errore il programma termina restituendo il valore (non nullo) 1.

### 5.4.6. Lettura da file di testo

Per leggere da file di testo occorre:

1. aprire il file e verificarne la corretta apertura
2. leggere il testo fino al punto desiderato
3. chiudere il file e verificarne la corretta chiusura

Per la lettura da file, due funzioni comunemente utilizzate sono le seguenti:

- int fscanf(FILE* file, const char* formato, ...)
- char* fgets(char* line, int n, FILE* file)

fscanf si comporta in maniera analoga a scanf, prendendo però come primo parametro il puntatore al file da cui leggere l'input. Raggiunta la fine del file, fscanf restituisce la costante EOF.

Analogamente fgets legge una stringa (fino al carattere di fine linea) dal file specificato.

*Esempio:* Il seguente frammento di codice legge il file miofile.txt parola per parola e ne stampa il contenuto su schermo.

**leggi-file.c**

```
FILE* file = fopen("/home/studente/Documents/miofile.
    txt", "r");

if (file == NULL){
    printf("Errore nell'apertura del file\n");
    exit(1);
}
char s[256];
while (fscanf(file,"%s",s) != EOF){
    printf("%s",s);
}
int close = fclose(file);
if (close == EOF){
    printf("Errore nella chiusura del file\n");
    exit(1);
}
```

La funzione fgets prende in input un puntatore ad array di caratteri (line), un intero n e il puntatore al file da cui si vuole leggere (file), legge la riga corrente del file, partendo dalla prima ed avanzando ad ogni invocazione, e la memorizza nell'array puntato da line. Il valore n indica il massimo numero di caratteri che line può contenere meno 1 (l'ultimo carattere è usato dal terminatore di stringa). Ad ogni invocazione, fgets legge o la riga corrente del file (se essa non contiene più di n-1 caratteri) oppure i prossimi n-1 caratteri. Quando viene raggiunta la fine del file, la funzione restituisce il puntatore NULL, altrimenti restituisce il valore del suo primo argomento (ovvero un riferimento alla stringa appena letta).

*Esempio:* Il seguente frammento di codice legge il file miofile.txt riga per riga e ne stampa il contenuto su schermo.

**leggi-file-fgets.c**

```
FILE* file = fopen("/home/studente/Documents/miofile.
    txt", "r");

if (file == NULL){
    printf("Errore nell'apertura del file\n");
    exit(1);
}
char line[256];
while (fgets(line,sizeof(line),file) != NULL){
    printf("%s",line);
}
int close = fclose(file);
if (close == EOF){
    printf("Errore nella chiusura del file\n");
    exit(1);
}
```

Infine, è anche possible leggere un file carattere per carattere, usando la funzione:

```
int fgetc(FILE* file)
```

Questa funzione legge il carattere corrente del file a cui il parametro file fa riferimento, avanzando ad ogni invocazione, e restituendo il codice ASCII dell'ultimo carattere letto. Raggiunta la fine del file, fgetc restituisce la costante EOF.

#### 5.4.6.1. Schema di ciclo di lettura da file

Osserviamo che la lettura da file avviene secondo lo schema di ciclo seguente:

```
FILE* f = fopen(...);
//...
while (!<fine-file>) {
  // leggi prossimo blocco (carattere, parola o riga)
}
fclose(f);
//...
```

# 6. Organizzazione di un programma in file multipli

## 6.1. Organizzazione di un programma

Programmi complessi possono essere organizzati in più file. Questo consente una maggiore leggibilità e possibilità di riuso.

I file contenenti specifiche di programmi si distinguono in

- *file di intestazione* o *header* (estensione `.h`)
- *file di implementazione* (estensione `.c`).

Nei file header vengono inserite solamente le dichiarazioni delle funzioni, mentre nei file di implementazione anche il corpo che implementa la funzione. Solitamente la funzione `main` si trova in un file separato da quello in cui vengono definite altre funzioni.

Ogni file che usa funzioni definite altrove deve includere il relativo file header (non il file con l'implementazione `c`!)

*Esempio:*

File `f.h`

```
// dichiarazione delle funzioni radiceQuadrata e
    stampa
double radiceQuadrata(double x);
void stampa(double x);
```

File `f.c`

```
#include <stdio>
#include <math>
#include "f.h"

// implementazione della funzione radiceQuadrata
double radiceQuadrata(double x) {
    return sqrt(x);
}
// implementazione della funzione stampa
void stampa(double x) {
    printf("x = %f\n", f);
}
```

File `main.c`

```
#include "f.h"

// implementazione della funzione main
int main(int x)
{
    stampa(radiceQuadrata(25));
}
```

File multipli vengono compilati indicando tutti (e soli) i file `.c`. Esempio:

```
gcc -o test f.c main.c
```

### 6.1.1. Variabili globali definite `extern`

Quando si considerano file multipli, si può dichiarare `extern` una variabile globale. In questo caso per la variabile non viene allocata la memoria.

Quindi quando i programmi definiti su file diversi condividono la stessa variabile, questa viene dichiarata ed inizializzata in uno solo di essi, mentre negli altri viene dichiarata `extern`.

File a.c

```
float pi = 3.14f; // Viene allocata memoria per la
    variabile
```

File b.c

```
extern float pi; // La variabile pi e' definita in un
    altro file
```

L'uso di variabili `extern` è fortemente scoraggiato in quanto possibile causa di inconsistenze. Lo scambio di informazione tra funzioni deve essere basato quanto più possibile sull'uso dei parametri.

## 6.2. Il processo di compilazione

Il processo di generazione dei file eseguibili è costituito da tre fasi.

- preprocessore
- generazione codice oggetto
- collegamento (*linking*)

### 6.2.1. Il preprocessore

Trasforma un programma con direttive di preprocessamento denotate dal simbolo `#` in un programma C in cui le direttive sono state eliminate, dopo essere state eseguite.

- direttive di inclusione
- definizioni di macro
- compilazione condizionale
- altre direttive (si rimanda al manuale)

#### 6.2.1.1. Direttiva `include`

```
#include <stdio.h>
```

La linea viene sostituita dal testo del file corrispondente.

```
gcc -E filespanso.c nomefile.c
```

Produce un file C risultato del preprocessore.

#### 6.2.1.2. Direttiva `define`

Abbiamo già visto l'uso di `define` per definire valori costanti

```
#define pi 3.14f
```

A differenza della dichiarazione di una costante, l'effetto di questa definizione consiste nella sostituzione testuale (da parte del preprocessore) di tutte le occorrenze dell'identificatore `pi` con `3.14f`.

```
#define identificatore elenco-sostituzione
```

Si noti che tutti i simboli che si trovano a destra dell'identificatore (spazi esclusi) vengono presi come sostituzione.

Inoltre le definizioni restano valide fino alla fine del file.

Esempi

```
#define N = 100  /*** SBAGLIATO ***/

#define BOOL int
#define TRUE 1
#define FALSE 0

#define BEGIN {
#define END }
```

### 6.2.2. Macro parametriche

Tramite la direttiva define si possono definire anche macro con parametri.

**define con parametri**

Sintassi:

```
#define identificatore(x1, ...xn) sostituzione
```

(x1, ...xn) sono i parametri ed andranno sostituiti testualmente ai corrispondenti parametri formali delle espressioni: identificatore(y1, ...,yn)

Esempio

```
#define MAX(x,y) ((x)>(y)?(x):(y))
...
i = MAX(j+k, m-n);
```

diventa

```
i = ((j+k)>(m-n)?(j+k):(m-n));
```

Si noti che in questo caso abbiamo una definizione di macro con sostituzione di parametri per nome. Inoltre in questo caso l'intero corpo della macro viene sostituito testualmente in corrispondenza dell'uso. Si noti anche che non viene effettuato nessun controllo di tipo.

Le definizioni di macro possono utilizzare diversi altri accorgimenti sintattici (ad esempio l'uso del simbolo #) per i quali si rimanda al manuale del linguaggio.

Esistono inoltre diverse macro predefinite: _DATE_, _TIME_,...

### 6.2.3. Compilazione condizionale

Il compilatore C, più precisamente il preprocessore consente di utilizzare delle direttive che hanno una forma simile alle istruzioni condizionali, ma un effetto completamente diverso, in quanto consentono di selezionare quali parti di codice dovranno essere compilate. Sebbene tali direttive siano utili solo quando si devono gestire grandi quantità di codice, e quindi esulino dagli scopi di questa dispensa, si accenna brevemente al loro utilizzo.

**Compilazione condizionale**

*Sintassi:*

```
#if condizione1
  sequenza-istruzioni1
#elif condizione2
  sequenza-istruzioni2
...
#else
  sequenza-istruzioni-else
```

- `condizione` è un'espressione
- `sequenza-istruzioni` è una sequenza di istruzioni

*Semantica:*
Viene valutata prima la `condizione1`. Se la valutazione fornisce un valore diverso da 0, viene inclusa nel programma `sequenza-istruzioni1`. Lo stesso avviene per tutte le successive coppie condizione sequenza-istruzioni, corrispondenti alle sezioni #elif. Se nessuna delle condizioni fornisce un valore maggiore di 0, viene inclusa nel programma `sequenza-istruzioni-else`.

*Esempio:*

```
...
#define PALMARE 1
...
#if PALMARE
  short a,b;
#else
  int a,b;
```

L'esecuzione di questo frammento di codice produce un programma che dichiara le variabili a,b di tipo short, quando la variabile PALMARE è definita pari ad 1, altrimenti le dichiara di tipo int.

#### 6.2.3.1. Direttive #ifdef ed #ifndef

Queste varianti della direttiva #if consentono di verificare se un identificatore è stato precedentemente definito o meno.

*Esempio:*

```
#ifndef MYMATH_H
#define MYMATH_H
// calcola le equazioni di secondo grado
int secondoGrado(double a,double b,double c, double*
    x1r,
    double* x1i,double* x2r,double* x2i);

// calcola la radice quadrata con il metodo di Newton
float radiceNewton(float x);
#endif
```

In questo caso la compilazione condizionale consente di evitare di includere delle definizioni se queste sono già state precedentemente incluse, evitando in questo modo di introdurre ridondanze nel codice.

### 6.2.4. Struttura del compilatore

Il processo di compilazione è articolato in diverse fasi:

- Analisi lessicale (sequenza di token)
- Analisi sintattica (albero sintattico)
- Analisi semantica statica (albero sintattico annotato)
- Generazione codice intermedio
- Ottimizzazione
- Generazione codice macchina

A ciascuna fase corrisponde un formato di output prodotto, in alcuni casi la fase di ottimizzazione del codice dipende dalla particolare architettura di elaborazione considerata e pertanto le ultime due fasi sono mescolate. Esistono compilatori che effettuano tutte le fasi del processo di compilazione in una sola passata, altri che effettuano più passate, ciascuna delle quali produce un output completo per la fase di compilazione successiva (es. preprocessore C).

#### 6.2.4.1. Fasi della compilazione

La compilazione si divide anche in

- analisi/sintesi (prime 3 ed ultime 3 fasi)
- front-end/back-end (prime 4 ed ultime 2 fasi)

La prima suddivisione, che distingue l'analisi dalla sintesi, ha come risultato intermedio l'albero sintattico del programma in input corredato di informazioni relative alla semantica del programma.

La seconda suddivisione ha invece una carattere tipicamente ingegneristico, in quanto la prima parte del processo che termina con la generazione del codice intermedio è indipendente dalla particolare architettura di elaboratore per la quale viene generato il codice. Pertanto, la realizzazione del compilatore per un linguaggio di programmazione

per diverse architetture ha in comune il front-end, mentre il back-end deve essere differenziato in base alle caratteristiche dell'architettura sulla quale verrà eseguito il programma.

Per alcuni linguaggi di alto livello il C rappresenta il linguaggio intermedio. Cioè il compilatore per il linguaggio produce codice C che a sua volta viene compilato con il compilatore C per produrre il codice eseguibile

Si usa il termine *cross-compilatore* per indicare un compilatore per un'architettura diversa da quella sulla quale è implementato il compilatore stesso.

La realizzazione di compilatori sfrutta spesso una tecnica denominata *bootstrapping* che si basa sulla realizzazione del compilatore di un piccolo sottoinsieme del linguaggio, che poi viene utilizzato per scrivere il compilatore del linguaggio stesso. In questo modo, il compilatore del linguaggio viene scritto nel linguaggio stesso.

### 6.2.5. Compilazione incrementale

Quando il programma prevede molti componenti, per snellire il processo di generazione del codice eseguibile, i file possono essere compilati separatamente e in modo incrementale. Con riferimento all'esempio precedente, il comando:

```
gcc -c f.c
```

produce il file `f.o` che contiene il codice oggetto, cioè il codice tradotto in linguaggio macchina. Il codice oggetto, per poter essere eseguito, deve prima essere collegato agli altri moduli che ad esso fanno riferimento o a cui esso fa riferimento. Questa operazione si chiama *collegamento (linking)* ed usualmente viene fatta dal compilatore stesso. Il collegamento ha lo scopo di risolvere tutti i riferimenti ai simboli del programma. In C il collegamento viene effettuato con il comando `gcc`:

```
gcc -o test f.o main.c
```

consente di fare il collegamento del codice oggetto precedentemente compilato.

#### 6.2.5.1. Costruzione incrementale del codice eseguibile

La sequenza completa di operazioni relativa all'esempio precedente risulta essere:

```
gcc -c f.c
gcc -c main.c
gcc -o main f.o main.o
```

I primi due comandi generano il codice oggetto corrispondente ai file `f.o` e `main.o`, rispettivamente. Il terzo comando effettua il collegamento e genera il file eseguibile `main`.

#### 6.2.5.2. Codice rilocabile

Il codice generato dal compilatore si definisce codice *rilocabile*. Con questo termine si caratterizza il fatto che gli indirizzi di memoria sono indicati nel programma in modo relativo, facendo riferimento ad una collocazione del programma arbitraria.

Questo accorgimento consente di utilizzare in modo efficiente la memoria a disposizione del calcolatore, dato che la zona di memoria in cui il programma viene posizionato in fase di esecuzione, può essere cambiata, a seconda delle necessità. Tuttavia, ciò comporta che il programma eseguibile debba subire un'ulteriore trasformazione, che riguarda il calcolo degli indirizzi di memoria.

La *rilocazione* del codice consiste nella determinazione degli indirizzi di memoria in cui viene memorizzato il programma al momento dell'esecuzione.

## 6.3. Compilazione tramite `make`

Quando si costruiscono sistemi complessi, costituiti da molti moduli da compilare separatamente, diventa indispensabile utilizzare degli strumenti per gestire il processo di compilazione. Negli ambienti di programmazione come ad esempio *Geany*, il processo di costruzione del codice eseguibile viene controllato in modo semplificato con meccanismi specifici.

Lo strumento di base offerto da UNIX a questo scopo è il comando `make`.

La specifica dei passi di compilazione avviene tramite un file costituito da una sequenza di regole, chiamato `makefile`.

*Esempio:*

```
provamymath: provamymath.o mymath.o
  g++ -o provamymath provamymath.o mymath.o

provamymath.o: provamymath.c mymath.h
  g++ -c provamymath.c

mymath.o: mymath.c mymath.h
  g++ -c mymath.c
```

Ciascuna regola definisce un passo del processo di compilazione ed è costituita da un identificatore che corrisponde al nome del file prodotto dal passo di compilazione ad essa corrispondente, da un elenco di dipendenze, e, nella riga seguente, preceduto dal carattere "tab" il comando di compilazione.

L'esecuzione del comando `make` comporta la verifica di quali file sono stati modificati dopo l'ultima compilazione e l'esecuzione dei comandi influenzati (dipendenti) dai cambiamenti individuati, in modo da minimizzare il tempo di compilazione.

*Esempio:* Se è stato modificato il file `mymath.c` non occorre ricompilare il file `provamymath.o`.

Per una trattazione completa del comando `make` si rimanda ai manuali di UNIX. Vale la pena notare che questo strumento, che è indipendente dal linguaggio di programmazione utilizzato, offre numerose funzionalità, il cui uso richiede una conoscenza approfondita. Per questo motivo, sono stati sviluppati altri strumenti con l'obiettivo di semplificare l'utilizzo di `make`; tra i più diffusi `cmake`. Si noti, infine, che le funzionalità di gestione del processo di compilazione offerte dagli ambienti di programmazione, sono in genere implementate tramite il comando `make`.

# 7. Tipi di dato strutturati

## 7.1. Record

Supponiamo di dover realizzare un'applicazione per manipolare i dati anagrafici di persone: nome, cognome e data di nascita. Sarebbe molto comodo poter trattare ciascuna persona come un'unica variable contenente due campi stringa per memorizzare nome e cognome, e tre campi di tipo intero (short) per memorizzare giorno, mese e anno di nascita. Il tipo di questa variabile sarebbe concettualmente simile ad un array (di cinque elementi), ma con un'importante differenza: i tipi in essa contenuti non sono omogenei. Strutture dati di questo tipo vengono chiamate *record*.

### 7.1.1. Definire variabili di tipo record con struct

In C è possibile definire variabili di tipo record tramite il costrutto struct come segue:

**Dichiarazione di variabile di tipo record**

*Sintassi:*

```
struct {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
} lista-variabili;
```

dove:

- *dichiarazione-variabile-i* è la dichiarazione dell'*i*-esimo campo del record, fornita secondo la sintassi per la dichiarazione di variabili;
- *lista-variabili* è la lista delle variabili di tipo record che si vuole dichiarare.

*Semantica:*
Vengono create le variabili *menzionate* in *lista-variabili* e, per ciascuna di esse, allocata memoria per un un record avente come membri i campi dichiarati in *dichiarazione-variabile-i*, $i \in 1, \ldots, n$.

*Esempio:* Il seguente frammento di codice mostra la dichiarazione di due variabili che potrebbero essere usate per memorizzare le informazioni rilevanti di altrettante persone.

```
struct{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
} persona_1, persona_2;
```

Le variabili persona_1 e persona_2 possono essere usate a partire dal momento della loro dichiarazione.

Anche nel caso di variabili di tipo record, come per le variabili di tipo primitivo, la dichiarazione ha per effetto l'allocazione della memoria necessaria a contenere valori del tipo della variabile. Nel caso dei record, la memoria allocata è un'area contigua, la cui dimensione è sufficiente a contenere tutti i campi presenti nel tipo.

*Esempio:* La figura seguente mostra l'area di memoria allocata per la variabile persona_1.

persona_1

| nome | cognome | giorno | mese | anno |
|---|---|---|---|---|
| (4 byte) | (4 byte) | (2 byte) | (2 byte) | (2 byte) |

Come si può vedere, la memoria allocata ha una struttura simile a quella di un array, con la differenza che i campi possono essere di dimensione diversa.

### 7.1.2. Creare nuovi tipi di dato record con struct

Sebbene il C permetta di dichiarare variabili di tipo record specificandone contestualmente la struttura (come visto sopra), questa pratica è sconsigliata in quanto, nel caso di più dichiarazioni di variabile in diversi punti del programma, richiede che la stessa struttura sia ripetuta più volte, con conseguenze negative anche sulla manutenibilità del programma. Inoltre, e forse questo è l'aspetto più importante, variabili dichiarate in punti diversi sono considerate di tipi diversi, anche nel caso in cui i rispettivi record siano identici.

*Esempio:* Il seguente frammento di codice contiene dichiarazioni di variabile con record aventi strutture identiche, in punti diversi del programma.

```
int main (int argc, char** argv){
//...
    struct{
        char* nome;
        char* cognome;
        short int giorno;
        short int mese;
        short int anno;
    } persona_1, persona_2;
//...
    struct{
        char* nome;
        char* cognome;
        short int giorno;
        short int mese;
        short int anno;
    } persona_3;
}
```

Se dopo aver realizzato il programma si dovessero apportare modifiche alla struttura del record, ad esempio aggiungere un campo, queste andrebbero effettuate in entrambe le dichiarazioni. Si noti inoltre che le variabili persona_1 e persona_2 sono dello stesso tipo, ma la variabile persona_3 è considerata di un altro tipo, nonostante l'uguaglianza nella struttura dei record.

### 7.1.3. Tag di struttura

Un primo modo per identificare un record è tramite i cosiddetti *tag di struttura*, ovvero etichette che identificano la struttura di un record.

**Dichiarazione di un tag di struttura**

*Sintassi:*

```
struct nome-tag {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
};
```

dove:

- *nome-tag* è l'etichetta che si vuole usare per identificare la struttura;

*Semantica:* Viene definito il tag *nome-tag*.

Una volta definiti, i tag di struttura possono essere usati per definire variabili.

*Esempio:* Nel seguente frammento di codice, viene definito il tag di struttura persona, successivamente utilizzato nelle dichiarazioni delle variabili persona_1, persona_2 e persona_3.

**persona-tag.h**

```
// File persona-tag.h
#ifndef PERSONATAG_H
#define PERSONATAG_H

// Definizione tag struttura
struct persona{
    char *nome, *cognome;
    short int giorno, mese, anno;
};

#endif
```

**persona-tag.c**

```c
// File persona-tag.c

#include "persona-tag.h"

int main(int argc, char** argv){
    //...
    struct persona persona_1, persona_2;
    //...
    struct persona persona_3;
}
```

Per definire variabili di tipo record mediante tag di struttura, è necessario l'uso della parola chiave struct prima del tag di struttura, come mostrato nell'esempio. Variabili definite in questo modo hanno tipi compatibili. Si noti la strutturazione del codice in più file: nel file header (persona-tag.h) viene definito il tag di struttura, successivamente usato nel file persona-tag.c.

### 7.1.4. Definire tipi di dato record con typedef

Il secondo modo messo a disposizione dal C per identificare una struttura con un nome simbolico consiste nel definire un tipo record.

**Definizione di un nuovo tipo di dato record**

*Sintassi:*

```
typedef struct {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
} nome;
```

dove:

- *nome* è il nome del nuovo tipo che si vuole definire;
- *dichiarazione-variabile-i* è la dichiarazione dell'*i*-esimo campo del nuovo tipo;

*Semantica:*

Viene definito un nuovo tipo di dato record con nome *nome*, avente come membri i campi in esso dichiarati, ciascuno del rispettivo tipo.

Si noti la presenza del carattere ;, sempre obbligatoria, al termine della definizione. Come già discusso, i nuovi tipi di dato vengono di norma definiti nei file header, nella sezione relativa alle dichiarazioni di tipo, dopo le direttive di inclusione e prima di dichiarare eventuali funzioni.

*Esempio:* Il seguente frammento di codice mostra una possibile definizione del record Persona.

**persona.h**

```
// File persona.h
#ifndef PERSONA_H
#define PERSONA_H

typedef struct{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
} Persona;

#endif
```

Questa dichiarazione sta di fatto introducendo un nuovo tipo di dato, Persona, che sarà utilizzabile come ogni altro tipo.

Si può anche definire un nuovo tipo di dato record combinando typedef e tag di struttura.

*Esempio:* Nel seguente file, il tag di struttura persona viene usato per definire il tipo record Persona.

```
//Definizione tag:
struct persona{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
};

// Definizione tipo
typedef struct persona Persona;
```

Questa dichiarazione di tipo è equivalente alla precedente.

La definizione di un nuovo tipo per identificare un tipo di dato record è la soluzione da preferire. Salvo casi particolari, questo è il modo in cui saranno definiti i tipi di dato record nel seguito.

In C++ è possibile definire un nuovo tipo record direttamente con la definizione struct.

Nell'esempio precedente, in C++, persona è già un tipo record e può essere usato senza dover definire un nuovo tipo Persona tramite typedef.

### 7.1.5. Variabili di tipo record

Una volta definito un tipo di dato record, esso, come ogni tipo definito dall'utente, diventa disponibile per l'uso.

*Esempio:* Nel seguente programma viene dichiarata la variabile persona_1 di tipo Persona

```
#include "persona.h"
//...
int main(int argc, char** argv){
  Persona persona_1;
}
```

Si osservi che la variabile persona_1 è dichiarata come una comune variabile, trattando Persona come un comune tipo di dato. In particolare, non è richiesto l'uso della parola chiave struct nella dichiarazione di variabile. Chiaramente, variabili distinte di tipo Persona sono dello stesso tipo.

### 7.1.6. Accesso ai campi di un record

Per accedere al campo di un record si utilizza l'operatore . (punto). I campi di un record possono essere trattati, in maniera simile a quanto avviene per gli array, come variabili.

*Esempio:* Il seguente programma dichiara la variabile p1 di tipo Persona, ne inizializza i campi e successivamente li stampa.

**persona.c**

```
#include <stdio.h>
#include "persona.h"

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

    Persona p1;
    p1.nome = "Mario";
    p1.cognome = "Rossi";
    p1.giorno = 7;
    p1.mese = 4;
    p1.anno = 1964;

    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

### 7.1.7. Inizializzazione di variabili di tipo record

Un'alternativa all'inizializzazione dei campi di un record uno ad uno è l'utilizzo delle parentesi, in maniera simile a quanto visto per gli array. Questa può avvenire su base posizionale o tramite *designatori*, overo espressioni del tipo *.campo-i*.

*Esempio:* La seguente linea di codice dichiara ed inizializza la variabile p1 di tipo Persona, associando l'*i*-esimo valore all'*i*-esimo campo, secondo l'ordine di comparizione nella definizione del tipo:

```
Persona p1 = {"Mario","Rossi",7,4,1964};
```

La seguente linea di codice dichiara ed inizializza la variabile p1 di tipo Persona assegnando esplicitamente a ciascun campo un valore:

```
Persona p1 = {
    .cognome="Rossi",
    .giorno = 7,
    .mese = 4,
    .nome="Mario",
    .anno = 1964
};
```

In questo caso, l'ordine con cui i campi vengono assegnati non conta, in quanto il campo a cui assegnare il valore è identificato dal designatore.

Entrambe le inizializzazioni degli esempi appena visti hanno esattamente lo stesso effetto della dichiarazione e delle assegnazioni campo a campo viste in precedenza.

L'inizializzazione di record può aver luogo solo contestualmente alla sua dichiarazione.

### 7.1.8. Assegnazione e uguaglianza

#### 7.1.8.1. Assegnazione di record

Per quanto riguarda l'assegnazione, le variabili di tipo record vengono trattate allo stesso modo delle variabili di tipi primitivi. In particolare, è possibile assegnare ad una variabile di tipo record un valore di tipo record di tipo compatibile con quello della variabile.

*Esempio:*

```
Persona p1 = ... ;
Persona p2;
p2 = p1;
```

L'effetto dell'assegnazione è la copia campo a campo del record restituito dall'espressione p1, cioè il contenuto della variabile, nel record contenuto nella variabile p2. Si osservi la differenza rispetto agli array, per i quali l'assegnazione è disponibile solo considerando gli array come puntatori e avviene quindi con condivisione di memoria.

Nella copia campo a campo, anche eventuali campi di tipo array *allocati staticamente* (ad esempio char[256]) all'interno del record vengono copiati. Gli array allocati dinamicamente (ad esempio char *) invece vengono copiati come puntatori e quindi comportano una condivisione di memoria.

Se si considera che l'assegnazione non è un'operazione disponibile su variabili di tipo array, questo comportamento potrebbe apparire sorprendente. Tuttavia, si osservi che il nome di una variabile di tipo struttura identifica una variabile. Quindi, poiché l'assegnazione tra variabili, ad esempio p2 = p1, prevede la copia del contenuto della memoria di una variabile nella memoria dell'altra, se la prima variabile contiene un array allocato staticamente come membro, esso viene automaticamente copiato per effetto dell'assegnazione tra le variabili di tipo record. Una variabile di un array invece identifica l'indirizzo di memoria dell'array e pertanto l'assegnazione opera su puntatori. In un certo senso, l'uso di una struttura rende disponibile, indirettamente, l'assegnazione tra array allocati staticamente.

*Esempio:* Si consideri il seguente programma. Il record PuntoColorato, definito nel file punto-colorato.h rappresenta un punto nello spazio cartesiano, a cui è associato un colore. Nello stesso file è inoltre dichiarata la funzione stampaPuntoColorato che stampa su schermo il contenuto del parametro p di tipo PuntoColorato. La definizione di tale funzione è fornita nel file punto-colorato.c. Il file punto-colorato-main.c mostra invece un esempio d'uso del record e della funzione dichiarati nel file header.

**punto-colorato.h**

```
// File punto-colorato.h
#ifndef PUNTOCOLORATO_H
#define PUNTOCOLORATO_H

typedef struct{
    float coord[3]; // coordinate nello spazio
    char colore[256]; // colore del punto
} PuntoColorato;

void stampaPuntoColorato(PuntoColorato p);
#endif
```

**punto-colorato.c**

```
// File punto-colorato.c

#include <stdio.h>
#include "punto-colorato.h"

void stampaPuntoColorato(PuntoColorato p){
```

```
    printf("((%f,%f,%f),%s)\n",
           p.coord[0], p.coord[1], p.coord[2],
           p.colore);
}
```

**punto-colorato-main.c**

```
// File punto-colorato-main.c
#include <stdio.h>
#include "punto-colorato.h"

int main(int argc, char** argv){
    PuntoColorato pc1 = {{0,0,0},"bianco"};
    PuntoColorato pc2 = pc1;

    stampaPuntoColorato(pc1);
    stampaPuntoColorato(pc2);

    pc1.coord[0] = 3; // coord di pc1 cambia

    stampaPuntoColorato(pc2); // pc2 non cambia
}
```

Come detto, l'effetto dell'assegnazione è quello di copiare campo a campo il record p1 nel record p2. La seguente figura mostra lo stato della memoria immediatamente dopo l'esecuzione della linea PuntoColorato pc2 = pc1;

pc1

| coord | | | colore |
|---|---|---|---|
| 0 | 0 | 0 | "bianco" |

pc2

| coord | | | colore |
|---|---|---|---|
| 0 | 0 | 0 | "bianco" |

Si osservi che ciascun record contiene un proprio array distinto dall'altro. Pertanto, dopo l'esecuzione, nel programma principale, della linea pc1.coord[0] = 3; , l'array contenuto in pc1 sarà cambiato, mentre quello in pc2 resterà invariato:

pc1

| coord | | | colore |
|---|---|---|---|
| 3.0 | 0 | 0 | "bianco" |

pc2

| coord | | | colore |
|---|---|---|---|
| 0 | 0 | 0 | "bianco" |

Si noti che in questo esempio il campo colore del record PuntoColorato è dichiarato in maniera statica char[256] e viene quindi copiato

dall'istruzione di assegnazione. Si ricorda anche che un array allocato staticamente non può essere assegnato con l'istruzione di assegnazione in seguito alla sua dichiarazione, quindi nel codice mostrato in precedenza l'istruzione pc1.colore = "bianco" provocherebbe un errore a tempo di compilazione, in quanto la stringa "bianco" è di tipo const char [7] e l'istruzione sarebbe intepretata come assegnazione tra array statici non consentita in C. Per accedere in scrittura all'area di memoria del campo colore si può usare ad esempio la funzione strcpy.

#### 7.1.8.2. Uguaglianza tra record

Sebbene il C metta a disposizione l'operatore di assegnazione, esso non permette il test uguaglianza tra record. In altre parole, gli operatori == e != non sono applicabili a record.

### 7.1.9. Puntatori a record

È anche possibile dichiarare puntatori a variabili di tipo record.

*Esempio:* Il seguente frammento di codice dichiara una variabile p di tipo puntatore a Persona

```
Persona* p;
```

Per accedere ai campi di un record tramite il puntatore, una prima alternativa consiste nell'accedere prima alla variabile puntata dal puntatore, mediante l'operatore *, e successivamente accedere al campo d'interesse.

*Esempio:* Nel seguente frammento di codice, l'accesso al campo nome della variabile p di tipo Persona viene effettuato tramite il puntatore punt_p a Persona.

```
Persona p = {"Mario","Rossi",7,4,1964};
Persona *punt_p = &p;
printf("Nome: %s\n",(*punt_p).nome);
```

Si osservi l'uso delle parentesi nell'espressione (*punt_p).nome, necessarie in quanto l'operatore . ha precedenza più alta rispetto all'operatore *.

La seconda alternativa, più comoda e comunemente usata, consiste nell'uso dell'operatore -> (freccia a destra).

*Esempio:* Si consideri il seguente frammento di codice

```
Persona p = {"Mario","Rossi",7,4,1964};
Persona *punt_p = &p;
```

Le seguenti espressioni sono equivalenti:

```
(*punt_p).nome
punt_p->nome
```

Pertanto, entrambe le seguenti istruzioni stampano la stringa Mario

```
printf("%s\n", (*punt_p).nome);
printf("%s\n", punt_p->nome);
```

Passaggio di parametri e restituzione di un risultato di tipo record da parte di funzioni avvengono secondo gli stessi meccanismi visti per le variabili di tipi primitivi.

*Esempio:* La seguente funzione prende in input un parametro di tipo Persona ed un intero che rappresenta un anno e restituisce l'età della persona, calcolata come differenza tra l'anno corrente e l'anno di nascita della persona.

```
#include <stdio.h>
#include "persona.h"

int calcolaEta(Persona p, int anno_corrente){
    return anno_corrente - p.anno;
}

// Esempio d'uso:

int main(){
    Persona p = {"Mario","Rossi",7,4,1964};
    printf("eta': %d\n",calcolaEta(p,2018));
}
```

Diversamente da quanto accade per gli array, il passaggio di parametri di tipo record avviene *per valore*: al momento dell'invocazione della funzione, il record viene copiato nel RDA della funzione.

*Esempio:* La seguente funzione prende in input un parametro di tipo Persona e vi apporta delle modifiche. Tuttavia, grazie al meccanismo di passaggio per valore, tali modifiche non sono visibili al modulo chiamante.

```c
#include <stdio.h>
#include "persona.h"

void cambiaEta(Persona p, int nuovo_anno){
    p.anno = nuovo_anno;
}

// Esempio d'uso:

int main(){
    Persona p = {"Mario","Rossi",7,4,1964};
    cambiaEta(p,2000);
    printf("anno: %d\n",p.anno); // stampa 1964
}
```

Anche il caso in cui una funzione prende in input un puntatore a record è sostanzialmente analogo al caso di variabili di tipi primitivi. In particolare, tramite puntatori è possibile effettuare side-effect sul record puntato dal parametro, in maniera tale da rendere visibili al modulo chiamante le modifiche apportate.

*Esempio:* La seguente funzione prende in input un puntatore a record di tipo Persona e ne assegna i campi a valori di default. Poiché effettuate tramite puntatore, le modiche effettuate dalla funzione sono visibili nel programma principale, dopo la sua invocazione.

```c
#include <stdio.h>
#include "persona.h"

void inizializzaPersona(Persona* p){
    p -> nome = "";
    p -> cognome = "";
    p -> giorno = 0;
    p -> mese = 0;
    p -> anno = 0;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona p1;
    inizializzaPersona(&p1);
    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

Al pari dei valori di tipi primitivi, una funzione può restituire un record.

*Esempio:* La seguente funzione assegna alla variabile locale p di tipo Persona un record i cui campi sono popolati con il valore dei parametri. Il contenuto della variabile viene quindi restituito.

```c
#include <stdio.h>
#include "persona.h"

Persona nuovaPersona(char* nome, char* cognome, int g,
     int m, int a){
    Persona p;
    p.nome = nome;
    p.cognome = cognome;
    p.giorno = g;
    p.mese = m;
    p.anno = a;
  return p;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona p1 = nuovaPersona("Mario", "Rossi",
    10,2,1990);
    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

Si osservi che la funzione restituisce una copia del valore memorizzato nella variabile p che viene assegnato, nella funzione main, alla variabile p1. Essendo il valore un record, l'assegnazione avviene, come visto, campo a campo.

### 7.1.10. Allocazione dinamica e deallocazione di record

La funzione malloc può essere usata per allocare dinamicamente variabili di tipo record.

```
Persona *p1 = (Persona*) malloc(sizeof(Persona));
```

Per i campi di tipo puntatore (ad esempio, array dichiarati dinamicamente, come char *) non viene allocata memoria. Pertanto è necessario allocare (e in seguito deallocare) esplicitamente la memoria per tutti i campi di tipo puntatore.

Per il resto, non vi sono sostanziali differenze con il caso di variabili di tipi primitivi, eccetto che, una volta allocata una variabile di tipo record, per accedere ai suoi membri occorre utilizzare l'operatore -> (o gli operatori * e . opportunamente combinati).

Anche la deallocazione viene eseguita in maniera analoga al caso di variabili di tipi primitivi, ovvero tramite la funzione free:

```
free(p1);
```

Nel caso in cui il record contiene campi di tipo puntatore allocati dinamicamente, è necessario rilasciare la memoria associata a tali campi, prima di rilasciare la memoria associata al record.

*Esempio:* Il seguente programma mostra la dichiarazione dinamica di una variabile di tipo Persona e l'accesso ai suoi campi.

```
#include <stdio.h>
#include <stdlib.h>
#include "persona.h"

int main(int argc, char** argv){
    Persona* p1 = (Persona*) malloc(sizeof(Persona));
    p1 -> nome = "Mario";
    p1 -> cognome = "Rossi";
    p1 -> giorno = 7;
    p1 -> mese = 4;
    p1 -> anno = 1964;

    printf("%s %s, %d/%d/%d\n", p1->nome, p1->cognome,
           p1->giorno, p1->mese, p1->anno);

    free(p1);
}
```

*Esempio:* La seguente funzione alloca dinamicamente un record di tipo Persona e ne restituisce il puntatore. Nella funzione main, il record viene utilizzato e, quando non piú necessario, deallocato.

```
#include <stdio.h>
#include <stdlib.h>
#include "persona.h"

Persona* puntNuovaPersona(char* nome, char* cognome,
    int g, int m, int a){
  Persona* p = (Persona*) malloc(sizeof(Persona));
    p -> nome = nome;
    p -> cognome = cognome;
    p -> giorno = g;
    p -> mese = m;
    p -> anno = a;
  return p;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona* p1 = puntNuovaPersona("Mario", "Rossi",
    10,2,1990);
    printf("%s %s, %d/%d/%d\n", p1 -> nome, p1 ->
    cognome, p1 -> giorno, p1 -> mese, p1 -> anno);
  // ...
  free(p1);
}
```

### 7.1.11. Record per la rappresentazione di matrici

I record sono particolarmente utili per definire un nuovo tipo di dato per memorizzare matrici allocate dinamicamente.

*Esempio:* dichiarazione di un tipo Mat per rappresentare matrici di float.

```
typedef struct {
  int rows;
  int cols;
  float **row_ptrs;
} Mat;
```

Nella definizione precedente, Mat è un nuovo tipo che contiene tre campi: rows: numero di righe della matrice, cols: numero di colonne della matrice, row_ptrs: puntatore all'array di array di float che contiene i valori della matrice.

L'accesso agli elementi della matrice avviene tramite i campi del record.

```
Mat *m = ...;  // allocazione dinamica della matrice

for (int i=0; i<m->rows; i++) {
  for (int j=0; j<m->cols; j++) {
     ... m->row_ptrs[i][j]  // operazione su elemento
    i,j
  }
}
```

### 7.1.12. Record annidati

Come detto, una volta definito un tipo record, esso può essere usato come un qualsiasi altro tipo, ad esempio come campo di un record.

*Esempio:* Nel seguente file il tipo Punto viene sfruttato per fornire una definizione alternativa del tipo PuntoColorato.

**punto-colorato2.h**

```
#ifndef PUNTOCOLORATO2_H
#define PUNTOCOLORATO2_H

typedef struct{
    float X;
```

```
    float Y;
    float Z;
} Punto;

typedef struct{
    Punto punto;
    char colore[256];
} PuntoColorato2;

#endif
```

Un esempio d'uso del tipo PuntoColorato2 è il seguente:

**punto-colorato2-main.c**

```
#include <stdio.h>
#include "punto-colorato2.h"

int main (int argc, char** argv){
    Punto p = {2.0f,2.2f,3.5f};
    PuntoColorato2 pc = {p,"verde"};

    printf("((%f,%f,%f),%s)\n",
        pc.punto.X,pc.punto.Y,pc.punto.Z,pc.colore);
}
```

Si osservi l'uso dell'operatore . per accedere prima al campo punto di pc (ottenuto con l'espressione pc.punto) e successivamente per accedere ai campi di punto, ad esempio pc.punto.X per ottenerne la coordinata X.

L'unica limitazione alla struttura di un record consiste nel fatto che quando si definisce un tipo record T, la sua struttura non può menzionare il tipo T al suo interno. Questa restrizione può tuttavia essere superata tramite l'uso di tag (v. seguito).

### 7.1.13. Record autoreferenziali

Un caso di record di particolare interesse è quello in cui si vuole che un record di tipo T faccia riferimento, tramite un suo campo, ad un altro record dello stesso tipo. Sebbene questo non introduca nessuna novità da un punto di vista concettuale, la limitazione sopra discussa impone l'uso di tag.

*Esempio:* Nel seguente frammento di codice viene definito un record con tag di struttura parente, utilizzato all'interno della struttura stessa. Il tag viene successivamente usato per definire il tipo Parente.

**parente.h**

```
#ifndef PARENTE_H
#define PARENTE_H

// Tag di struttura:
struct parente{
    char nome[256];
    char cognome[256];
    struct parente* padre;
    struct parente* madre;
};

// Definizione di tipo:
typedef struct parente Parente;
#endif
```

Si osservi che all'interno della definizione del record non si possono dichiarare campi di tipo struct parente, ad esempio padre, ma solo puntatori ad essi. Questo è dovuto al fatto che, nel momento in cui viene dichiarato il campo, il record parente è ancora in corso di definizione, ovvero è *incompleto*. Mentre il C vieta la dichiarazione di variabili di tipo incompleto, permette la dichiarazione di puntatori a tali variabili.

Faremo ampiamente uso di record di questo tipo quando tratteremo le *strutture collegate*.

### 7.1.14. Array di record

Infine, notiamo che è anche possibile dichiarare array di record.

*Esempio:* Il seguente programma dichiara un array famiglia di record di tipo Persona e ne inizializza la prima componente.

```
#include <stdio.h>
#include "persona.h"

int main (int argc, char** argv){
    Persona famiglia[3];

    famiglia[0].nome = "Mario";
    famiglia[0].cognome = "Rossi";
    famiglia[0].giorno = 7;
    famiglia[0].mese = 4;
    famiglia[0].anno = 1964;

    //...
}
```

Si noti che l'accesso alle componenti di ciascun record si effettua accedendo dapprima al record tramite l'operatore [] e successivamente usando il punto. Ad esempio l'espressione famiglia[0].nome denota il campo nome del record memorizzato come prima componente dell'array famiglia, ovvero famiglia[0].

## 7.2. Unioni

Un ulteriore tipo di dato strutturato messo a disposizione del C è union, o unione. union è un tipo che si comporta in maniera simile ad un record ma, a differenza di questo, *memorizza solo un campo alla volta*.

Variabili di tipo union e tipi union vengono rispettivamente dichiarate e definiti allo stesso modo dei record, sostituendo la parola chiave struct con union.

*Esempio:* Il seguente frammento di codice definisce il tipo di dato chardouble come una unione contenente i campi c di tipo char e d di tipo double.

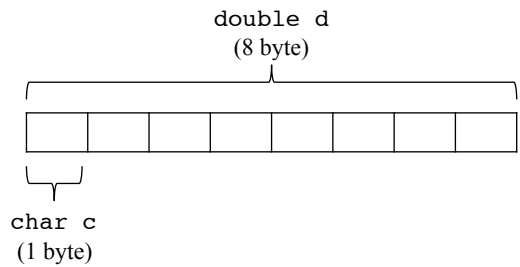
```
#ifndef CHARDOUBLE_H
#define CHARDOUBLE_H

typedef union{
    char c;
    double d;
} chardouble;

#endif
```

Una volta definito, il tipo di dato può essere usato per dichiarare variabili.

*Esempio:*

```
chardouble cd;
```

La dichiarazione di variabile ha per effetto l'allocazione di uno spazio di memoria di dimensione pari alla *dimensione massima tra quelle dei tipi dei campi della union*. Ad esempio, la dimensione della variabile cd appena definita è pari a sizeof(double) byte (tipicamente 8). La figura seguente mostra l'organizzazione della memoria associata alla variabile cd:




La caratteristica distintiva dei tipi union consiste nel fatto che *solo l'ultimo campo a cui è stato assegnato un valore è significativo*, ovvero contiene effettivamente il valore ad esso assegnato, mentre il valore contenuto negli altri campi è soggetto a modifiche arbitrarie. Questo deriva dal fatto che diversi campi condividono una stessa regione di memoria (ad es. il primo byte nella figura precedente) e quindi le modifiche apportate ad ognuno di essi hanno un impatto anche sugli altri. Per capire come ciò avvenga, si consideri nuovamente l'organizzazione della memoria della variabile cd. Come si può vedere i due campi c e d della variabile condividono il primo byte, pertanto ogni assegnazione al campo c avrà un impatto sul valore corrente del campo d e viceversa.

L'accesso ai campi di una union si effettua con le stesse modalità del caso dei record. Ad esempio, l'espressione cd.c denota il campo c della variabile c.

*Esempio:* Il seguente frammento di codice mostra l'uso di una variabile di tipo chardouble.

```
chardouble cd;
cd.c = 'a';
/* Solo il campo c e' significativo
(fino alla prossima assegnazione)*/
// ...
cd.d = 30.0;
/*Solo il campo d e' significativo
(fino alla prossima assegnazione)*/
//...
```

Il tipo di dato union è tipicamente usato per dichiarare variabili che possono assumere valori di diversi tipi. Ad esempio, la variabile cd può assumere, attraverso l'assegnazione ai suoi campi, valori di tipo char o double. Notiamo che un comportamento simile potrebbe essere ottenuto definendo un record, ad esempio:

```
typedef struct {
  char c;
  double d;
} doublechar;
```

quindi dichiarando una variabile doublechar dc, ed accedendo di volta in volta solo all'ultimo campo a cui è stato assegnato un valore. Si osservi tuttavia che questo approccio richiede una maggiore quantità di memoria.

Ad eccezione di questa importante differenza, le variabili di tipo union vengono trattate in C allo stesso modo dei record, in particolare per quanto riguarda:

- definizione di tipi e dichiarazione di variabili;
- passaggio di parametri (che avviene per valore);
- restituzione di valori di tipo union;
- accesso tramite puntatori (operatore ->).

## 7.3. Tipi di dato enumerati

In C è possibile definire tipi di dato che consistono in un insieme di valori enumerati esplicitamente.

**Definizione di un nuovo tipo di dato enumerato**

*Sintassi:*

```
typedef enum {
    valore-1,
    ...
    valore-n
} nome;
```

dove:

- *nome* è il nome del nuovo tipo che si vuole definire;
- *valore-i* è l'$i$-esimo elemento del tipo;

*Semantica:* Viene creato un nuovo tipo di dato enumerato i cui valori sono tutti e solo quelli elencati nella dichiarazione.

Dopo essere stato dichiarato, un tipo enumerato può essere usato come ogni altro tipo.

*Esempio:* Il seguente file header dichiara un tipo enumerato usato per memorizzare una direzione (alto, basso, destra, sinistra).

**direzione.h**

```
// File direzione.h

#ifndef DIREZIONE_H
#define DIREZIONE_H

typedef enum{
    ALTO,
    BASSO,
    DESTRA,
    SINISTRA
} Direzione;

#endif
```

Il tipo può essere usato per definire una variabile di tipo Direzione:

```
Direzione d;
```

La variabile può essere usata come una qualunque altra variabile:

```
d = BASSO;
//...
if (d == ALTO) {...}
if (d == BASSO) {...}
//...
```

# 8. Ricorsione

## 8.1. Funzioni ricorsive

Una funzione si dice *ricorsiva* se al suo interno contiene un'attivazione di sè stessa (eventualmente indirettamente, attraverso l'attivazione di altre funzioni). Vediamo alcuni esempi di funzioni matematiche sui naturali definite in modo ricorsivo.

*Esempio:* fattoriale

$$fatt(n) = \begin{cases} 1, & se\ n = 0 \quad (\texttt{caso base}) \\ n \cdot fatt(n-1), & se\ n > 0 \quad (\texttt{caso ricorsivo}) \end{cases}$$

La definizione ricorsiva di una funzione ha le seguenti caratteristiche:

- uno (o più) *casi base*, per i quali il risultato può essere determinato direttamente;
- uno (o più) *casi ricorsivi*, per i quali si riconduce il calcolo del risultato al calcolo della stessa funzione su un valore più piccolo o più semplice.

La definizione ricorsiva del fattoriale può essere implementata direttamente attraverso una funzione ricorsiva.

```
int fattoriale(int n) {
  if (n == 0)
    return 1;
  else
    return n * fattoriale(n - 1);
}
```

### 8.1.1. Esempio: implementazione ricorsiva della somma di due interi

Sfruttiamo la seguente definizione ricorsiva di somma tra due interi non negativi:

$$somma(x,y) = \begin{cases} x, & se\ y = 0 \\ 1 + somma(x, y-1), & se\ y > 0 \end{cases}$$

```
int somma(int x, int y) {
  if (y == 0)
    return x;
  else
    return 1 + somma(x, y-1);
}
```

### 8.1.2. Esempio: implementazione ricorsiva del prodotto tra due interi

Sfruttiamo la seguente definizione ricorsiva del prodotto tra due interi non negativi:

$$prodotto(x,y) = \begin{cases} 0, & se\ y = 0 \\ somma(x, prodotto(x, y-1)), & se\ y > 0 \end{cases}$$

Implementazione:

```
int prodotto(int x, int y) {
  if (y == 0)
    return 0;
  else
    return somma(x, prodotto(x, y-1));
}
```

### 8.1.3. Esempio: implementazione ricorsiva dell'elevamento a potenza

Sfruttiamo la seguente definizione ricorsiva dell'elevamento a potenza tra due interi non negativi:

$$potenza(b,e) = \begin{cases} 1, & se\ e = 0 \\ prodotto(b, potenza(b, e-1)), & se\ e > 0 \end{cases}$$

Implementazione:

```
int potenza(int b, int e) {
  if (e == 0)
    return 1;
  else
    return (prodotto(b, potenza(b, e-1)));
}
```

## 8.2. Confronto tra ricorsione e iterazione

Le funzioni implementate in modo ricorsivo ammettono anche un'implementazione iterativa, come già visto per somma, prodotto e potenza.

*Esempio:* implementazione iterativa del fattoriale, sfruttando la seguente definizione iterativa:

$$fatt(n) = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$$

```
int fattorialeIterativa(int n) {
  int ris = 1;
  while (n > 0) {
    ris = ris * n;
    n--;
  }
  return ris;
}
```

Caratteristiche dell'implementazione iterativa:

- inizializzazione:
  Es. ris = 1;
- operazione del ciclo, eseguita un numero di volte pari al numero di ripetizioni del ciclo:
  Es. ris = ris * n;

- terminazione:
  Es. n- -; consente di rendere la condizione (n > 0) del ciclo while falsa

Implementazione ricorsiva, sfruttando la definizione ricorsiva data precedentemente:

```
int fattoriale(int n) {
  if (n == 0)
    return 1;
  else
    return n * fattoriale(n - 1);
}
```

Caratteristiche dell'implementazione ricorsiva:

- passo base:
  Es. return 1;
- passo ricorsivo:
  Es. return n * fattoriale(n - 1) ;
- terminazione è garantita dal fatto che la chiamata ricorsiva fattoriale(n-1) diminuisce di uno il valore passato come parametro, per cui, se inizialmente n>0, prima o poi si arriverà ad un'attivazione in cui la condizione n==0 sarà vera e quindi viene eseguito solo il passo base.

### 8.2.1. Confronto tra ciclo di lettura e lettura ricorsiva

Struttura del ciclo di lettura per i file:

```
leggi primo elemento
while (elemento valido) {
  elabora elemento letto;
  leggi elemento successivo;
}
```

Struttura della lettura ricorsiva per i file:

```
leggi un elemento
if (elemento valido) {
  elabora elemento letto;
```

```
  chiama ricorsivamente lettura;
}
```

Struttura della lettura ricorsiva per tipi indicizzati generici:

```
if (dato vuoto) {
  passo base;
}
else {
  leggi un elemento
  elabora elemento letto;
  chiama ricorsivamente lettura sui dati rimanenti;
}
```

*Esempio:* copia di un file di input in un file di output.
Implementazione iterativa:

```
void copiaIterativa(FILE *i, FILE *o) {
    char x;
    x = fgetc(i);
    while (x != EOF) {
        fputc(x,o);
        x = fgetc(i);
    }
}
```

Implementazione ricorsiva:

```
void copiaRicorsiva(FILE * i, FILE * o) {
    char x;
    x = fgetc(i);
    if (x != EOF) {
        fputc(x,o);
        copiaRicorsiva(i,o);
    }
}
```

### 8.2.2. Esempio: gli ultimi saranno i primi

Vogliamo leggere i caratteri di un file di input e copiarli su un file di output, invertendone l'ordine. Facendo uso della ricorsione, questa operazione risulta particolarmente semplice.

```
void copiaInversa(FILE *i, FILE *o) {
    char x;
    x = fgetc(i);
    if (x != EOF) {
        copiaInversa(i,o);
        fputc(x,o);
    }
}
```

La funzione copiaInversa non si può facilmente formulare in modo iterativo, in quanto la scrittura su file in ordine inverso richiederebbe la memorizzazione dei caratteri letti in una struttura dati apposita. Analizzeremo questo esempio più avanti, mostrando in che modo le zone di memoria associate alle variabili locali delle attivazioni ricorsive fungono da memoria temporanea per i caratteri letti in input.

Facciamo notare la differenza tra questo tipo di ricorsione ed i casi più semplici visti in precedenza, in cui passare ad un'implementazione iterativa risulta immediato, come nel caso della funzione copiaRicorsiva. Questi casi sono quelli in cui l'ultima istruzione effettuata prima che la funzione termini è la chiamata ricorsiva (*tail recursion*). Alcuni compilatori sono in grado di riconoscere i casi di tail recursion, e di effetuare delle ottimizzazioni per generare un codice macchina più efficiente.

Facciamo invece notare che, in generale, un'implementazione ricorsiva potrebbe risultare più inefficiente di una corrispondente implementazione iterativa, a causa della necessità di gestire le chiamate ricorsive, come mostrato più avanti.

## 8.3. Schemi di ricorsione

Esistono diversi schemi ricorsivi degli algoritmi di base. Alcuni di essi sono illustrati di seguito.

### 8.3.1. Conteggio di elementi usando la ricorsione

Struttura della funzione ricorsiva per i file:

```
leggi un elemento;
if (!elemento valido)
   return 0;
else
```

```
   return 1 + risultato ricorsione;
```

Struttura della funzione ricorsiva per tipi indicizzati generici:

```
if (dato vuoto)
   return 0;
else
   return 1 + risultato ricorsione sui dati rimanenti;
```

#### 8.3.1.1. Esempio: lunghezza di una sequenza di caratteri

Caratterizzazione ricorsiva dell'operazione di contare i caratteri in un file di input *i*:

- se *i* è vuoto, restituisci 0;
- altrimenti, leggi il primo carattere in *i* e restituisci 1 più il numero di caratteri presenti nel resto del file *i*.

```
int contaCaratteri(FILE *i) {
    char c;
    c = fgetc(i);
    if (c == EOF)
        return 0;
    else
        return 1 + contaCaratteri(i);
}
```

### 8.3.2. Conteggio condizionato di elementi usando la ricorsione

Struttura della funzione ricorsiva per i file:

```
leggi un elemento;
if (!elemento valido)
   return 0;
else if (condizione)
   return 1 + risultato-della-chiamata-ricorsiva;
else
   return risultato-della-chiamata-ricorsiva;
```

Struttura della funzione ricorsiva per tipi indicizzati generici:

```
if (dato vuoto)
   return 0;
else if (condizione)
   return 1 + risultato ricorsione sui dati rimanenti;
else
   return risultato ricorsione sui dati rimanenti;
```

#### 8.3.2.1. Esempio: numero di occorrenze di un carattere in un file

Caratterizzazione ricorsiva dell'operazione di contare le occorrenze di un carattere *c* nel file di input *i*:

- se *i* è vuoto, restituisci 0;
- altrimenti, se il primo carattere di *i* è uguale a *c*, restituisci 1 più il numero di occorrenze di *c* nel resto del file *i*;
- altrimenti (ovvero se il primo carattere di *i* è diverso da *c*), allora restituisci il numero di occorrenze di *c* nel resto del file *i*.

```
int contaOccorrenzeCarattere(FILE *i, char x) {
    char c;
    c = fgetc(i);
    if (c == EOF)
        return 0;
    else if (c==x)
        return 1 + contaOccorrenzeCarattere(i,x);
    else
        return contaOccorrenzeCarattere(i,x);
}
```

### 8.3.3. Accumulazione usando la ricorsione

Supponiamo di voler effettuare un'operazione (ad esempio, la somma) tra tutti gli elementi di una collezione.

Struttura della funzione ricorsiva per i file:

```
leggi un elemento;
if (!elemento valido)
   return elemento-neutro-di-op;
```

```
else
   return valore-elemento op risultato-della-chiamata-ricorsiva;
```

dove elemento-neutro-di-op è l'elemento neutro rispetto all'operazione da effettuare (ad esempio, 0 per la somma, 1 per il prodotto, ecc.).

Struttura della funzione ricorsiva per tipi indicizzati generici:

```
if (dato vuoto)
   return elemento-neutro-di-op;
else
   return valore primo elemento <op>
          risultato ricorsione sui dati rimanenti;
```

#### 8.3.3.1. Esempio: somma di interi in un file di input

Caratterizzazione ricorsiva dell'operazione di sommare i valori letti da testiera:

- leggi un elemento *i*;
- se il file è terminato, restituisci 0;
- altrimenti, restituisci la somma tra il valore letto e la somma dei valori del resto del file.

```
int sommaValori(FILE *i) {
    int finefile;
    int v;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;
    else
        return v + sommaValori(i);
}
```

#### 8.3.3.2. Esempio: presenza di un valore in un insieme

Caratterizzazione ricorsiva dell'operazione di trovare un valore in un insieme di valori letti da tastiera:

- leggi un elemento $i$;
- se il file è terminato, restituisci false;
- altrimenti, se $i$ é il valore cercato, restituisci true;
- altrimenti procedi la ricerca nel resto del file.

```
int trovaValore(FILE *i, int x) {
    int v;
    int finefile;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;   // false
    else if (v==x)
        return 1;   // true
    else
        return trovaValore(i,x);
}
```

oppure

```
int trovaValore2(FILE *i, int x) {
    int v;
    int finefile;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;
    else
        return (v==x) || trovaValore2(i,x);
}
```

#### 8.3.3.3. Esempio: massimo di interi positivi letti da file

Caratterizzazione ricorsiva dell'operazione di trovare il massimo tra i valori di un insieme di interi positivi:

- leggi un elemento $i$;
- se il file è terminato, restituisci 0;
- altrimenti,
    1. trova il massimo $m$ tra i valori rimanenti nel file;
    2. restituisci il maggiore tra $i$ ed $m$.

```
int massimo(FILE *i) {
    int v;
    int finefile;
    finefile = fscanf(i,"%d", &v);
    if (finefile == EOF)
        return 0;
    else {
        int m = massimo(i);
        if (m>v) return m;
        else return v;
        // oppure
        // return (m>v)?m:v;
    }
}
```

## 8.4. Ricorsione su stringhe e array

Nella ricorsione su stringhe tipicamente il caso base è dato dalla stringa vuota, l'operazone viene effettuata sul primo carattere della stringa e la chiamata ricorsiva viene invocata sulla parte restante della stringa.

Data una stringa char *s:

- condizione del caso base: *s == '\0'

- accesso al primo elemento: *s

- parte restante della stringa: s+1

Struttura ricorsiva sulle stringhe

```
void f(char *s) {
  if (*s=='\0') { // caso base
    ...
  }
  else { // caso ricorsivo
    ... *s  // operazione sul primo carattere
    ... f(s+1)  // chiamata ricorsiva
  }
}
```

*Esempio:* stampa di una stringa.

```
void stampa(char *s) {
  if (*s=='\0') { // caso base
    printf("\n");
  }
  else { // caso ricorsivo
    printf("%c", *s);  // stampa il primo carattere
    stampa(s+1);       // stampa la parte restante
  }
}
```

Nella ricorsione su array tipicamente il caso base è dato dall'array vuoto, l'operazone viene effettuata sul primo elemento dell'array e la chiamata ricorsiva viene invocata sulla parte restante dell'array.

Dato un array int v[] di dimensione n:
- condizione del caso base: n == 0
- accesso al primo elemento: v[0]
- parte restante dell'array: v+1 di dimentione n-1

Struttura ricorsiva sugli array

```
void f(int v[], int n) {
  if (n==0) { // caso base
    ...
  }
  else {  // caso ricorsivo
    ... v[0]  // operazione sul primo elemento
    ... f(v+1,n-1)  // chiamata ricorsiva
  }
}
```

*Esempio:* stampa di un array.

```
void stampa(int v[], int n) {
  if (n==0) { // caso base
    printf("\n");
  }
  else { // caso ricorsivo
    printf("%d ", v[0]);  // stampa il primo carattere
    stampa(v+1,n-1);      // stampa la parte restante
  }
}
```

La ricorsione sugli array può essere implementata anche incrementando un argomento indice.

*Esempio:* stampa di un array.

```
void stampa(int v[], int i, int n) {
  if (i==n) { // caso base
    printf("\n");
  }
  else { // caso ricorsivo
    printf("%d ", v[i]);  // stampa il primo carattere
    stampa(v,i+1,n);      // stampa la parte restante
  }
}
```

In questo caso si deve invocare la funzione con il valore i=0.

## 8.5. Evoluzione della pila dei RDA nel caso di funzioni ricorsive

Nel caso di funzioni ricorsive, i meccanismi con cui evolvono la pila dei RDA ed il program counter sono identici al caso di funzioni non ricorsive. Tuttavia, è importante sottolineare che un RDA è associato ad *un'attivazione di una funzione* e non ad una funzione.

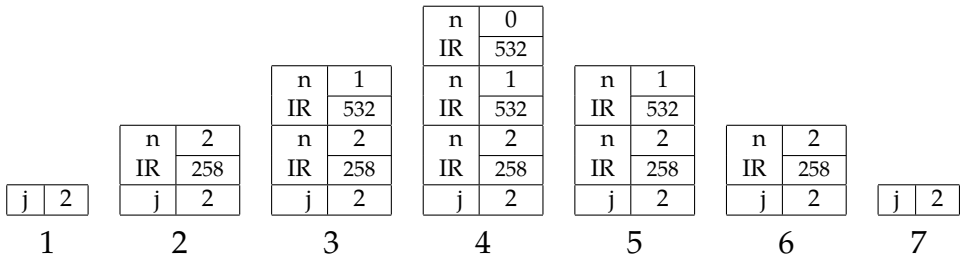
*Esempio:* consideriamo la seguente funzione ricorsiva e la sua attivazione dalla funzione main:

```
// funzione ausiliaria di stampa formattata
void stampa(int n, const char* str) {
    for (int i=0; i<10-n*2; i++) {
        printf(" ");
    }
    printf("ricorsivo(%d) - %s\n",n,str);
}

// funzione ricorsiva
void ricorsivo(int n) {
  if (n == 0) {
    stampa(n, "finito");
  }
  else {
    stampa(n,"attiva ricorsivo(n-1)");
    ricorsivo(n-1);
    stampa(n,"finito");
  }
}
```

```
int main() {
  int j;
  printf("Inserisci livello ricorsione\n");
  scanf("%d",&j);
  printf("Main - attiva ricorsivo(%d)\n",j);
  ricorsivo(j);
  printf("Main - Finito\n");
}
```

```
Inserisci livello ricorsione
3
Main - attiva ricorsivo(3)
    ricorsivo(3) - attiva ricorsivo(n-1)
      ricorsivo(2) - attiva ricorsivo(n-1)
        ricorsivo(1) - attiva ricorsivo(n-1)
          ricorsivo(0) - finito
        ricorsivo(1) - finito
      ricorsivo(2) - finito
    ricorsivo(3) - finito
Main - Finito
```

L'evoluzione della pila dei RDA per un livelo di ricersione 2 è mostrata qui di seguito. Abbiamo assunto che 258 sia l'indirizzo dell'istruzione che segue l'attivazione di ricorsivo(j) in main, e che 532 sia l'indirizzo dell'istruzione che segue l'attivazione di ricorsivo(n-1) in ricorsivo. Dal momento che le funzioni invocate non prevedono la restituzione di un valore di ritorno (il tipo di ritorno è void), i RDA non contengono una locazione di memoria per tale valore. Inoltre, non abbiamo indicato la funzione alla quale si riferisce ciascun RDA, in quanto il RDA in fondo alla pila è relativo a main, e tutti gli altri sono relativi ad attivazioni successive di ricorsivo.




Facciamo notare che per le diverse attivazioni ricorsive vengono creati diversi RDA sulla pila, con valori via via decrescenti del parametro

i, fino all'ultima attivazione ricorsiva, per la quale il parametro i assume valore 0. A questo punto non avviene più un'attivazione ricorsiva, viene stampato "finito", e l'attivazione termina. In cascata, avviene l'uscita dalle attivazioni precedenti, ogni volta preceduta dalla stampa di "ricorsivo(*i*) - finito".

Facciamo anche notare che codice associato alle diverse attivazioni ricorsive è sempre lo stesso, ovvero quello della funzione ricorsiva. Di conseguenza, l'indirizzo di ritorno memorizzato nei RDA per le diverse attivazioni ricorsive è sempre lo stesso (ovvero 532), tranne che per la prima attivazione, per la quale l'indirizzo di ritorno è quello di un'istruzione nella funzione main (ovvero 258).

## 8.6. Ricorsione multipla

Si ha **ricorsione multipla** quando un'attivazione di una funzione può causare *più di una attivazione ricorsiva* della stessa funzione.

*Esempio:* funzione ricorsiva per il calcolo dell'$n$-esimo numero di Fibonacci.

Fibonacci era un matematico pisano del 1200, interessato alla crescita di popolazioni. Ideò un modello matematico per stimare il numero di individui ad ogni generazione:

$F(n)$ ... numero di individui alla generazione $n$-esima

$$F(n) = \begin{cases} 0, & se\ n = 0 \\ 1, & se\ n = 1 \\ F(n-2) + F(n-1), & se\ n > 1 \end{cases}$$

$F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci, ed inizia con:
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Vediamo una funzione ricorsiva che, preso un intero positivo $n$, restituisce l'$n$-esimo numero di Fibonacci.

```
int fibonacci(int n) {
  if (n < 0) return -1;  // F(n) non e' definito per n
     negativo!
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

La chiamata ricorsiva attiverà due attivazioni della funzione ricorsiva.

### 8.6.1. Esempio: Torri di Hanoi

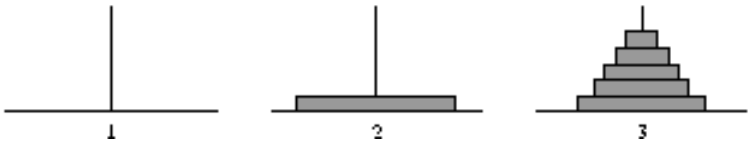
Il problema delle Torri di Hanoi ha origine da un'antica leggenda Vietnamita, secondo la quale un gruppo di monaci sta spostando una torre di 64 dischi (secondo la leggenda, quando i monaci avranno finito, verrà la fine del mondo). Lo spostamento della torre di dischi avviene secondo le seguenti regole:

- inizialmente, la torre di dischi di dimensione decrescente è posizionata su un perno 1;
- l'obiettivo è quello di spostarla su un perno 2, usando un perno 3 di appoggio;
- le condizioni per effettuare gli spostamenti sono:
  - tutti i dischi, tranne quello spostato, devono stare su una delle torri
  - è possibile spostare un solo disco alla volta, dalla cima di una torre alla cima di un'altra torre;
  - un disco non può mai stare su un disco più piccolo.

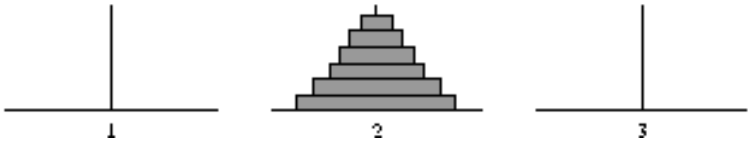
Lo stato iniziale (a), uno stato intermedio (b), e lo stato finale (c) per un insieme di 6 dischi sono mostrati nelle seguenti figure:

(a)


(b)




(c)




Vogliamo realizzare un programma che stampa la sequenza di spostamenti da fare. Per ogni spostamento vogliamo stampare un testo del tipo:

muovi un disco dal perno x al perno y

Idea: per spostare $n > 1$ dischi da 1 a 2, usando 3 come appoggio:

1. sposta $n - 1$ dischi da 1 a 3
2. sposta l'$n$-esimo disco da 1 a 2
3. sposta $n - 1$ dischi da 3 a 2

```
void muoviUnDisco(int sorg, int dest) {
    printf("muovi un disco da %d a %d\n", sorg, dest);
}

void muovi(int n, int sorg, int dest, int aux) {
    if (n == 1)
      muoviUnDisco(sorg, dest);
    else {
      muovi(n-1, sorg, aux, dest);
      muoviUnDisco(sorg, dest);
      muovi(n-1, aux, dest, sorg);
    }
  }
```

```
int main () {
    printf("Quanti dischi vuoi muovere?\n");
    int n;
    scanf("%d",&n);
    printf("Per muovere %d dischi da 1 a 2 con 3 come
    appoggio:\n",n);
    muovi(n, 1, 2, 3);
}
```

#### 8.6.1.1. Numero di attivazioni nel caso di ricorsione multipla

Quando si usa la ricorsione multipla, bisogna tenere presente che il numero di attivazioni ricorsive potrebbe essere *esponenziale* nella profondità delle chiamate ricorsive (cioè nell'altezza massima della pila dei RDA).

*Esempio:* Torri di Hanoi

$att(n)$ = numero di attivazioni di muoviUnDisco per $n$ dischi
= numero di spostamenti di un disco

$$att(n) = \begin{cases} 1, & se\ n = 1 \\ 1 + 2 \cdot att(n-1), & se\ n > 1 \end{cases}$$

Senza "$1+$" nel caso di $n > 1$, avremmo $att(n) = 2^{n-1}$. Ne segue che $att(n) > 2^{n-1}$.

Si noti che nel caso del problema delle Torri di Hanoi il numero esponenziale di attivazioni è una caratteristica intrinseca del problema, nel senso che non esiste una soluzione migliore.

### 8.6.2. Esercizio: attraversamento di una palude

Si consideri un'area paludosa costituita da $R \times C$ zone quadrate, con $R$ ed $C$ noti, ognuna delle quali può essere una zona di terraferma (transitabile) o una zona di sabbia mobile (non transitabile). Ogni zona della palude è identificata da una coppia di coordinate $\langle r, c \rangle$, con $0 \leq r < R$ e $0 \leq c < C$. Diremo che $r$ rappresenta la *riga* e $c$ la *colonna* della zona $\langle r, c \rangle$. Per *passaggio* si intende una sequenza di zone di terraferma adiacenti che attraversano la palude da sinistra (colonna pari a 0) a destra (colonna pari ad $C - 1$). Siamo interessati ai passaggi in cui ad ogni passo ci si muove verso destra, per cui da una zona in colonna $c$ si va ad una zona in colonna $c + 1$. In altre parole, la zona in posizione $\langle r, c \rangle$ si considera adiacente alle zone in posizione $\langle r - 1, c + 1 \rangle$, $\langle r, c + 1 \rangle$ e $\langle r + 1, c + 1 \rangle$, come mostrato nella seguente figura.

Nella figura che segue, il carattere '*' rappresenta una zona di terraferma mentre il carattere 'o' rappresenta una zona di sabbia mobile. La palude 1 è senza passaggi, mentre la palude 2 ha un passaggio (evidenziato).

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | * | o | o | * | o | o |
| 1 | o | * | o | o | o | o |
| 2 | o | o | * | * | o | o |
| 3 | * | * | o | o | o | o |
| 4 | * | * | * | o | * | * |

Palude 1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | * | o | o | * | o | o |
| 1 | * | o | o | o | o | o |
| 2 | o | * | o | o | o | * |
| 3 | o | o | * | * | * | o |
| 4 | o | * | o | o | o | o |

Palude 2

Si richiede di verificare l'esistenza di almeno un passaggio e stamparlo se esiste (se ne esiste più di uno è sufficiente stampare il primo trovato).

#### 8.6.2.1. Palude: rappresentazione di una palude

Per rappresentare una palude, definiamo un array di interi, e realizziamo un insieme di operazioni che consentono di utilizzarla:

- inizializzazione di una palude casuale, dati il numero di righe e di colonne, ed un valore reale compreso tra 0 e 1 che rappresenta la probabilità che una generica zona sia di terraferma;
- verifica se la zona di coordinate $\langle r, c \rangle$ è di terra;
- stampa la palude utilizzando i caratteri * e o per rappresentare le zone di terraferma e di sabbie mobili, rispettivamente.

```
int const righe = 10;
int const colonne = 10;
double probTerra = 0.5;

// dichiara la palude
int palude[righe][colonne];

// Operazioni sulla palude
int terra(int r, int c) {
    return (r >= 0) && (r < righe) &&
      (c >= 0) && (c < colonne) &&
      palude[r][c];
}
```

```
  void initPalude(double probTerra) {
     srand( time(NULL) );
     for (int r = 0; r < righe; r++)
        for (int c = 0; c < colonne; c++)
           palude[r][c] = rand()/(double)RAND_MAX <
    probTerra;
  }

void stampaPalude () {
    for (int r = 0; r < righe; r++) {
      for (int c = 0; c < colonne; c++)
         printf(palude[r][c]? "*" : "o");
      printf("\n");
    }
}
```

#### 8.6.2.2. Palude: soluzione dell'attraversamento

La soluzione richiede di trovare una sequenza di zone della palude, in cui la prima posizione sia in colonna 1, mentre l'ultima sia in colonna $C$. Ogni posizione della sequenza deve essere adiacente alla successiva. Per esempio, se la prima posizione è $\langle 3, 0\rangle$, la seconda può essere $\langle 4, 1\rangle$, ma non $\langle 3, 2\rangle$. Dal momento che ad ogni passo dobbiamo muoverci verso destra, il percorso sarà lungo esattamente $C$ passi.

Per esplorare la palude scegliamo di utilizzare un metodo ricorsivo. Questa scelta è la più intuitiva, dal momento che il processo di ricerca è inerentemente ricorsivo. L'algoritmo si può riassumere in questo modo: al primo passo si cerca una zona di terraferma nella prima colonna. Se c'è, si parte da quel punto. Al passo generico, ci si trova in una posizione $\langle r, c\rangle$. Se la posizione è di terraferma si può proseguire e si invoca ricorsivamente la ricerca sulle posizioni adiacenti, ovvero $\langle r-1, c+1\rangle$, $\langle r, c+1\rangle$ ed $\langle r+1, c+1\rangle$. Se invece la zona è di sabbia mobile non si può proseguire e la ricerca da quella zona termina. La ricerca termina quando si arriva ad una zona sull'ultima colonna, ovvero $c$ coincide con $colonne - 1$ e questa zona è una zona di terraferma.

Il generico passo di ricerca può essere implementato attraverso il seguente metodo ricorsivo cercaCammino, che riceve come parametri le coordinate $\langle r, c\rangle$ della zona dalla quale cercare il cammino.

```
int cercaCammino(int r, int c) {
  if (coordinate <r,c> della zona di palude non valide
      || <r,c> è una zona di sabbie mobili)
    return false;
  else if (<r,c> è sul bordo destro della palude)
    return true;
  else
    return cercaCammino(r-1, c+1) ||
           cercaCammino(r  , c+1) ||
           cercaCammino(r+1, c+1);
}
```

La funzione cercaCammino verifica solo se esiste un cammino da una posizione generica $\langle r, c\rangle$ fino all'ultima colonna. Dal momento che sono validi i cammini da una qualsiasi posizione della prima colonna, è necessario richiamare questo metodo in successione sulle posizioni $\langle r, 0\rangle$ della prima colonna, fino a quando non si è trovato un cammi-

no oppure si è arrivati all'ultima riga. Questo viene fatto dal metodo attraversaPalude.

#### 8.6.2.3. Palude: costruzione del cammino

Per rappresentare un cammino, facciamo notare che esso deve avere lunghezza pari al numero di colonne della palude, e che le zone attraversate dal cammino stanno su righe successive a partire da 0 fino ad arrivare alla colonna più a destra. Quindi, possiamo usare un array di *colonne* elementi interi, nel quale il valore del generico elemento di indice $c$ è pari all'indice $r$ di riga della posizione $\langle r, c \rangle$ attraversata dal cammino. Ad esempio il cammino evidenziato nella palude 2 di sopra è rappresentato dall'array {1,2,3,3,3,2}.

Nella seguente implementazione abbiamo scelto di aggiungere ai metodi cercaCammino e attraversaPalude un ulteriore parametro di tipo array di interi che rappresenta un cammino, e fare in modo che tali metodi facciano side-effect aggiornando in modo opportuno il cammino.

```
int cercaCammino(int r, int c, int camm[]) {
    if (!terra(r,c))
      return 0;
    else {
      camm[c] = r;
      if (c == colonne-1)
        return 1;
      else
        return cercaCammino(r-1, c+1, camm) ||
               cercaCammino(r,   c+1, camm) ||
               cercaCammino(r+1, c+1, camm);
    }
}
```

```
int attraversaPalude(int camm[]) {
   for (int r = 0; r < righe; r++)
      if (cercaCammino(r, 0, camm)) return 1;
   return 0;
}

int main() {
    initPalude();
    stampaPalude();
    int cammino [colonne];
    for (int c=0; c < colonne; c++) cammino[c] = 0;
    if (attraversaPalude(cammino))
        for (int c=0;  c < colonne; c++) printf("%d",
    cammino[c]);
    else
        printf("Cammino: cammino inesistente");
    printf("\n");
}
```

# 9. Strutture collegate lineari

## 9.1. Limitazioni degli array

L'uso di array per memorizzare collezioni di oggetti presenta alcune limitazioni nella gestione della memoria:

- la dimensione dell'array è stabilita al momento della sua creazione e può essere inefficiente modificarla successivamente;
- l'array utilizza uno spazio di memoria proporzionale alla sua dimensione, indipendentemente dal numero di elementi validi effettivamente memorizzati;
- per mantenere un ordinamento degli oggetti della collezione e inserire (o rimuovere) un valore in una posizione specifica dobbiamo fare uno spostamento per una buona parte degli elementi dell'array.

Le strutture collegate che introduciamo in questa unità sono definite quindi appositamente per consentire di allocare e deallocare memoria in maniera dinamica, a seconda delle esigenze del programma nella memorizzazione dei dati di interesse per l'applicazione.

## 9.2. Strutture collegate

Un meccanismo molto flessibile per la gestione dinamica della memoria è dato dalle *strutture collegate*, che vengono realizzate in maniera tale da consentire facilmente la modifica della propria struttura, gli inserimenti e le cancellazioni in posizioni specifiche, ecc.

In questa unità vedremo le *strutture collegate lineari* (SCL), che consentono quindi di memorizzare collezioni di oggetti organizzate sotto forma di sequenze (cioè in cui ogni elemento ha un solo successore).

Successivamente vedremo invece gli alberi, che sono un esempio di strutture collegate non lineari.

### 9.2.1. Dichiarazione di una Struttura Collegata Lineare

Una struttura collegata lineare (SCL) è definita tramite record.

```
typedef ... TipoInfoSCL;

struct ElemSCL {
  TipoInfoSCL info;
  struct ElemSCL *next;
};

typedef struct ElemSCL TipoNodoSCL;
typedef TipoNodoSCL * TipoSCL;
```

TipoInfoSCL è il tipo dei dati contenuti nella SCL.

TipoSCL è il tipo puntatore alla struttura che definisce i nodi (record di tipo TipoNodoSCL) della SCL.

### 9.2.2. Operazioni sulle strutture collegate lineari

Le operazioni più comuni sulle strutture collegate lineari sono:
operazioni che non modificano la SCL

- *scrittura* (su file)
- *ricerca* di un'informazione nella struttura
- calcolo della *dimensione* della struttura

operazioni che modificano il contenuto (ma non la struttura) della SCL

- *modifica* di elementi

operazioni che modificano la struttura della SCL

- *creazione*
- *inserimento* ed *eliminazione*

- *lettura* (da file)
- creazione tramite *copia*
- *deallocazione*

### 9.2.3. Creazione e collegamento di nodi

Le operazioni di base sulle SCL sono: l'inizializzazione di una SCL vuota, la creazione di un nodo e il collegamento tra due nodi.

SCL vuota:

```
TipoSCL scl = NULL;
```

Creazione di un nodo:

```
TipoSCL scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL
    ));
scl->info = e1;
scl->next = NULL;
```

Collegamento di nodi:

```
TipoSCL temp = (TipoNodoSCL*) malloc(sizeof(
    TipoNodoSCL));
temp->info = e2;
temp->next = NULL;
scl->next = temp;
```

## 9.3. Operazioni sul nodo in prima posizione

Le funzioni base per la gestione delle SCL sono inserimento ed eliminazione del nodo che si trova in prima posizione.

### 9.3.1. Inserimento di un nodo in prima posizione

```
void addSCL(TipoSCL *scl, TipoInfoSCL e) {
  TipoSCL temp = *scl;
  *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
  (*scl)->info = e;
  (*scl)->next = temp;
}
```

Si noti che la funzione addSCL opera correttamente anche se la struttura collegata è inizialmente vuota, nel qual caso viene costruita una SCL con un solo elemento.

*Esempio:* Il seguente frammento di codice genera una SCL contenente i valori (1, 2, 3).

```
TipoSCL scl = NULL;
addSCL(&scl,3);
addSCL(&scl,2);
addSCL(&scl,1);
```

### 9.3.2. Eliminazione di un nodo in prima posizione

```
void delSCL(TipoSCL * scl) {
  TipoSCL temp = *scl;
  *scl = (*scl)->next;
  free(temp);
}
```

Si noti che la funzione delSCL si comporta correttamente anche se la struttura collegata è costituita da un solo elemento, nel qual caso viene restituita una SCL vuota.

*Esempio:* Il seguente frammento di codice dealloca una SCL contenente tre elementi.

```
...
delSCL(&scl);
delSCL(&scl);
delSCL(&scl);
```

## 9.4. Ricorsione per le operazioni su SCL

L'implementazione ricorsiva delle operazioni sulle strutture collegate lineari risulta generalmente più semplice di quella iterativa. Ciò deriva dal modo in cui sono definite le strutture collegate. Una struttura collegata è:

- vuota;
- formata un elemento seguito da una struttura collegata.

### 9.4.1. Schema di ricorsione per SCL

Schema ricorsivo di lettura

```
if (SCL vuota)  {
  passo base
}
else {
  elaborazione primo elemento della SCL
  chiamata ricorsiva sul resto della SCL
}
```

Schema ricorsivo di scrittura

```
leggi dato
if (dato valido)  {
  inserimento del dato nella SCL
  chiamata ricorsiva sui dati rimanenti
}
```

### 9.4.2. Verifica SCL vuota

```
// restituisce true se scl e' vuota
int emptySCL(TipoSCL scl) {
  return scl == NULL;
}
```

## 9.5. Operazioni che non modificano la SCL

Le operazioni che non modificano la SCL sono definite come funzioni in cui il parametro usato per indicare la SCL è di tipo TipoSCL.

Ad esempio, la funzione di scrittura definita con void writeSCL(TipoSCL scl) viene invocata come segue:

```
TipoSCL scl;
...
writeSCL(scl);
```

Si noti che questo passaggio di parametri non garantisce comunque che la struttura o il contenuto della SCL rimangano invariati.

### 9.5.1. Scrittura di una SCL

Scrittura dell'informazione del nodo:

```
// scrittura dell'informazione nel nodo (esempio int)
void writeTipoInfo(TipoInfoSCL d) {
  printf("%d ",d);
}
// scrittura tramite file
void writeTipoInfoF(FILE *f, TipoInfoSCL d) {
  fprintf(f,"%d ",d);
}
```

Scrittura della SCL:

```
void writeSCL(TipoSCL scl) {
  if (emptySCL(scl))
    printf("\n");
  else {
    writeTipoInfo(scl->info);
    writeSCL(scl->next);
  }
}
```

Scrittura su file:

```
void writeSCLF(char *nfile, TipoSCL scl) {
  FILE *datafile;
  datafile = fopen(nfile, "w");
  if (datafile == NULL) {  // errore in apertura in
    scrittura
    printf("Errore apertura file '%s' in scrittura\n",
     nfile);
  }
  else {
       writeSCLF_r(datafile, scl);
       fclose(datafile);
  }
}
```

Funzione di scrittura ricorsiva:

```
void writeSCLF_r(FILE *o, TipoSCL scl) {
  if (!emptySCL(scl)) {
    writeTipoInfoF(o, scl->info);
    writeSCLF_r(o, scl->next);
  }
}
```

### 9.5.2. Verifica presenza di un elemento

```
// restituisce true se trova e in scl
int isinSCL(TipoSCL scl, TipoInfoSCL e) {
  if (emptySCL(scl))
    return 0;
  else if (scl->info==e)
    return 1;
  else
    return isinSCL(scl->next, e);
}
```

*Nota*: in generale, il confronto tra le informazioni contenute nei nodi viene effettuato con l'operatore == per i tipi di dati primitivi, ma occorre definire funzioni specifiche per i tipi strutturati definiti dall'utente.

### 9.5.3. Ricerca

```
// restituisce il puntatore al nodo che contiene e,
// se trova e in scl, altrimenti NULL
void findSCL(TipoSCL scl, TipoInfoSCL e, TipoSCL *ris)
     {
  if (emptySCL(scl))
    *ris = NULL;
  else if (scl->info==e)
    *ris = scl;
  else {
    findSCL(scl->next, e, ris);
  }
  return;
}
```

### 9.5.4. Lunghezza

```
// restituisce la lunghezza della scl
int lengthSCL(TipoSCL scl) {
  if (emptySCL(scl))
    return 0;
  else
    return 1 + lengthSCL(scl->next);
}
```

## 9.6. Operazioni che modificano il contenuto della SCL

Le operazioni che modificano il contenuto della SCL ma non la sua struttura sono definite come funzioni in cui il parametro usato per indicare la SCL è di tipo TipoSCL.

Ad esempio, la funzione di scrittura definita con void substElemSCL(TipoSCL scl, TipoInfoSCL e, TipoInfoSCL n) viene invocata come segue:

```
TipoSCL scl;
...
substElemSCL(scl, e, n);
```

Si noti che questo passaggio di parametri non garantisce comunque che la struttura della SCL rimanga invariata.

### 9.6.1. Modifica dell'informazione in un nodo

Sia p (il riferimento a) un nodo qualsiasi della struttura collegata, vogliamo modificare la sua informazione con il valore della stringa x.

```
TipoNodoSCL * p = ... // p e' il riferimento al nodo
TipoInfoSCL x = ... // x e' il nuovo valore da
    memorizzare in p
p->info = x;
```

In questo caso non è necessario modificare la struttura collegata, ma solamente il contenuto dell'informazione del nodo in questione.

### 9.6.2. Sostituzione di un elemento

```
// sostituisce la prima occorrenza dell'elemento e di
    scl con n
void substElemSCL(TipoSCL scl, TipoInfoSCL e,
    TipoInfoSCL n) {
  if (!emptySCL(scl)) {
    if(scl->info==e)
      scl->info = n;
    else
      substElemSCL(scl->next, e, n);
  }
}
```

### 9.6.3. Sostituzione di occorrenze multiple

```
// sostituisce tutte le occorrenze dell'elemento e di
    scl con n
void substNElemSCL(TipoSCL scl, TipoInfoSCL e,
    TipoInfoSCL n) {
  if (!emptySCL(scl)) {
    if(scl->info==e)
      scl->info = n;
    substNElemSCL(scl->next, e, n);
  }
}
```

## 9.7. Operazioni che modificano la SCL

Le operazioni che modificano la struttura della SCL sono definite come funzioni in cui il parametro usato per indicare la SCL è di tipo TipoSCL *.

Ad esempio, la funzione di scrittura definita con void readSCL(TipoSCL *scl) viene invocata come segue:

```
TipoSCL scl;
...
readSCL(&scl);
```

### 9.7.1. Costruzione di una SCL

Costruzione di una SCL di n elementi inizializzati con il valore e:

```
void creaSCL(TipoSCL *scl, int n, TipoInfoSCL e) {
  if (n == 0)
    *scl = NULL;
  else {
    *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
    (*scl)->info = e;
    creaSCL(&((*scl)->next), n-1, e);
  }
}
```

### 9.7.2. Lettura di una SCL

Lettura dell'informazione del nodo:

```
// lettura dell'informazione nel nodo (esempio int)
int readTipoInfo(TipoInfoSCL *d) {
  return scanf("%d",d);
}

// lettura tramite file
int readTipoInfoF(FILE *f, TipoInfoSCL *d) {
  return fscanf(f,"%d",d);
}
```

Lettura della SCL:

```
void readSCL(TipoSCL *scl) {
  TipoInfoSCL dat;
  if (readTipoInfo(&dat)==EOF)
    *scl = NULL;
  else {
    *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
    (*scl)->info = dat;
    readSCL(&((*scl)->next));
  }
}
```

Variante della lettura con addSCL:

```
void readAddSCL(TipoSCL * scl) {
  TipoInfoSCL dat;
  if (readTipoInfo(&dat)==EOF)
    *scl = NULL;
  else {
    readAddSCL(scl);
    addSCL(scl,dat);
  }
}
```

Lettura da file:

```
void readSCLF(char *nfile, int *n, TipoSCL *scl) {
  FILE *datafile;
  datafile = fopen(nfile, "r");
  (*n) = 0;
  if (datafile == NULL) {  // errore in apertura in
    lettura
        printf("Errore apertura file '%s' in lettura\n
    ", nfile);
    *scl = NULL;
  }
  else
    readSCLF_r(datafile, n, scl);
    fclose(datafile);
}
```

Funzione di lettura ricorsiva:

```
void readSCLF_r(FILE *i, int *n, TipoSCL *scl) {
  TipoInfoSCL dat;
  if (readTipoInfoF(i, &dat)==EOF)
    *scl = NULL;
  else {
    (*n)++;
    *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
    (*scl)->info = dat;
    readSCLF_r(i, n, &((*scl)->next));
  }
}
```

### 9.7.3. Copia

```
// copia scl in ris
void copySCL(TipoSCL scl, TipoSCL *ris) {
  if (emptySCL(scl))
    *ris=NULL;
  else {
    *ris = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
    (*ris)->info = scl->info;
    copySCL(scl->next, &((*ris)->next));
  }
}
```

### 9.7.4. Eliminazione

```
void eliminaSCL(TipoSCL *scl) {
  if (*scl != NULL) {
    TipoSCL p = *scl;
    eliminaSCL(&((*scl)->next));
    free(p);
  }
}
```

## 9.8. Operazioni basate sulla posizione

Le operazioni basate sulla posizione più comuni sono:

- ricerca dell'elemento in posizione n
  void findPosSCL(TipoSCL scl, int n, TipoSCL * ris)

- inserimento di un elemento in posizione n
  void addPosSCL(TipoSCL * scl, TipoInfoSCL e, int n);

- eliminazione di un elemento in posizione n
  void delPosSCL(TipoSCL * scl, int n);

Numeriamo gli elementi di una SCL a partire da 0.

### 9.8.1. Ricerca di un elemento tramite posizione

```
// restituisce il puntatore al nodo in posizione n
void findPosSCL(TipoSCL scl, int n, TipoSCL *ris) {
  if (n == 0)
    *ris = scl;
  else
    findPosSCL(scl->next, n-1, ris);
}
```

### 9.8.2. Inserimento in posizione data

```
// inserisce un nodo in posizione n
void addPosSCL(TipoSCL * scl, TipoInfoSCL e, int n){
  if (n == 0) {
    TipoSCL temp = *scl;
    *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
    (*scl)->info = e;
    (*scl)->next = temp;
  }
  else
    addPosSCL(&((*scl)->next), e, n-1);
}
```

### 9.8.3. Eliminazione di un elemento in posizione data

```
// elimina un nodo in posizione n
void delPosSCL(TipoSCL *scl, int n) {
  if (scl == NULL) {
    return;
  }
  else if (n == 0) {
    TipoSCL temp = *scl;
    (*scl) = (*scl)->next;
    free(temp);
  }
  else
    delPosSCL(&((*scl)->next), n-1);
}
```

## 9.9. Operazioni iterative su SCL

Per effettuare operazioni su tutti gli elementi di una struttura collegata lineare, oppure su uno specifico elemento caratterizzato da una proprietà, è possibile anche effettuare cicli di scansione per accedere agli elementi.

Lo schema di ciclo per accedere a tutti i nodi della struttura il cui primo nodo è puntato dalla variabile scl è il seguente.

```
TipoSCL scl = ...
TipoSCL p = scl;
while (p!=NULL) {
   elaborazione del nodo puntato da p
   p = p->next;
}
```

### 9.9.1. Operazioni che non modificano la SCL

Scrittura

```
void writeSCLi(TipoSCL scl) {
  while (!emptySCL(scl)) {
    writeTipoInfo(scl->info);
    scl=scl->next;
  }
  printf("\n");
}
```

Verifica presenza di un elemento

```
// restituisce true se trova e in scl
int isInSCLi(TipoSCL scl, TipoInfoSCL e) {
  int trovato = 0;
  while (!emptySCL(scl) && !trovato) {
    if (scl->info==e)
      trovato = 1; /* forza l'uscita dal ciclo */
    else
      scl=scl->next; /* aggiorna scl al resto della
    lista */
      }
    return trovato;
}
```

Lunghezza

```
// restituisce la lunghezza della scl
int lengthSCLi(TipoSCL scl) {
  int cont = 0;
  while (!emptySCL(scl)) {
    cont++;
    scl=scl->next;
      }
    return cont;
}
```

### 9.9.2. Operazioni che modificano il contenuto della SCL

Sostituzione di un elemento

```
// sostituisce la prima occorrenza dell'elemento e di
    scl con n
void substElemSCLi(TipoSCL scl, TipoInfoSCL e,
    TipoInfoSCL n) {
  int trovato = 0;
  while (!emptySCL(scl) && !trovato) {
    if(scl->info==e) {
      scl->info = n;
      trovato = 1;
    }
    else
      scl = scl->next;
  }
}
```

Sostituzione di un elemento

```
// sostituisce tutte le occorrenze dell'elemento e di
    scl con n
void substNElemSCLi(TipoSCL scl, TipoInfoSCL e,
    TipoInfoSCL n) {
  while (!emptySCL(scl)) {
    if(scl->info==e)
      scl->info = n;
    scl = scl->next;
  }
}
```

### 9.9.3. Operazioni che modificano la struttura della SCL

Costruzione

Si usa usa la tecnica del *nodo generatore*.

Il nodo generatore è un nodo ausiliario che viene anteposto alla lista che vogliamo costruire e da cui partono le operazioni di creazione dei nodi successivi. Al termine della creazione della lista tale nodo viene rimosso e viene restituita la lista a partire dal nodo successivo. Tale soluzione è usata per trattare uniformemente tutti gli elementi della lista, compreso il primo. Si osservi infatti che il metodo è corretto anche nel caso in cui la lista su cui viene invocato sia la lista vuota.

```
void creaSCLi(TipoSCL *scl, int n, TipoInfoSCL e) {
  TipoSCL prec;  // puntatore a elemento precedente
  // creazione del nodo generatore
  prec = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
  // scansione e creazione della lista
  *scl = prec;
  while (n > 0) {
    prec->next = (TipoNodoSCL*) malloc(sizeof(
    TipoNodoSCL));
    prec = prec->next;
    prec->info = e;
    n--;
  }
  prec->next = NULL; // chiusura della scl
  // eliminazione del nodo generatore
  prec = *scl;
  *scl = (*scl)->next;
  free(prec);
}
```

Lettura da file

```
void readSCLFi(char *nfile, int * n, TipoSCL * scl) {
  FILE *datafile;
  datafile = fopen(nfile, "r");
  (*n) =0;
  if (datafile == NULL) {  // errore in apertura in
    lettura
        printf("Errore apertura file '%s' in lettura\n
    ", nfile);
    *scl = NULL;
    return;
  }
  TipoInfoSCL elem;
  /* creazione del nodo generatore */
  *scl = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
  TipoSCL paux = *scl;
     /* ciclo lettura */
  while (readTipoInfoF(i, &elem) != EOF) {
        /* allocazione di un nodo e lettura del nuovo
    elemento */
      paux->next = (TipoNodoSCL*) malloc(sizeof(
    TipoNodoSCL));
      paux = paux->next;
      paux->info = elem;
    (*n)++;
  }
  paux->next = NULL;  /* chiusura della scl */
     /* cancellazione del record generatore */
  paux = *scl;
  *scl = (*scl)->next;
  free(paux);
  fclose(datafile);
}
```

Copia

```
void copySCLi(TipoSCL scl, TipoSCL *ris) {
  TipoSCL prec;  // puntatore a elemento precedente
  // creazione del nodo generatore
  prec = (TipoNodoSCL*) malloc(sizeof(TipoNodoSCL));
  // scansione e copia della lista
  *ris = prec;
  while (!emptySCL(scl)) {
    // copia dell'elemento corrente
    prec->next = (TipoNodoSCL*) malloc(sizeof(
    TipoNodoSCL));
    prec = prec->next;
    prec->info = scl->info;
    scl = scl->next;
  }
  prec->next = NULL; // chiusura della scl
  // eliminazione del nodo generatore
  prec = *ris;
  *ris = (*ris)->next;
  free(prec);
}
```

Inserimento in posizione n

```
void addPosSCLi(TipoSCL *pscl, TipoInfoSCL e,
    TipoInfoSCL n) {
  // nodo generatore
  TipoSCL g = (TipoSCL) malloc(sizeof(TipoNodoSCL));
  g->next = *pscl;
  TipoSCL q = g;
  while (q->next!=NULL && q->next->info!=e) {
    q = q->next;
  }
  if (q->next!=NULL) {
    add(&(q->next),n);
  }
  *pscl=g->next;
  free(g);
}
```

Eliminazione primo nodo contenente elemento dato

```
void eliminaInfoSCLi(TipoSCL *scl, TipoInfoSCL e) {
  /* si assume scl!=NULL e n>=0 */
  while (!emptySCL(*scl) && (*scl) -> info != e) {
    scl = &((*scl)-> next);
  }
  if (*scl != NULL){
    TipoSCL temp = *scl;
    *scl = temp -> next;
    free(temp);
  }
}
```

Eliminazione nodi contenenti elemento dato

```
void eliminaTuttiInfoSCLi(TipoSCL *scl, TipoInfoSCL e)
     {
  /* si assume scl!=NULL e n>=0 */
  TipoSCL temp;
  while (!emptySCL(*scl)) {
    if ((*scl) -> info == e){
      temp = *scl;
      *scl = temp -> next;
      free(temp);
    }
    scl = &((*scl)-> next);
  }
}
```

Eliminazione in posizione data

```
void eliminaPosSCLi(TipoSCL *scl, int n) {
  /* si assume scl!=NULL e n>=0 */
  while (!emptySCL(*scl) && n>0) {
    scl = &((*scl)-> next);
    n--;
  }
  TipoSCL temp = *scl;
  *scl = temp -> next;
  free(temp);
}
```

Eliminazione

```
void eliminaSCLi(TipoSCL *scl) {
  while (!emptySCL(*scl)) {
      TipoSCL p = *scl;
      *scl = (*scl)->next;
      free(p);
    }
}
```

Inversa

```
void invertiSCLi(TipoSCL *scl) {
  TipoSCL prec = NULL; /* elemento precedente */
  TipoSCL suc;         /* elemento successivo */
  while (!emptySCL(*scl)) {
    suc = *scl;
    *scl = (*scl)->next;
    suc->next = prec;
    prec = suc;
  }
  *scl = prec;
}
```

### 9.9.4. Operazioni che modificano la struttura della SCL (senza l'ausilio del nodo generatore)

Funzioni iterative che modificano la struttura di una SCL possono anche non utilizzare la tecnica del nodo generatore. In questo caso, si tiene in memoria, analogamente al caso ricorsivo, una variabile `TipoSCL*` che punta sempre a una lista rappresentata dal campo `next` del nodo precedente. Questa sezione contiene una serie di esempi che sfruttano questa tecnica.

Costruzione senza l'utilizzo del nodo generatore

```
void creaSCLip(TipoSCL *pscl, int n, TipoInfoSCL e) {
  while (n > 0) {
    add(pscl, e);
    n--;
  }
}
```

Copia senza l'utilizzo del nodo generatore

```
void copySCLip(TipoSCL scl, TipoSCL *pris) {
  *pris = NULL;
  while (scl != NULL) {
    addSCL(pris, scl->info);
    scl = scl->next;
    pris = &((*pris)->next);
}
```

Inserimento in posizione n senza l'utilizzo del nodo generatore

```
void addPosSCLip(TipoSCL *pscl, TipoInfoSCL e, int n)
    {
  while (n > 0 && *pscl!=NULL) {
    n--;
    pscl = &((*pscl)->next);
  }
  if (n == 0) {
    addSCL(pscl, e);
  }
}
```

## 9.10. SCL: Implementazione funzionale

Nelle implementazioni funzionali, il risultato delle operazioni viene restituito in una nuova SCL, anziché modificare una variabile già allocata.

*Esempio:*

```
TipoSCL addSCL(TipoSCL scl, TipoInfoSCL e);
```

invece di

```
void addSCL(TipoSCL *scl, TipoInfoSCL e);
```

### 9.10.1. Funzioni primitive

```
// restituisce true se scl e' NULL
int emptySCL(TipoSCL scl);

// restituisce il primo elemento di una scl non vuota
TipoInfoSCL primoSCL(TipoSCL scl);

// restituisce il resto di una scl non vuota
TipoSCL restoSCL(TipoSCL scl);

// aggiunge l'elemento e in prima posizione alla SCL
TipoSCL addSCL(TipoSCL scl, TipoInfoSCL e);
```

### 9.10.2. Primo

```
TipoInfoSCL primoSCL(TipoSCL scl) {
  if (!emptySCL(scl))
    return scl->info;
  else {
    printf("primo di una lista vuota");
    return ErrInfoSCL;
  }
}
```

### 9.10.3. Resto

```
TipoSCL restoSCL(TipoSCL scl) {
  if (!emptySCL(scl))
    return scl->next;
  else {
    printf("resto di una lista vuota");
    return NULL;
  }
}
```

### 9.10.4. Costruzione di una SCL

```
TipoSCL addSCL(TipoSCL scl, TipoInfoSCL e){
  TipoSCL temp = (TipoNodoSCL*) malloc(sizeof(
    TipoNodoSCL));
  temp->info = e;
  temp->next = scl;
  return temp;
}
```

### 9.10.5. Copia di una SCL

```
// copia la SCL
TipoSCL copySCL(TipoSCL scl){
  if (emptySCL(scl))
    return NULL;
  else
    return addSCL(copySCL(restoSCL(scl)),
                  primoSCL(scl));
}
```

### 9.10.6. Inserimento in posizione n

Implementazione tramite memoria condivisa

```
// aggiunge e in posizione n > 0
// assume che la SCL contenga almeno n-1 elementi
TipoSCL addPosSCL(TipoSCL scl, TipoInfoSCL e, int n) {
  if (n == 0)
    return addSCL(scl,e);
  else
    return addSCL(addPosSCL(restoSCL(scl),e,n-1),
                  primoSCL(scl));
}
```

Implementazione tramite copia

```
// aggiunge e in posizione n > 0
// assume che la SCL contenga almeno n-1 elementi
TipoSCL addPosSCL(TipoSCL scl, TipoInfoSCL e, int n) {
  if (n == 0)
    return addSCL(copySCL(scl),e);
  else
    return addSCL(addPosSCL(restoSCL(scl),e,n-1),
                  primoSCL(scl));
}
```

### 9.10.7. Eliminazione in posizione n

Implementazione tramite memoria condivisa

```
// restituisce una SCL senza l'elemento in posizione n
// assume che la SCL contenga almeno n elementi
TipoSCL delPosSCL(TipoSCL scl, int n) {
  if (n == 0)
    return restoSCL(scl);
  else
    return addSCL(delPosSCL(restoSCL(scl),n-1),
                  primoSCL(scl));
}
```

Implementazione tramite copia

```
// restituisce una SCL senza l'elemento in posizione n
// assume che la SCL contenga almeno n elementi
TipoSCL delPosSCL(TipoSCL scl, int n) {
  if (n == 0)
    return copySCL(restoSCL(scl));
  else
    return addSCL(delPosSCL(restoSCL(scl),n-1),
                  primoSCL(scl));
}
```

### 9.10.8. Modifica in posizione n

Implementazione tramite memoria condivisa

```
// modifica elemento in posizione n > 0 con valore e
// assume che la SCL contenga almeno n-1 elementi
TipoSCL setPosSCL(TipoSCL scl, TipoInfoSCL e, int n) {
  if (n == 0)
    return addSCL(scl->next,e);
  else
    return addSCL(setPosSCL(restoSCL(scl),e,n-1),
                  primoSCL(scl));
}
```

Implementazione tramite copia

```
// modifica elemento in posizione n > 0 con valore e
// assume che la SCL contenga almeno n-1 elementi
TipoSCL setPosSCL(TipoSCL scl, TipoInfoSCL e, int n) {
  if (n == 0)
    return addSCL(copySCL(scl->next),e);
  else
    return addSCL(addPosSCL(restoSCL(scl),e,n-1),
                  primoSCL(scl));
}
```

# 10. I tipi di dato astratti

## 10.1. Nozione di tipo astratto

La nozione di algoritmo è indipendente dalla sua codifica nel linguaggio di programmazione; analogamente, con *tipo di dato astratto* (o struttura dati astratta) si intende la specifica di un tipo di dato indipendente dalla sua implementazione e dal linguaggio di programmazione usato per la sua codifica. Intuitivamente, un tipo di dato astratto è una collezione di elementi su cui è possibile eseguire un insieme prefissato di operazioni.

Un *tipo di dato astratto* è costituito da tre componenti:

- il *dominio di interesse*, ovvero l'insieme degli elementi propri del tipo, ed eventuali *altri domini*, ovvero insiemi necessari ad eseguire le operazioni del tipo
- un insieme di *costanti*, usate per denotare valori particolari del dominio d'interesse
- un insieme di *funzioni*, rappresentative delle operazioni proprie del tipo, che operano sugli elementi del dominio di interesse, utilizzando, ove necessario, elementi degli eventuali altri domini.

Un tipo di dato astratto rappresenta la specifica matematica di un insieme di dati e delle operazioni ad esso associate. In quanto tale, essa è indipendente dalla modalità di rappresentazione dei dati e, a maggior ragione, dal particolare linguaggio di programmazione usato.

Una caratteristica fondamentale dei tipi di dato astratto è che a partire da essi si può progettare un algoritmo senza dover far riferimento ad una particolare rappresentazione (o implementazione) del tipo di dato

stesso né dello specifico linguaggio di programmazione usato. Questo comporta un'enorme semplificazione del processo di creazione di un programma, in quanto permette al progettista di focalizzarsi sulle operazioni da eseguire (cosa fare) piuttosto che sul modo in cui esse devono essere realizzate (come fare).

### 10.1.1. Il tipo astratto Booleano

Un semplice esempio di tipo di dato astratto è il tipo *Booleano*. La sua specifica informale è la seguente:

- dominio di interesse: {*vero*, *falso*}
- costanti: TRUE e FALSE, che denotano rispettivamente *vero* e *falso*
- funzioni: and, or, not

Le costanti possono anche essere definite tramite funzioni (senza argomenti). All'occorrenza, nel seguito sfrutteremo anche questa possibilità per definire le costanti di un tipo.

Una volta definite le operazioni primitive di un tipo, esse possono essere usate per definire funzioni più complesse. Si pensi, ad esempio, all'uso di espressioni booleane all'interno di un algoritmo: partendo dalle operazioni primitive del tipo, vengono definite funzioni arbitrariamente complesse.

### 10.1.2. Tipi di dato astratti comuni

In generale, è possibile specificare tipi di dato astratti di qualsiasi natura. Alcuni di essi, tuttavia, si distingono per la loro generalità, cui consegue una vasta diffusione e grande importanza. I tipi di dato astratto più comunemente utilizzati sono:

1. Lineari: liste, pile, code, insiemi
2. Non lineari: alberi binari, alberi n-ari, grafi

### 10.1.3. Utilizzi dei tipi astratti

La nozione di tipo astratto consente di concettualizzare:

- i tipi di dato utilizzati nella progettazione di algoritmi e nella realizzazione dei corrispondenti programmi, quali array, liste, insiemi,

pile, code, alberi, grafi ecc.; si usa spesso il termine *struttura di dati* per riferirsi a questi tipi di dato.

- dati di qualsiasi tipo, che si ritengono importanti in una applicazione.

È importante comprendere che un progettista software non è solo interessato a utilizzare tipi di dato consolidati e ampiamente studiati per realizzare programmi, ma anche a ideare nuovi tipi astratti per rappresentare attraverso essi il dominio di interesse di una applicazione.

Consideriamo, ad esempio, un'applicazione in cui sono rilevanti gli studenti iscritti ad un corso universitario. Possiamo pensare a queste informazioni come ad un tipo astratto che ha come dominio l'insieme degli studenti e come operazioni fondamentali, ad esempio, l'assegnazione di un numero di matricola, l'iscrizione ad un certo anno di corso, la scelta di un piano di studi, e così via.

## 10.2. Specifica di tipi astratti

Per fornire la specifica di un tipo astratto adottiamo un metodo basato sull'uso schematico del linguaggio naturale. Tale metodo, pur non essendo propriamente formale, ci permette di descrivere senza ambiguità i tipi astratti.

**TipoAstratto** T

**Domini**

D1 : descrizione del dominio D1

...

Dm : descrizione del dominio Dm

**Costanti**

C1 : descrizione della costante C1

...

Ck : descrizione della costante Ck

**Funzioni**

F1 : descrizione della funzione F1

...

**Fn** : descrizione della funzione Fn

**FineTipoAstratto**

Le descrizioni dei domini e delle costanti sono fornite in forma sintetica in linguaggio naturale, mentre la descrizione delle funzioni è più articolata. Tipicamente, D1 viene considerato il dominio di interesse.

### 10.2.1. Specifica di tipi astratti: Booleano

Forniamo, a titolo esemplificativo, la specifica formale del tipo di dato astratto *Booleano*.

**TipoAstratto Booleano**
**Domini**

**bool** : dominio di interesse { *vero*, *falso* }

**Costanti**

**TRUE** : il valore *vero*
**FALSE** : il valore *falso*

**Funzioni**

**and(bool a, bool b) $\mapsto$ bool**
pre: nessuna
post: RESULT è il risultato della congiunzione logica tra a e b

**or(bool a, bool b) $\mapsto$ bool**
pre: nessuna
post: RESULT è il risultato della disgiunzione logica tra a e b

**not(bool a) $\mapsto$ bool**
pre: nessuna
post: RESULT è il risultato della negazione logica di a

**FineTipoAstratto**

### 10.2.2. I tipi astratti come enti matematici

Si noti che la specifica di un tipo astratto non si riferisce in alcun modo né alla rappresentazione dei valori del dominio d'interesse, né al fatto che tali valori saranno poi eventualmente memorizzati nelle variabili del programma.

Le operazioni associate al tipo vengono descritte semplicemente come funzioni matematiche che calcolano valori a fronte di altri valori, e non come meccanismi che modificano variabili.

In altre parole, in fase di concettualizzazione, le operazioni associate ad un tipo astratto sono specificate mediante funzioni matematiche.

Nella successiva fase di realizzazione si procederà alla scelta di come tradurre le varie operazioni del tipo, e si potrà quindi realizzare ogni operazione o come una funzione che calcola nuovi valori (in linea con quanto descritto nella specifica) o come una funzione che modifica le variabili che rappresentano i dati sui quali è invocata.

## 10.3. Implementazione dei tipi di dato astratti

Il tipo di dato astratto deve essere realizzato usando i costrutti del linguaggio di programmazione:

1. rappresentazione dei *domini* usando i *tipi concreti* del linguaggio di programmazione
2. codifica delle *costanti* attraverso i *costrutti* del linguaggio di programmazione
3. realizzazione delle *operazioni* attraverso opportune *funzioni* del linguaggio di programmazione

### 10.3.1. Scelta dello schema realizzativo

Per *schema realizzativo* si intende la modalit à con cui un tipo astratto è effettivamente implementato. Essenzialmente, scegliere uno *schema realizzativo* corrisponde a prendere le seguenti due decisioni:

- se le funzioni effettuino o meno side-effect sugli elementi del dominio di interesse
- se le strutture dati coinvolte condividano o meno la memoria utilizzata per la rappresentazione degli elementi del tipo

La scelta dello schema realizzativo costituisce una delle scelte più critiche nella realizzazione di un tipo di dato; essa è dettata tanto da considerazioni relative alle prestazioni quanto da considerazioni di natura modellistica, ovvero dipendenti dall'entità che si vuole modellare con il tipo di dato in esame. In particolare, se questo viene usato per modellare *valori* (si pensi ad esempio a numeri complessi, punti del piano cartesiano, etc.) è tipicamente indicata une realizzazione che non effettui side-effect. Quando invece il tipo è introdotto allo scopo di modellare *entità* (ad es., persone, oggetti fisici, etc.) è normalmente privilegiata una realizzazione con side-effect. Nel primo caso, infatti, appare naturale pensare un valore come immutabile (il valore 3, in quanto valore, non

può essere trasformato in 4), mentre nel secondo ci si può ragionevolmente aspettare che un'entità possa cambiare alcune delle sue proprietà (ad es., potremmo cambiare il colore di una macchina). Queste osservazioni costituiscono delle linee guida generali, tuttavia la scelta dello schema realizzativo è particolarmente importante e deve essere valutata accuratamente caso per caso. In questo corso non vengono affrontate nel dettaglio le motivazioni a supporto di ciascuno schema realizzativo ma soltanto le conseguenze, sul piano dell'implementazione, della scelta effettuata.

### 10.3.2. Realizzazione delle operazioni

La prima scelta da effettuare riguarda il modo in cui si implementano le operazioni del tipo astratto. In base a questa scelta, classifichiamo gli schemi realizzativi come:

- **con side-effect**, nel caso in cui le funzioni effettuino modifiche sugli elementi del dominio di interesse (opportunamente rappresentati) forniti in input

- **funzionali**, nel caso in cui le funzioni restituiscano nuovi elementi come risultato, senza modificare quelli forniti in input.

### 10.3.3. Realizzazioni con side-effect

Nelle realizzazioni con side-effect le funzioni che realizzano le operazioni del tipo astratto eseguono side-effect sui dati di input.

Generalmente, questo schema realizzativo è più efficiente, in quanto permette di operare su dati già presenti in memoria, senza dover usare risorse (tempo, memoria) per la loro creazione.

Tuttavia, proprio la possibilità di effettuare side-effect richiede particolare attenzione nell'evitare il problema dell'*interferenza*, termine con cui si indica la modifica *indesiderata* di una struttura dati come conseguenza di un'operazione su un'altra struttura dati.

### 10.3.4. Realizzazioni funzionali

Nello schema realizzativo funzionale, le funzioni che implementano le operazioni del tipo astratto restituiscono *nuovi* valori del tipo come risultato dell'operazione, senza modificare i dati di input.

Le funzioni sono realizzate seguendo la specifica matematica dei tipi astratti di dato, risultando quindi molto eleganti dal punto di vista formale e semplici da usare.

Le realizzazioni funzionali possono però comportare problemi di inefficienza. Infatti, la necessità di restituire sempre nuovi elementi combinata con il vincolo di non eseguire side-effect può richiedere operazioni di copia (eventualmente parziale) delle strutture manipolate.

### 10.3.5. Condivisione di memoria tra dati

La seconda decisione il progettista deve prendere riguarda la modalità di gestione della memoria utilizzata per la rappresentazione dei dati. Tale decisione è particolarmente significativa quando il metodo di rappresentazione del tipo astratto prevede l'uso di strutture collegate.

Le alternative possibili sono le seguenti:

- **realizzazione senza condivisione di memoria**: le funzioni operano in modo da assicurare che le rappresentazioni di dati diversi non condividano mai memoria;
- **realizzazione con condivisione di memoria**: le funzioni non impediscono che le rappresentazioni di dati diversi condividano memoria.

Nelle realizzazioni con condivisione, le funzioni possono accedere alle strutture che modellano dati diversi da quelli di input. Esiste cioè la possibilità che dati distinti siano rappresentati mediante strutture (ad esempio collegate) che occupano la stessa locazione fisica. Ad esempio, due strutture collegate potrebbero condividere il contenuto a partire da un dato elemento in poi. Le funzioni che accedono ai dati condivisi devono essere realizzate prestando particolare attenzione a non compromettere la consistenza delle strutture che condividono tali dati (problema dell'interferenza).

Generalmente, la condivisione comporta notevoli benefici tanto dal punto di vista dell'efficienza quanto dell'occupazione della memoria.

### 10.3.6. Schemi realizzativi privilegiati

Per *schema realizzativo* s'intende una combinazione di scelte realizzative tra

- funzionali o con side-effect

- con o senza condivisione di memoria

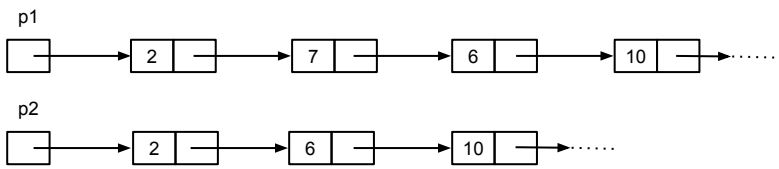
Conseguentemente abbiamo 4 schemi realizzativi possibili:

- funzionale, senza condivisione di memoria;
- funzionale, con condivisione di memoria.
- con side-effect, senza condivisione di memoria;
- con side-effect, con condivisione di memoria;

### 10.3.7. Funzionale, senza condivisione

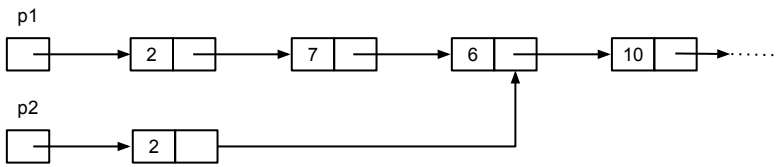
Es.: eliminazione del nodo in posizione 1

```
p2 = delPos(p1,1);
```




### 10.3.8. Funzionale, con condivisione

```
p2 = delPos(p1,1);
```

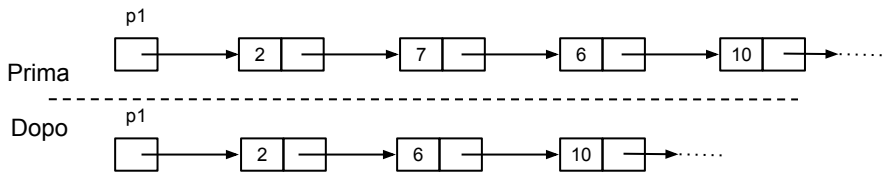



Osservazioni:

- prefisso a monte della modifica copiato (non condiviso)
- suffisso a valle della modifica condiviso (non copiato)

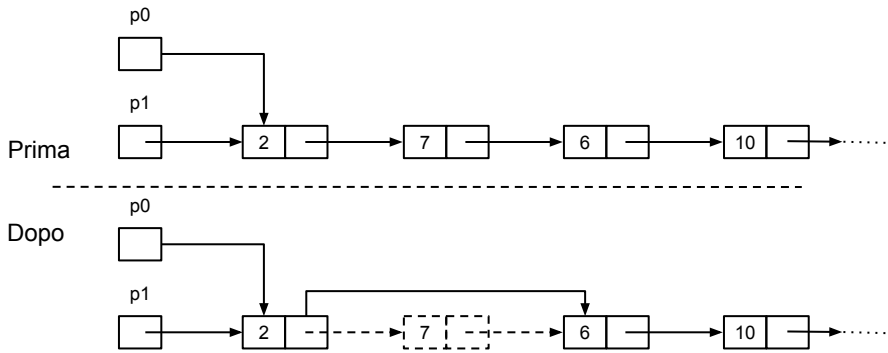
### 10.3.9. Side-effect, senza condivisione

```
delPos(&p1,1);
```




### 10.3.10. Side-effect, con condivisione

```
delPos(&p1,1);
```




In questo caso sono necessari meccanismi complessi di gestione dei puntatori per evitare interferenza.
Solo due schemi sono interessanti dal punto di vista pratico:

- lo schema con side-effect senza condivisione di memoria, per realizzare dati **mutabili**, cioè manipolati da funzioni che effettuano side-effect;
- lo schema funzionale con condivisione di memoria, per realizzare oggetti **immutabili**, cioè manipolati da funzioni che non effettuano mai side-effect.

Gli altri schemi comportano varie difficoltà implementative, non bilanciate da evidenti benefici. Ad esempio, uno schema con side-effect e condivisione di memoria richiede una gestione complessa della memoria al fine di evitare interferenza, senza che questo comporti un significativo risparmio della memoria usata, mentre uno schema funzionale senza condivisione di memoria tipicamente comporta spreco di memoria dovuto all'intutile replicazione di strutture dati uguali.

### 10.3.11. Implementazione in C del tipo Booleano

Mostriamo ora un esempio di realizzazione del tipo di dato *Booleano* nel linguaggio C. Scegliamo uno schema realizzativo funzionale senza condivisione di memoria, in quanto il tipo rappresenta un'astrazione di valori matematici. Si noti che la realizzazione proposta nel seguito ha solo scopi didattici e non risulta particolarmente conveniente. Rimane consigliato l'uso degli operatori booleani primitivi.

Per il dominio d'interesse, scegliamo di rappresentare:

- il valore *FALSO* con 0
- il valore *VERO* con 1.

Pertanto, rappresentiamo il dominio bool con il tipo int in C (di cui verranno, effettivamente, usati solo 2 valori):

```
typedef int bool;
```

Per le *Costanti* introduciamo la seguente definizione:

```
#define FALSE 0
#define TRUE 1
```

Per le *Operazioni* abbiamo le seguenti definizioni:

```
bool not(bool x) {
  if (x == TRUE)
    return FALSE;
  else
    return TRUE;
}

bool and(bool x, bool y) {
  if (x == TRUE && y == TRUE)
    return TRUE;
  else
    return FALSE;
}

bool or(bool x, bool y) {
  if (x == TRUE || y == TRUE)
    return TRUE;
  else
    return FALSE;
}
```

### 10.3.12. Osservazioni

- Per uno stesso tipo astratto si possono avere *più implementazioni* diverse.
- Una implementazione potrebbe avere delle *limitazioni* rispetto al tipo di dato astratto.
- Un tipo di dato può essere *parametrico*, cioè definito a partire da uno o più tipi di dato (i tipi che seguono ne sono un esempio).

Nel seguito ometteremo il termine "astratto", a cui abitualmente viene associata una caratterizzazione matematica e ci riferiremo semplicemente ai tipi di dato fornendo per essi una specifica semi-formale e diverse possibili implementazioni.

## 10.4. Tipo astratto NumeroComplesso

Un altro semplice esempio sono i numeri complessi, il cui tipo astratto può essere definito come segue:

**TipoAstratto** NumeroComplesso

**Domini**

C : dominio dei complessi – dominio di interesse

**R** : dominio di reali

**Funzioni**

**creaComplesso(R r, R i) $\mapsto$ C**
pre: nessuna
post: RESULT è il numero complesso avente r come parte reale e i come parte immaginaria

**Funzioni**

**reale(C c) $\mapsto$ R**
pre: nessuna
post: RESULT è il valore della parte reale del numero complesso c

**immaginaria(C c) $\mapsto$ R**
pre: nessuna
post: RESULT è il valore della parte immaginaria del numero complesso c

**modulo(C c) $\mapsto$ R**
pre: nessuna
post: RESULT è il modulo del numero complesso c

**fase(C c) $\mapsto$ R**
pre: nessuna
post: RESULT è la fase del numero complesso c

**FineTipoAstratto**

### 10.4.1. Realizzazione del tipo NumeroComplesso

Dominio di interesse: record con due campi di valori reali.

Dominio dei reali: si usa il tipo double del C.

```
typedef struct{
  double re, im;
}  NumeroComplesso;
```

Funzioni:

```
NumeroComplesso creaComplesso(double r, double i) {
  NumeroComplesso res;
  res.re = r; res.im = i;
  return res;
}

double reale(NumeroComplesso c) {return c.re;}

double immaginaria(NumeroComplesso c) {return c.im;}

double modulo(NumeroComplesso c) {
  return sqrt(pow(c.re,2)+pow(c.im,2));
}

double fase(NumeroComplesso c) {
  return atan2(c.im,c.re);
}
```

## 10.5. Tipo astratto Coppia

Il tipo astratto Coppia rappresenta il prodotto cartesiano di due domini. I valori di tale tipo sono coppie di valori presi da due domini specificati, e le operazioni associate sono semplicemente la costruzione, il calcolo del valore della prima componente, il calcolo del valore della seconda componente. La specifica del tipo Coppia è la seguente.

**TipoAstratto Coppia(T1,T2)**
**Domini**

**Coppia** : dominio di interesse del tipo
**T1** : dominio dei valori che possono comparire come prima componente delle coppie
**T2** : dominio dei valori che possono comparire come seconda componente delle coppie

**Funzioni**

**formaCoppia(T1 a, T2 b) $\mapsto$ Coppia**
pre: nessuna
post: RESULT è il valore corrispondente alla coppia la cui prima componente è a e la seconda è b

**primaComponente(Coppia c) $\mapsto$ T1**
pre: nessuna
post: RESULT è la prima componente della coppia c

**secondaComponente(Coppia c) $\mapsto$ T2**
pre: nessuna
post: RESULT è la seconda componente della coppia c

**FineTipoAstratto**

### 10.5.1. Realizzazione del tipo Coppia

Dominio di interesse: record con due campi

Altri domini: si usano i domini del linguaggio o tipi definiti dall'utente.

```
typedef ... T1;

typedef ... T2;

struct Coppia {
  T1 primo;
  T2 secondo;
};
```

Funzioni:

```
Coppia formaCoppia(T1 a, T2 b) {
  Coppia c;
  c.primo = a; c.secondo = b;
  return c;
}

T1 primaComponente(Coppia c) {
  return c.primo;
}

T2 secondaComponente(Coppia c) {
  return c.secondo;
}
```

## 10.6. Il tipo astratto Insieme

Un *insieme* è una collezione non ordinata di elementi omogenei, senza ripetizioni. Per denotare un insieme si usa abitualmente la notazione parentetica:

- {} denota l'insieme vuoto
- {e1, e2, ...} denota un insieme contenente gli elementi e1, e2, ecc.

Si noti che le notazioni { e1, e2 } e { e2, e1 } denotano lo stesso insieme.

**TipoAstratto Insieme(T)**

**Domini**

**Ins** : dominio di interesse

**T** : dominio degli elementi dell'insieme

**Funzioni**

**insiemeVuoto() $\mapsto$ Ins**
pre: nessuna
post: RESULT è l'insieme vuoto

**estVuoto(Ins i) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se i è il valore corrispondente all'insieme vuoto, false altrimenti

**Funzioni**

**inserisci(Ins i, T e) $\mapsto$ Ins**
pre: nessuna
post: RESULT è l'insieme ottenuto dall'insieme i aggiungendo l'elemento e; se e appartiene già a i allora RESULT coincide con i

**elimina(Ins i, T e) $\mapsto$ Ins**
pre: nessuna
post: RESULT è l'insieme ottenuto dall'insieme i eliminando l'elemento e; se e non appartiene a i allora RESULT coincide con i

**membro(Ins i, T e) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se l'elemento e appartiene all'insieme i, false altrimenti

**FineTipoAstratto**

### 10.6.1. Realizzazione del tipo astratto *Insieme*

Considereremo la realizzazione del tipo astratto in due varianti, corrispondenti ad altrettante rappresentazioni degli elementi di dominio: mediante array (rappresentazione indicizzata) e SCL (rappresentazione collegata). Per il momento, rappresentiamo il dominio di interesse tramite il tipo Insieme, senza preoccuparci della sua definizione concreta, che verrà fornita in seguito quando saranno presentate le due realizzazioni.

Come già discusso, accanto alla rappresentazione degli elementi del dominio, occorre scegliere tra gli schemi realizzativi:

- funzionale con condivisione di memoria
- side-effect senza condivisione di memoria

In questo corso non ci occupiamo di affrontare tale scelta ma di come realizzare il tipo astratto, una volta che la scelta sia stata effettuata. Per i due schemi realizzativi sopra riportati, abbiamo le seguenti intestazioni delle funzioni del tipo astratto:

- funzionale con condivisione di memoria

```
Insieme insiemeVuoto();
Insieme inserisci(Insieme ins, T e);
Insieme elimina(Insieme ins, T e);
bool estVuoto(Insieme ins);
bool membro(Insieme ins, T e);
```

- side-effect senza condivisione di memoria

```
Insieme* insiemeVuoto();
void inserisci(Insieme *ins, T e);
void elimina(Insieme *ins, T e);
bool estVuoto(Insieme *ins);
bool membro(Insieme *ins, T e);
```

Vedremo ora due esempi di realizzazione del tipo *Insieme* secondo uno schema con side-effect senza condivisione di memoria (un esempio di realizzazione funzionale con condivisione di memoria sarà proposto successivamente). Le due varianti si distinguono per il modo in cui i dati sono rappresentati: nel primo si usa una rappresentazione mediante SCL mentre nel secondo l'insieme è rappresentato mediante array. Come anticipato, alle diverse rappresentazioni corrispondono diverse definizioni del tipo Insieme (e, conseguentemente, diverse implementazioni delle funzioni).

**Rappresentazione mediante SCL** Abbiamo la seguente definizione del tipo Insieme:

```
typedef int T; // Cambia a seconda del tipo trattato

struct NodoSCL {
    T info;
    struct NodoSCL* next;
};

typedef struct NodoSCL TipoNodo;
typedef TipoNodo* Insieme;
```

Si osservi che in questo esempio si assume che il tipo degli elementi T sia il tipo intero (prima riga). La realizzazione risultante può essere

facilmente adattata ad un qualsiasi altro tipo cambiando esclusivamente la definizione di T mediante typedef.

Con le scelte effettuate sopra, le funzioni che implementano il tipo sono le seguenti:

```
Insieme* insiemeVuoto() {
  Insieme* r = (Insieme*) malloc(sizeof(Insieme));
  *r = NULL;
  return r;
}
```

```
bool estVuoto(Insieme* ins) {
    return *ins == NULL;
}
```

```
void inserisci(Insieme *ins, T e) {
    if (!membro(ins,e)) {
        TipoNodo* n = (TipoNodo*) malloc(sizeof(
    TipoNodo));
        n->info = e;
        n->next = *ins;
        *ins = n;
    }
}
```

```
void elimina(Insieme *ins, T e) {
    if (*ins == NULL){
      return;
    }
    NodoSCL* p = *ins;
    if (p->info == e){
      *ins = p->next;
      free(p);
      return;
    }
    elimina(&((*ins) -> next),e);
}
```

```
bool  membro(Insieme* ins, T e) {
    NodoSCL* p = *ins;
    while (p!=NULL) {
        if (p -> info == e){
          return true;
        }
        p = p->next;
    }
    return false;
}
```

Si osservi che, mentre le funzioni fin qui presentate permettono di manipolare insiemi generici, nessuna di esse permette la visita dell'insieme. In altre parole, il tipo non mette a disposizioni funzioni primitive che permettano di visitare tutti gli elementi contenuti nell'insieme. Ciò è chiaramente possibile conoscendo la rappresentazione del tipo. Ad esempio, sapendo che l'insieme è rappresentato come SCL e conoscendo la struttura dei suoi nodi, la visita dell'insieme si riduce essenzialmente alla visita di una SCL. Questa modalità di accesso ha tuttavia l'inconveniente di dipendere dalla rappresentazione del tipo e quindi di imporre la riscrittura delle funzioni di visita da parte delle funzioni (o programmi) cliente. Sarebbe, invece, preferibile una soluzione che garantisse piena compatibiltà del codice cliente con qualsiasi implementazione del tipo. Più avanti verrà presentato un approccio di questo tipo.

**Rappresentazione mediante array** Mostriamo ora una semplice implementazione del tipo *Insieme* rappresentato mediante array di dimensione fissa; assumiamo cioè che gli insiemi trattati non contengano più di numero prefissato di elementi (ad es. 100). La generalizzazione ad insiemi di qualsiasi cardinalità non pone particolari difficoltà.

Definiamo il tipo Insieme come segue:

```
typedef int T;

typedef struct {
    int size; // dimensione dell'array
    int nelem; // num. elementi validi
    T* data; // array
} Insieme;
```

Una possibile realizzazione delle funzioni è la seguente:

```
Insieme* insiemeVuoto() {
    Insieme* ins = (Insieme *)malloc(sizeof(Insieme));
    ins->size = 100;
    ins->nelem = 0;
    ins->data = (T*)malloc(ins->size*sizeof(T));
    return ins;
}
```

```
bool estVuoto(Insieme *ins) {
    return ins->nelem == 0;
}
```

```
void inserisci(Insieme *ins, T e) {
    if (!membro(ins,e)) {
        if (ins->nelem < ins->size) {
            ins->data[ins->nelem] = e;
            ins->nelem ++;
        }
        else {
            printf("ERRORE: array pieno\n");
        }
    }
}
```

```
void elimina(Insieme *ins, T e) {
    int i=0;
    while (i<ins->nelem && ins->data[i]!=e)
        i++;
    if (i==ins->nelem) {
        // nessuna operazione, elemento non presente
    }
    else {
        // spostare altri elementi
        for (int j=i; j<ins->nelem-1; j++)
            ins->data[j] = ins->data[j+1];
        ins->nelem --;
    }
}
```

```
bool membro(Insieme *ins, T e) {
    bool r = false;
    for (int i=0; i<ins->nelem && !r; i++) {
        r = ins->data[i]==e;
    }
    return r;
}
```

Considerazioni analoghe al caso di rappresentazione dell'insieme con SCL possono essere fatte per quanto riguarda l'accesso agli elementi dell'insieme. Proponiamo qui di seguito una possibile soluzione al problema.

## 10.7. Il tipo astratto *Iteratore*

Con il termine *Iteratore* si indica un tipo astratto i cui elementi permettono di accedere in modo sequenziale a tutti gli elementi di una collezione. Non sempre l'uso di iteratori è necessario quando vengono manipolate collezioni. Questo è possibile, ad esempio, combinando opportunamente le funzioni primitive dei tipi Lista, Pila e Coda.

Il tipo Iteratore viene definito in modo indipendente dalla collezione e ovviamente dalla rispettiva implementazione; esso è quindi compatibile con diversi tipi di dato lineari (ad esempio, liste, insiemi, pile, code), e in alcuni casi sono anche con dati non-lineari (ad esempio, alberi).

L'ordine con cui vengono visitati gli elementi della collezione dipende dal tipo di collezione. Ad esempio, in una lista l'ordine sarà dal primo elemento all'ultimo, in un insieme l'ordine sarà arbitrario.

Il tipo astratto *Iteratore* è definito come segue:

**TipoAstratto** **Iteratore(C, T)**

**Domini**

**It** : dominio di interesse

**C** : dominio delle collezioni su cui applichiamo l'iteratore

**T** : dominio degli elementi della collezione

**Funzioni**

**crea(C c)** $\mapsto$ **It**
pre: nessuna
post: RESULT è un iteratore per la collezione c inizializzato per puntare ad un primo elemento della collezione

**Funzioni**

**hasNext(It i) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se l'iteratore i punta ad un elemento valido della collezione, false altrimenti

**next(It i) $\mapsto$ T**
pre: i punta ad un elemento valido
post: RESULT è il valore dell'elemento puntato dall'iteratore i, l'iteratore viene incrementato per puntare ad un prossimo elemento della collezione non ancora visitato

**FineTipoAstratto**

### 10.7.1. Realizzazione dell'Iteratore (SCL)

Sebbene da un punto di vista tecnico un iteratore possa essere realizzato sia con uno schema funzionale che con side-effect, il secondo risulta più appropriato in quanto gli iteratori astraggono entità mutabili, il cui stato cambia dopo la visita di un elemento.

La rappresentazione di un iteratore deve ovviamente includere le informazioni necessarie ad eseguire la visita. Osserviamo che, a questo scopo, l'informazione minima necessaria risulta essere un riferimento al prossimo elemento da visitare. A seconda della struttura da visitare, possono essere necessarie informazioni aggiuntive.

Vediamo nel seguito l'implementazione di un iteratore secondo schema realizzativo con side-effect che permetta di accedere agli elementi di una collezione rappresentata mediante SCL o array. Realizzeremo l'iteratore per gli insiemi nelle due varianti implementative viste sopra.

**Iteratore per il tipo Insieme rappresentato mediante SCL** Facendo riferimento all'implementazione del tipo *Insieme* con rappresentazione mediante SCL, implementiamo l'iteratore nel tipo IteratoreInsieme, definito come segue:

```
typedef struct{
  NodoSCL* ptr;
} IteratoreInsieme;
```

Intuitivamente, l'iteratore è un riferimento al primo elemento del sottoinsieme non ancora visitato.

Con questa scelta, una possibile implementazione delle funzioni del tipo è la seguente:

```
IteratoreInsieme* creaIteratoreInsieme(Insieme* ins) {
  IteratoreInsieme* r = (IteratoreInsieme*) malloc(
    sizeof(IteratoreInsieme));
  r->ptr = *ins;
  return r;
}
```

```
bool hasNext(IteratoreInsieme* it) {
    return it->ptr!=NULL;
}
```

```
T next(IteratoreInsieme *it) {
    T r = TERRORVALUE;
    if (it->ptr!=NULL) {
        r = it->ptr->info;
        it->ptr = it->ptr->next;
    }
    else
        printf("ERRORE Iteratore non valido.\n");
    return r;
}
```

**Iteratore per il tipo Insieme rappresentato mediante array** In questo caso è necessario mantenere informazioni aggiuntive rispetto al riferimento all'elemento da restituire, ad esempio il numero di elementi significativi memorizzati nell'array. Scegliamo di mantenere queste informazioni tramite riferimento alla struttura da visitare:

```
typedef struct {
    Insieme* ins; // riferimento all'insieme da
    visitare
    int ptr; // indice del prossimo elemento
} IteratoreInsieme;
```

Anche in questo caso, l'iteratore è un riferimento al primo elemento del sottoinsieme non ancora visitato. Con questa scelta, una possibile implementazione delle funzioni del tipo è la seguente:

```
IteratoreInsieme* creaIteratoreInsieme(Insieme *ins) {
    IteratoreInsieme *it = (IteratoreInsieme *)malloc(
    sizeof(IteratoreInsieme));
    it->ins = ins;
    it->ptr = 0;
    return it;
}
```

```
int hasNext(IteratoreInsieme *it) {
    return it->ptr < it->ins->nelem;
}
```

```
T next(IteratoreInsieme *it) {
    T r = TERRORVALUE;
    if (hasNext(it)) {
        r = it->ins->data[it->ptr];
        it->ptr ++;
    }
    else
        printf("ERRORE Iteratore non valido.\n");
    return r;
}
```

### 10.7.2. Uso dell'Iteratore

Vediamo ora un semplice esempio di uso dell'iteratore. Realizziamo una funzione che permetta di stampare tutti gli elementi contenuti in un insieme (di interi), indipendentemente da come questo sia rappresentato.

```
void stampa(Insieme* ins) {
    IteratoreInsieme* it = creaIteratoreInsieme(ins);
    while (hasNext(it)) {
        T e = next(it);
        printf("%d ", e);
    }
    printf("\n");
}
```

Ad ogni iterazione del ciclo while, la funzione controlla se è presente un elemento da visitare (hasNext): in caso affermativo, tramite la funzione next, l'elemento viene estratto, quindi stampato, ed il riferimento al

prossimo elemento viene posizionato sull'elemento seguente; in caso negativo, il ciclo termina.

È facile verificare che il codice sopra mostrato funziona indipendentemente dalla modalità di rappresentazione scelta per gli elementi del tipo.

## 10.8. Tipo astratto Lista

Una *lista* è una collezione ordinata di dati omogenei. La lista, ed in generale qualunque collezione, rappresenta un classico esempio di *tipo parametrico*, ovvero un tipo di dato che coinvolge uno o altri tipi di dato, ma il cui comportamento è indipendente da esso.

Se TipoInfo è il tipo degli elementi, il dominio del tipo di dato lista è costituito da tutte le sequenze di elementi di tipo TipoInfo. Per denotare una lista si usa abitualmente la notazione parentetica:

- () denota la lista vuota
- (e1 e2 ...) denota una lista il cui primo elemento è e1, il secondo e2, ecc.

Si noti che ( e1, e2 ) ed ( e2, e1 ) denotano liste distinte, in quanto anche l'ordine degli elementi contenuti caratterizza una lista.

La definizione del tipo astratto Lista è la seguente: [1]

**TipoAstratto Lista(T)**

**Domini**

**Lista** : dominio di interesse del tipo

**T** : dominio degli elementi che formano le liste

**Funzioni**

**listaVuota() $\mapsto$ Lista**
pre: nessuna
post: RESULT è la lista vuota

**estVuota(Lista l) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se la lista l è vuota, false altrimenti

[1] I nomi car e cdr in riferimento alle liste erano usati nelle prime implementazioni del LISP (List Processor), un linguaggio funzionale interamente basato su liste, ideato da John McCarty nel 1958. Essi fuorono suggeriti dalle abbreviazioni per "contents of the address part of register number" (car) e "contents of the decrement part of register number" (cdr), utilizzate in riferimento alla macchina IBM 704, su cui i precursori del Lisp furono implementati.

**Funzioni**

**cons(T e, Lista l) $\mapsto$ Lista**
pre: nessuna
post: RESULT è la lista ottenuta da l inserendo e come primo elemento

**car(Lista l) $\mapsto$ T**
pre: l non è la lista vuota
post: RESULT è il primo elemento di l

**cdr(Lista l) $\mapsto$ Lista**
pre: l non è la lista vuota
post: RESULT è la lista ottenuta da l eliminando il primo elemento

**FineTipoAstratto**

### 10.8.1. Realizzazione del tipo Lista

Procederemo in maniera simile a quanto fatto per il tipo Insieme, ovvero mostrando due varianti del tipo Lista. Sceglieremo di implementare le funzioni seguendo uno schema realizzativo funzionale con condivisione di memoria, scegliendo prima una rappresentazione mediante SCL e successivamente mediante array. Rappresentiamo il dominio di interesse, ovvero l'insieme delle liste, tramite il tipo TipoLista, la cui definizione dipende dalla rappresentazione adottata.

Prima di descrivere la realizzazione, riportiamo le intestazioni delle funzioni del tipo astratto nei due possibili schemi realizzativi :

- funzionale con condivisione di memoria

```
TipoLista listaVuota();
int estVuota(TipoLista l);
TipoLista cons(T e, TipoLista l);
T car(TipoLista l);
TipoLista cdr(TipoLista l);
```

- side-effect senza condivisione di memoria

```
TipoLista* listaVuota();
bool estVuota(TipoLista* l);
void cons(T e, TipoLista* l);
T car(TipoLista* l);
void cdr(TipoLista* l);
```

**Rappresentazione mediante SCL** Se decidiamo di rappresentare le liste mediante SCL definiamo TipoLista come un riferimento al primo elemento della SCL.

```
typedef int T;

struct NodoSCL {
  T info;
  struct NodoSCL *next;
};

typedef struct NodoSCL TipoNodo;

typedef TipoNodo* TipoLista;
```

Con questa scelta, le funzioni che implementano il tipo (secondo uno schema realizzativo funzionale) sono le seguenti:

```
TipoLista listaVuota() {return NULL;}
```

```
int estVuota(TipoLista l) {return (l==NULL);}
```

```
TipoLista cons(T e, TipoLista l){
  TipoLista n = (TipoLista) malloc(sizeof(TipoNodo));
  n->info = e;
  n->next = l;
  return n;
}
```

```
T car(TipoLista l){
  if (l==NULL){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  return l->info;
}
```

```
TipoLista cdr(TipoLista l){
  if (l==NULL){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  return l->next;
}
```

**Rappresentazione mediante array** Se invece decidiamo di rappresentare le liste mediante array, definiamo TipoLista come un record contenente:

- un campo puntatore data usato come riferimento all'array che modella la lista;
- un campo intero n contenente la dimensione dell'array.

La definizione di TipoLista è pertanto la seguente:

```
typedef struct {
  T* data;
  int n;
} TipoLista;
```

Una possibile realizzazione delle funzioni, coerente con le scelte effettuate, è la seguente:

```
TipoLista listaVuota() {
  TipoLista l;
  l.n=0;
  return l;
}
```

```
int estVuota(TipoLista l) {
  return (l.n==0);
}
```

```
TipoLista cons(T e, TipoLista l){
  TipoLista r;
  r.n=l.n+1;
  r.data = (T*) malloc((r.n)*sizeof(T));
  //Copia l.data in r.data a partire dalla seconda
    componente
  for (int i = 1; i < r.n; i++){
    r.data[i] = l.data[i-1];
  }
  // Inserisce e come primo elemento di r.data
  r.data[0]=e;
  return r;
}
```

```
T car(TipoLista l){
  if (l.n==0){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  return l.data[0];
}
```

```
TipoLista cdr(TipoLista l){
  if (l.n==0){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  TipoLista r;
  r.n = l.n-1;
  r.data = &(l.data[1]); // condivisione di memoria
  return r;
}
```

La differenza tra le due implementazioni risiede nella specifica di TipoLista e nel corpo delle funzioni. Infatti, funzioni che, nelle due implementazioni, realizzano la stessa funzione astratta, presentano *esattamente* stessa segnatura (in quanto questa dipende, in ultima analisi, solo dallo schema realizzativo scelto). Come conseguenza di ciò, qualunque funzione che faccia uso di queste funzioni è perfettamente compatibile con entrambe le implementazioni. In altre parole, le funzioni *cliente* del tipo astratto non hanno bisogno di conoscere i dettagli implementativi del tipo stesso. Si noti come questo effetto (desiderabile, in quanto favorente il riuso di codice) sia conseguenza delle scelte effettuate in

fase di realizzazione, in particolare, dell'aver definito il tipo generico TipoLista, definito di volta in volta, a seconda della rappresentazione scelta. Mostriamo di seguito un esempio di uso di tali implementazioni.

### 10.8.2. Esempi di uso della Lista

Definiamo, a partire dall'implementazione del tipo astratto, alcune funzioni sulle liste:

Calcolo della lunghezza della lista (funzione length): data una lista, ne calcola la lunghezza;

```
int length(TipoLista l){
  if (estVuota(l)) return 0;
  return 1 + length(cdr(l));
}
```

Inserimento di un elemento in coda alla lista (funzione append): data una lista ed un elemento da aggiungere, restituisce una nuova lista ottenuto dalla prima aggiungendovi il nuovo elemento in coda;

```
TipoLista append(TipoLista l, TipoInfo e){
  if(estVuota(l)){
    return cons(e,l);
  }
  return cons(car(l),append(cdr(l),e));
}
```

Concatenazione di due liste (funzione concat): date due liste l1 ed l2, ne restituisce una nuova, ottenuta concatenando ad l1 gli elementi di l2, mantenendone l'ordine;

```
TipoLista concat(TipoLista l1, TipoLista l2){
  if (estVuota(l2)){
    return (l1);
  }
  return concat(append(l1,car(l2)),cdr(l2));
}
```

Restituzione dell'elemento in posizione i di una lista (funzione get): data una lista ed un intero i, restituisce l'elemento della lista in posizione i;

```
TipoInfo get(TipoLista l, int i){
  if (i < 0 || estVuota(l)){
    printf("ERRORE: lista vuota o indice fuori dai
    limiti!\n");
    exit(1);
  }
  if (i==0) return car(l);
  return get(cdr(l),i-1);
}
```

Inserimento di un nuovo elemento in posizione i (funzione ins): data una lista, un intero i, e un elemento da inserire, restituisce una nuova lista ottenuta dalla primo aggiungendovi il nuovo elemento in posizione i.

```
TipoLista ins(TipoLista l, int i, TipoInfo e){
  if (i < 0 || (i>0 && estVuota(l))) {
    printf("ERRORE: indice fuori dai limiti!\n");
    exit(1);
  }
  if (i==0) return cons(e,l);
  return cons(car(l),ins(cdr(l), i-1, e));
}
```

Come anticipato, tutte le funzioni cliente sopra definite sono indipendenti dalla rappresentazione scelta. Ovviamente lo stesso non può dirsi per le funzioni del tipo, la cui implementazione è fortemente legata alla sua rappresentazione. Si osservi anche che, come detto, le funzioni qui definite sono compatibili esclusivamente con uno schema realizzativo funzionale.

## 10.9. Tipo astratto Coda

Una *coda* (o *queue*) è una sequenza di elementi omogenei gestiti con politica *first-in-first-out (FIFO)*, ovvero in cui è possibile inserire ed estrarre elementi, garantendo che l'elemento estratto sia quello presente nella coda da più tempo.

La gestione di dati tramite coda permette di elaborare i dati nell'ordine di arrivo.

La specifica del tipo Coda è la seguente.

**TipoAstratto Coda(T)**

**Domini**

**Coda** : dominio di interesse del tipo

**T** : dominio degli elementi delle code

**Funzioni**

**codaVuota() $\mapsto$ Coda**
pre: nessuna
post: RESULT è la coda vuota

**estVuota(Coda c) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se c è il valore corrispondente alla coda vuota, false altrimenti corrispondente alla coda vuota, false altrimenti

**Funzioni**

**inCoda(Coda c, T e) $\mapsto$ Coda**
pre: nessuna
post: RESULT è la coda ottenuta dalla coda c inserendo l'elemento e, che ne diventa l'ultimo elemento della coda

**outCoda(Coda c) $\mapsto$ Coda**
pre: c non è la coda vuota
post: RESULT è la coda ottenuta dalla coda c eliminando l'elemento in testa, cioè che tra quelli presenti era stato inserito per primo

**primo(Coda c) $\mapsto$ T**
pre: c non è la coda vuota
post: RESULT è l'elemento in testa alla coda c, cioè l'elemento che tra quelli presenti in c era stato inserito per primo

**FineTipoAstratto**

### 10.9.1. Realizzazione del tipo Coda

Come nel caso delle liste, entrambi gli schemi relizzativi sono plausibili. Vediamo qui due esempi di realizzazione secondo lo schema con side-effect senza condivisione di memoria. Anche in questi casi considereremo le due rappresentazioni dei dati mediante SCL o array.

Rappresentiamo il dominio di interesse tramite il tipo Coda. A tale proposito, valgono esattamente le stesse considerazioni fatte nel caso del tipo astratto Lista.

**Rappresentazione mediante SCL** Se decidiamo di rappresentare le code mediante SCL, definiamo Coda come un riferimento al primo elemento della SCL.

```
typedef int T;

struct NodoSCL {
  T info;
  struct NodoSCL *next;
};

typedef struct NodoSCL TipoNodo;
typedef TipoNodo* Coda;
```

Con questa scelta, le funzioni che implementano il tipo (secondo lo schema realizzativo scelto) sono le seguenti:

```
Coda* codaVuota() {
  return (Coda*) malloc(sizeof(Coda));
}
```

```
bool estVuota(Coda* c) {return (*c==NULL);}
```

```
void inCoda(Coda* c, T e){
  if (*c == NULL){
    *c = (Coda) malloc(sizeof(TipoNodo));
    (*c) -> info = e; (*c) -> next = NULL;
  }
  else inCoda(&((*c) -> next), e);
}
```

```
void outCoda(Coda* c){
  if (c == NULL || *c == NULL){
    printf("ERRORE: input NULL o coda vuota");
  }
  Coda tmp = *c;
  *c = (*c) -> next;
  free(tmp);
}
```

```
T primo(Coda* c){
  if (*c == NULL){
    printf("ERRORE: coda vuota");
    exit(1);
  }
  return (*c)->info;
}
```

**Rappresentazione mediante array** La rappresentazione mediante array, nel caso con side-effect, soffre di problemi di efficienza, in quanto ogni inserimento o estrazione implica la riallocazione dell'intero array. Per ovviare a ciò adottiamo la tecnica del *raddoppiamento/dimezzamento*. Nel dettaglio, rappresentiamo una coda con un record contenente:

- un campo data che fa riferimento ad un array di elementi di tipo T
- un campo intero size dove memorizziamo la dimensione dell'array data
- un campo intero nelem per memorizzare il numero di elementi della coda.

La coda è rappresentata dai primi nelem elementi di data, dove l'elemento in posizione 0 rappresenta la testa della coda, ovvero il prossimo elemento che sarà estratto, e l'elemento in posizione nelem-1 l'ultimo elemento.

Gestiamo l'array data come segue:

- inizialmente l'array è vuoto, con size e nelem nulli
- quando viene effettuato un inserimento, se data è pieno, allora data viene ridimensionato con una dimensione pari al doppio della dimensione della coda risultante
- quando viene effettuata un'estrazione, se la dimensione della coda risultante è minore di size/2 la dimensione dell'array viene dimezzata.

In questo modo si riduce la frequenza delle riallocazioni (si noti che ciò implica, in generale, spreco di memoria).

Con queste scelte, abbiamo la seguente definizione del tipo Coda:

```
typedef struct {
  T* data; // elemento data[0]: testa della coda
  int size; // dimesione array
  int nelem; // dimensione coda
} Coda;
```

Conseguentemente, una possibile realizzazione delle funzioni è la seguente:

```
Coda* codaVuota() {
  Coda* r = (Coda*) malloc (sizeof(Coda));
  r->data = NULL;
  r->size = 0;
  r->nelem = 0;
  return r;
}
```

```
bool estVuota(Coda* c) {return (c->nelem==0);}
```

```
void inCoda(Coda* c, T e){
  c->nelem++;
  if (c->nelem > c->size){
    // Raddoppiamento dimensione array
    c -> size = 2 * c-> nelem;
    c -> data = (T*) realloc(c->data,c->size*sizeof(T)
    );
  }
  c->data[c->nelem -1] = e;
}
```

```
void outCoda(Coda* c){
  if (c == NULL || c->nelem == 0){
    printf("ERRORE: input NULL o coda vuota");
    exit(1);
  }
  c->nelem--;
  if (c->nelem < c->size/2){
    // Dimezzamento array
    c->size /= 2;
    c->data = (T*) realloc(c->data,c->size*sizeof(T));
  }
  // Copia tutti gli elementi della coda nella
    componente precedente
  for (int i = 1; i <= c->nelem; i++){
    c->data[i-1]=c->data[i];
  }
}
```

```
T primo(Coda* c){
  if (c->nelem == 0){
    printf("ERRORE: coda vuota");
    exit(1);
  }
  return c->data[0];
}
```

È facile vedere anche in questo caso che eventuali funzioni cliente sono perfettamente compatibili con entrambe le implementazioini, indipendentemente dalla modalità di rappresentazione scelta.

## 10.10. Tipo astratto Pila

Una *pila* (o *stack*) è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella pila deve essere quello che è stato inserito per ultimo.

Questa politica di accesso viene detta *LIFO* ("*L*ast *I*n, *F*irst *O*ut").

Con una pila si risolvono facilmente i problemi in cui i dati vanno elaborati in ordine inverso dal loro arrivo (dal più recente al più vecchio). Un tipico esempio d'uso della struttura dati pila è quello in cui essa viene utilizzata per la gestione delle invocazioni di funzione, ovvero

la pila dei RDA, dove l'invocazione più recente è la porssima che deve essere processata.

Il tipo di dato astratto Pila è definito come segue.

**TipoAstratto Pila(T)**

**Domini**

**Pila** : dominio di interesse del tipo

**T** : dominio degli elementi delle pile

**Funzioni**

**pilaVuota() $\mapsto$ Pila**
pre: nessuna
post: RESULT è la pila vuota

**estVuota(Pila p) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se p è il valore corrispondente alla pila vuota, false altrimenti

**Funzioni**

**push(Pila p, T e) $\mapsto$ Pila**
pre: nessuna
post: RESULT è la pila ottenuta dalla pila p inserendo l'elemento e, che ne diventa l'elemento affiorante

**pop(Pila p) $\mapsto$ Pila**
pre: p non è la pila vuota
post: RESULT è la pila ottenuta dalla pila p eliminando l'elemento affiorante

**top(Pila p) $\mapsto$ T**
pre: p non è la pila vuota
post: RESULT è l'elemento affiorante della pila p

**FineTipoAstratto**

### 10.10.1. Realizzazione della Pila

Le pile possono essere rappresentate mediante strutture collegate lineari. La scelta dello schema realizzativo, quanto la modalità di rappresentazione dei dati devono essere valutate caso per caso.

Omettiamo la realizzazione in C del tipo Pila, lasciandola, in tutte le sue varianti come utile esercizio. A tale proposito, si faccia riferimento agli esempi sopra proposti per i tipi *Lista* e *Coda*

# 11. Costo dei programmi, algoritmi di ricerca e di ordinamento

## 11.1. Costo dei programmi

Il costo di un programma dipende dalle risorse utilizzate: ci concentriamo su spazio di memoria e tempo di elaborazione. Altre risorse di interesse sono: la quantità di traffico generata su una rete di calcolatori; la quantità di dati che devono essere trasferiti da e su disco, ecc.

Stabilire il costo computazionale di una funzione/programma/algoritmo serve a confrontare diversi funzioni/programmi/algoritmi che risolvono lo stesso problema, in modo da scegliere il più efficiente.

### 11.1.1. Misura del tempo

La misura del tempo può essere effettuata valutando il tempo della CPU necessario all'esecuzione del programma su un insieme di dati di ingresso particolari che fungono da banco di prova (benchmark), oppure tramite l'analisi di costo del programma.

Difficoltà della misura:

- Uno stesso programma potrebbe comportarsi diversamente se eseguito su due diversi computer.
- Se abbiamo due diversi programmi che ordinano una collezione di dati, non è detto che si comportino allo stesso modo sugli stessi dati di ingresso.
- Uno stesso algoritmo scritto in due diversi linguaggi di programmazione, potrebbe esibire comportamenti diversi a causa dei differenti compilatori dei due linguaggi.

- Uno stesso programma che riceve in input 100 nomi (come stringhe) potrebbe impiegare k secondi se ciascuna stringa è formata da un singolo carattere e 200 k secondi se ciascuna stringa è formata da 200 caratteri.

### 11.1.2. Modello di costo

Una misura del costo computazionale soddisfacente deve:

- basarsi su un modello di calcolo astratto, indipendente dalla particolare macchina
- svincolarsi dalla configurazione dei dati in ingresso, ad esempio basandosi sulle configurazioni più sfavorevoli (caso peggiore), così da garantire che le prestazioni nei casi reali saranno al più costose quanto il caso analizzato
- essere una funzione (non un numero!!!) della dimensione dell'input (la configurazione l'abbiamo fattorizzata via considerando il caso peggiore)
- essere asintotica, cioè fornire una idea dell'andamento del costo all'aumentare della dimensione dell'input(si noti che essere troppo precisi non avrebbe senso visto che non consideriamo la macchina effettiva che esegue il calcolo)

Cioè siamo interessati al *costo asintotico nel caso peggiore*.

### 11.1.3. Modello di costo

Vogliamo definire un modello di costo (cioè del tempo di esecuzione di un programma) che sia quindi indipendente dall'implementazione.

Definiamo una funzione di costo come segue.

- istruzione atomica (operazioni e confronti su tipi di dati primitivi) ha costo costante
- istruzione di ingresso o di uscita (tipo primitivo) ha costo costante
- istruzione di assegnazione ha costo costante
- blocco di istruzioni

```
{
    <istruzione_1>
    ....
    <instruzione_k>
}
```

ha costo pari a $c_1 + \ldots + c_k$, dove $c_i$ è il costo dell'istruzione $i$-esima del blocco.

- istruzione condizionale

```
if (<condizione>)
    <parte if>
else
    <parte else>
```

ha costo pari a $c + k$ se la condizione è vera oppure $c + h$ se la condizione è falsa, dove $c$ è il costo del calcolo della condizione, $h$ è il costo della parte if e $h$ il costo della parte else.

- istruzione di ciclo (consideriamo una istruzione for, le altre istruzioni di ciclo, come il while, si possono ridurre a queste)

```
for (int i =0; i<=k; i++)
    <corpo-del-ciclo>
```

ha costo pari a $c_1 + k(c_2 + h) + c_3$, dove $c_1$ e il costo di inizializzazione del ciclo, $c_2$ è il costo del confronto e dell'incremento che vengono eseguiti ogni volta che il programma entra nel ciclo, $h$ è il costo totale di tutte le istruzioni nel corpo del ciclo, $k$ è il numero di volte che viene eseguito il ciclo, $c_3$ è il costo dell'ultimo confronto prima dell'uscita dal ciclo. Il costo di un ciclo può essere approssimato asintoticamente a $c + kd$, in cui $c$ riassume i costi delle operazioni svolte una sola volta, e $d$ riassume i costi delle operazioni svolte in ogni esecuzione del ciclo.

- invocazione di una funzione ha costo pari al costo di esecuzione di tutte le istruzioni del corpo della funzione.

Denotiamo con $T(n)$ il costo di un programma misurato come funzione della dimensione $n$ dei dati di ingresso.

### 11.1.4. Analisi per casi

Nella valutazione del costo computazionale dei programmi si distinguono tre casi.

- Caso peggiore: il caso peggiore corrisponde al tempo massimo che richiede un problema di dimensione $n$.
- Caso migliore: il caso migliore corrisponde al tempo minimo che richiede un problema di dimensione $n$.
- Caso medio: il caso medio corrisponde al tempo di soluzione medio per un problema di dimensione $n$ e richiede di valutare il programma con diverse configurazioni di input.

Se non si specifica diversamente, si considera il caso peggiore per determinare il costo.

### 11.1.5. Studio del comportamento asintotico

Il comportamento asintotico del costo di un programma si basa sull'idea di trascurare costanti moltiplicative e termini di ordine inferiore e fornisce quindi un nozione del costo all'aumentare della dimensione dell'input.

Il comportamento asintotico inoltre rende l'analisi del costo insensibile rispetto alle approssimazioni introdotte nel modello di costo adottato e consente di semplificare i calcoli.

Esempi: il costo $an + b$ ha comportamento asintotico lineare, il costo $an^2 + bn + c$ ha comportamento asintotico quadratico, il costo $alog_b n + c$ ha comportamento asintotico logaritmico, ecc.

### 11.1.6. Notazione O-grande

**Definizione.** Una funzione $f(n)$ è $O(g(n))$ (che si legge O-grande di $g$) se e solo se esistono due costanti positive $c$ e $n_0$, tali che

$$|f(n)| \leq cg(n)$$

per tutti i valori $n \geq n_0$.

La definizione di notazione O-grande ci dice che da un certo valore $n_0$ in poi, la dimensione di $f(n)$ non supera quella di $g(n)$ dato un certo fattore di scala $c$.

Considerando O-grande siamo interessati all'andamento asintotico della funzione di costo del programma. Ad esempio, se consideriamo un programma che ha tempo di esecuzione $T(n) = (n+1)^2$, allora $T(n)$ è $O(n^2)$. Infatti se poniamo $c = 2$ e $n_0 = 3$ abbiamo che $(n+1)^2 \geq 2n^2$, per ogni $n \geq 3$.

### 11.1.7. Costo di un programma/algoritmo

**Definizione.** Un algoritmo/programma A ha costo $O(g(n))$ se la quantità di tempo (spazio) sufficiente per eseguire A su un input di dimensione $n$ nel caso peggiore è $O(g(n))$.

### 11.1.8. Funzioni di costo

La notazione O-grande permette di definire le seguenti funzioni di costo.

$O(1)$ funzione di costo costante (che non dipende dai dati in ingresso).

$O(logn)$ funzione di costo logaritmica

$O(n)$ funzione di costo lineare

$O(nlogn)$ funzione di costo $nlogn$ (quasi-lineare)

$O(n^2)$ funzione di costo quadratica

$O(n^k)$ funzione di costo polinomiale quando è limitata da un polinomio del tipo $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, dove $k$ è una costante

$O(k^n)$ funzione di costo esponenziale quando è limitata da una funzione esponenziale $k^n$, con $k$ costante maggiore di 1

### 11.1.9. Valutazione semplificata: operazioni dominanti

Sia A un programma con costo $O(f_A(n))$, una operazione si dice *dominante* se, nel caso peggiore, viene ripetuta un numero di volte $g(n)$ tale che $f_A(n) = O(g(n))$.

Se un programma A ha una operazione dominante che viene eseguita un numero di volte $O(g(n))$, allora il costo del programma è $O(g(n))$.

Tipicamente per individuare le operazioni dominanti basta esaminare le istruzioni e i confronti eseguiti nei cicli più interni dei programmi, oppure eseguiti nell'ambito delle attivazioni ricorsive.

## 11.2. Algoritmi di ricerca

Il problema della ricerca di un elemento all'interno di una collezione si definisce nel seguente modo: data una collezione di valori $C$ e un certo valore $k$, restituire vero se $k$ fa parte della collezione $C$, falso altrimenti.

### 11.2.1. Ricerca sequenziale

La ricerca sequenziale è un algoritmo che confronta tutti gli elementi della collezione con l'elemento da cercare.

```
// verifica la presenza del valore k nell'array a
bool ricercaSequenziale(int *a, int n, int k) {
   bool r = false;
   for (int i = 0; i < n && !r; i++)
      r = r || (a[i]==k);
   return r;
}
```

Nota: La condizione !r nel controllo del ciclo for non cambia il caso peggiore e quindi non ha effetto sull'analisi del costo nel caso peggiore.

#### 11.2.1.1. Costo della ricerca sequenziale

Il caso peggiore per la funzione ricercaSequenziale è il caso in cui l'elemento cercato k non occorre nell'array a.

Costo per un array di dimensione $n$ (dimensione dell'input):

inizializzazione i = 0: 1
confronti i < a: $n$
confronti a[i] == k: $n$
istruzioni i++: $n$
istruzione return true: 1
totale: $3n + 3$, cioè $O(n)$

La ricerca sequenziale ha un costo lineare.

*Esempio:* Sia l'array a = [3, 5, 7 ,11, 17 ,19, 23, 31] e l'elemento k cercato 6, il costo della ricerca sequenziale è proporzionale a 8.

### 11.2.2. Ricerca binaria

Se la collezione in cui cercare l'elemento è ordinata, si può implementare un algoritmo più efficiente, cioè con costo minore rispetto a quello della ricerca sequenziale.

Sia $a$ un vettore di $n$ elementi $a_0, \ldots, a_{n-1}$ ordinati in modo crescente e $k$ l'intero da cercare in $a$.

Sia left l'indice del primo elemento di $a$, right l'indice dell'ultimo elemento di $a$, med l'indice dell'elemento centrale pari a (left + right)/2.

```
Se non ci sono valori tra left e right , allora
    la ricerca di k non ha avuto successo: ritorna false
Altrimenti
    Se a[med] = k allora
        abbiamo trovato l'elemento e ritorna true
    Se a[med] < k allora
        cerchiamo k nella metà destra del vettore
        ponendo left = med
    se a[med] > k allora
        cerchiamo k nella metà sinistra del vettore
        ponendo right = med
```

Il codice C della ricerca binaria è il seguente.

```
bool ricercaBinaria(int *a, int n, int k) {
   int left = 0, right = n-1;
   while (left<=right) {
      int med = (left+right)/2;
      if (a[med]==k)
         return true;
      else if (a[med]<k)
         left = med+1;
      else // a[med]>k
         right = med-1;
   }
   return false;
}
```

#### 11.2.2.1. Costo della ricerca binaria

Per valutare il costo della ricerca binaria, consideriamo l'istruzione dominante, che è qualsiasi operazione all'interno del ciclo while: ad esempio, la condizione left <= right oppure istruzione med = (left+right)/2, ecc.

Il caso peggiore si presenta di nuovo quando l'elemento k non è presente nell'array, quindi la funzione termina dopo aver eseguito tutte le iterazioni del ciclo fino al caso in cui la codizione left <= right diventa

falsa. Ad ogni iterazione vengono scartati metà degli elementi dell'array. Infatti nella prima iterazione, gli estremi left e right racchiudono $n$ elementi, nella seconda iterazione sono $n/2$ elementi, nell'iterazione $i$-esima avremo $n/2^i$ elementi.

Il procedimento continua fino a quando $n/2^i \geq 1$, cioè $2^i \leq n$, ovvero $i \leq log_2(n)$. Il numero di iterazioni è al massimo pari ad $log(n)$ e poiché il costo di ciascuna iterazione è costante, abbiamo che il costo di ricercaBinaria è $O(log(n))$.

*Esempio:* Sia l'array a = [3, 5, 7 ,11, 17 ,19, 23, 31] e l'elemento k cercato 6, il costo della ricerca binaria è proporzionale a $log_2(8) = 3$.

## 11.3. Algoritmi di ordinamento

Problema: Data una sequenza di elementi in ordine qualsiasi, fornire una sequenza ordinata degli stessi elementi.

Questo è un problema fondamentale, che si presenta in moltissimi contesti e in diverse forme:

- ordinamento degli elementi di un vettore in memoria centrale
- ordinamento di una collezione in memoria centrale
- ordinamento di informazioni memorizzate in un file su memoria di massa (file ad accesso casuale o sequenziale)

Consideriamo inizialmente il problema di ordinare un array a di n elementi di tipo T (ad esempio, int), secondo un criterio di ordinamento definito su T (ad esempio, la relazione $<$ su dati di tipo int).

### 11.3.1. Ordinamento per selezione (Selection Sort)

*Esempio:* Ordinamento di una pila di carte:

```
seleziona la carta più piccola e mettila in prima
posizione seleziona la più piccola tra le rimanenti
e mettila in posizione 2 ... quando rimane una sola
carta, mettila in ultima posizione
```

Posso operare in maniera simile su un array di interi:

```
individua il valore minimo e
      scambia tale valore con quello in posizione 1
```

```
individua il valore minimo a partire da posizione 2 e
      scambialo con quello in posizione 2
...
individua il valore minimo a partire da posizione k e
      scambialo con quello in posizione k
fermati quando si arriva all'ultimo elemento
      che sarà il più grande in ultima posizione
```

L'algoritmo di ordinamento per selezione si può quindi descrivere nel seguente modo:

```
for (i=0; i<n-1; i++)
  trova il valore più piccolo da i a n-1,
        sia esso in posizione jmin
  scambia gli elementi in posizione i e jmin
```

Implementazione

```
void selectionSort(int* a, int n) {
  for (int i=0; i<n-1; i++) {
    // trova il piu' piccolo elemento da i a n-1
    int jmin = i;
    for (int j=i+1; j<n; j++) {
      if (a[j]<a[jmin])
        jmin = j;
    }
    // scambia gli elementi i e jmin
    int aux = a[jmin];
    a[jmin] = a[i];
    a[i] = aux;
  }
}
```

#### 11.3.1.1. Ordinamento per selezione: costo

La funzione selectionSort è implementata usando due cicli annidati. Il ciclo esterno è ripetuto $n-1$ volte, mentre il ciclo interno ha un costo che dipende dalla dimensione della porzione di array considerata. Infine le operazioni di confronto e di scambio hanno costo costante.

Le operazioni di confronto vengono effettuate sempre indipendentemente dallo stato dell'array in input. Anche se l'array è inizialmente già

ordinato, ogni valore viene comunque confrontato con tutti gli altri. Il numeri di scambi invece dipende dall'array in input e può essere pari a 0, se l'array è già ordinato, o pari a $n$ se l'array è ordinato in ordine inverso.

Non esiste un caso peggiore per quanto riguarda il numero di confronti, mentre per il numero di scambi il caso peggiore si presenta quando l'array è ordinato in ordine inverso.

Il numero di totale di operazioni di confronto è dato da:

$$(n-1)+(n-2)+.....+2+1=\frac{n(n-1)}{2}$$

Quindi il costo di selectionSort è $T(c_1(n^2-n)/2+c_2n)$, dove $c_1$ indica il costo delle operazioni del ciclo più interno (confronti e assegnazioni), mentre $c_2$ è il costo delle operazioni di scambio. Il costo asintotico è $O(n^2)$.

### 11.3.2. Ordinamento a bolle (Bubble Sort)

L'ordinamento a bolle si basa sulla seguente proprietà: se tutte le coppie di elementi adiacenti sono ordinate, allora tutta la sequenza è ordinata.

Il procedimento di ordinamento a bolle è il seguente:

```
Operazione base:
    se due elementi adiacenti non sono ordinati
        scambiali
Metodo:
    verifica se la collezione è ordinata e
        scambia tutte le coppie non ordinate
```

Algoritmo:

```
for (i=0; i<n-1; i++)
   scambia le coppie di elementi adiacenti non ordinate
   termina se non e' stato effettuato nessuno scambio
```

L'operazione di scambio di tutte le coppie non ordinate in una unica scansione della sequenza può essere eseguita sia nel verso della sequenza che nel verso opposto. Quando eseguita nel verso della sequenza,

l'elemento più grande si troverà in ultima posizione. Quando eseguita nel verso opposto, l'elemento più piccolo si troverà in prima posizione.

In generale, all'$i$-esimo passo del ciclo esterno gli elementi successivi alla posizione $i$ o quelli fino alla posizione $i$ (a seconda del verso con sui si effettuano le operazioni di scambio) sono stati ordinati.

Implementazione

```
void bubbleSort(int* a, int n) {
  bool ordinato = false;
  for (int i=0; i<n-1 && !ordinato; i++) {
    ordinato = true;
    for (int j=n-1; j>i; j--)
      if (a[j-1] > a[j]) {
        int aux = a[j-1];
        a[j-1] = a[j];
        a[j] = aux;
        ordinato = false;
      }
  }
}
```

Ordinamento a bolle: esempio

```
Collezione  ancora non ordinata:
3 1 4 1 5 9 2 6 5 4

Passo 0: 1 3 1 4 2 5 9 4 6 5
Passo 1: 1 1 3 2 4 4 5 9 5 6
Passo 2: 1 1 2 3 4 4 5 5 9 6
Passo 3: 1 1 2 3 4 4 5 5 6 9
Passo 4: 1 1 2 3 4 4 5 5 6 9

Collezione ordinata:
1 1 2 3 4 4 5 5 6 9
```

#### 11.3.2.1. Ordinamento a bolle: costo

In generale, il ciclo esterno viene eseguito $n-1$ volte, mentre il ciclo interno si riduce di 1 elemento ad ogni passo. Il numero di confronti sarà quindi pari a

$$(n-1)+(n-2)+.....+2+1=\frac{n(n-1)}{2}$$

Il caso peggiore si presenta quando l'array è ordinato in ordine inverso. Se la collezione in input è già ordinata, il costo si riduce a $(n-1)$

in quanto vengono eseguiti solamente $n-1$ confronti e il ciclo esterno viene eseguito 1 sola volta.

L'algoritmo di ordinamento a bolle ha costo $O(n^2)$ nel caso peggiore, e costo $O(n)$ nel caso migliore, ovvero quando la collezione è già ordinata.

### 11.3.3. Ordinamento per fusione (Merge Sort)

L'algoritmo di ordinamento per fusione (Merge Sort), è un algoritmo ricorsivo che si basa sull'operazione di fusione di due sequenze ordinate in una nuova sequenza ordinata contenente tutti gli elementi delle due sequenze. L'operazione di fusione di due sequenze ordinate si può eseguire con costo lineare.

L'algoritmo per fusione è definito dai seguenti passi.

```
Se n<2 allora l'array è già ordinato. Termina
Altrimenti (n>=2)
   1. Ordina la metà sinistra dell'array
   2. Ordina la metà destra dell'array
   3. Fondi le due parti ordinate in un array ordinato
```

Implementazione

```
// funzione ausiliaria ricorsiva
void mergeSort_r(int* v, int inf, int sup) {
  if (inf < sup) {
    int med = (inf+sup)/2;
    mergeSort_r(v,inf,med);
    mergeSort_r(v,med+1,sup);
    merge(v,inf,med,sup);
  }
}

// funzione principale
void mergeSort(int* v, int n) {
  mergeSort_r(v,0,n-1);
}
```

```
// funzione di fusione ( v[inf, ..., med]  e v[med+1,
    ..., sup] )
void merge(int* v, int inf, int med, int sup) {
  int m = sup-inf+1;
  int a[m]; // array ausiliario per la copia
  int i1 = inf;
  int i2 = med+1;
  int i = 0;
  while ((i1 <= med) && (i2 <= sup)) { // entrambi i
    vettori contengono elementi
    if (v[i1] <= v[i2]) {
      a[i] = v[i1];
      i1++;
      i++;
    }
    else {
      a[i] = v[i2];
      i2++;
      i++;
    }
  }
  if (i2 > sup) // e' finitq prima la seconda parte
    del vettore
    for (int k = i1; k <= med; k++) {
       a[i] = v[k];
      i++;
    }
  else // e' finita prima la prima parte del vettore
    for (int k = i2; k <= sup; k++) {
      a[i] = v[k];
      i++;
    }
  // copiamo il vettore ausiliario nel vettore
    originario
  for(int k = 0; k < m; k++) {
    v[inf+k] = a[k];
  }
}
```

#### 11.3.3.1. Ordinamento per fusione: costo

Per valutare il costo computazionale dell'algoritmo Merge Sort si noti che: 1) ad ogni passo il vettore viene diviso in due parti uguali, sulle quali viene richiamata la funzione ricorsiva; 2) la procedura di fusione di due vettori ordinati di dimensioni $n/2$ è pari a $O(n)$,

Il numero di chiamate ricorsive del Merge Sort per un vettore di $n$ elementi è pari a $logn$. Infatti alla $k$-esima chiamata ricorsiva la dimensione del vettore sarà $n/2^k$ e il procedimento continua finché $n/2^k \geq 2$, cioè finché $k \leq logn/2$. Ad ogni chiamata ricorsiva viene invocata la

funzione merge che ha costo $O(n)$.

In definitiva l'algoritmo Merge Sort ha costo pari a $O(nlogn)$.

### 11.3.4. Ordinamento veloce (Quick Sort)

L'ordinamento veloce si basa sul seguente metodo: si prende in esame un elemento x del vettore (detto anche pivot) e, mediante una scansione del vettore stesso, si determina la posizione in cui esso si dovrà trovare nel vettore ordinato (sia essa k); durante la stessa scansione si portano tutti gli elementi più piccoli di x nelle posizioni precedenti k, tutti gli elementi più grandi di x nelle posizioni seguenti k e infine x in posizione k. Quindi si ripete ricorsivamente la procedura di ordinamento sulla porzione di array precedente all'indice k e sulla porzione di array successiva all'indice k.

Esempio:

```
v = 6 12 7 8 5 4 9 3 1
x = 6  ->  k = 4
v' = 5 4 3 1 6 12 7 8 9
ordinamento del sotto-vettore [5 4 3 1]
ordinamento del sotto-vettore [12 7 8 9]
```

La scelta del pivot può essere effettuata secondo diversi criteri. Nell'implementazione che segue viene scelto semplicemente il primo elemento del vettore.

Implementazione

```
// funzione ausiliaria ricorsiva
void quickSort_r(int* v, int inf, int sup) {
  if (inf < sup) {
    int i = inf, j = sup, pivot = v[inf];
    while (i < j) {
      while (v[j] > pivot)
        j--;
      while (i < j && v[i] <= pivot)
        i++;
      if (i < j) {
        // ora v[i] > pivot e v[j] <= pivot e i < j
        // quindi scambio v[i] e v[j]
        int a = v[i];
        v[i] = v[j];
        v[j] = a;
      }
    }
    if (inf != j) {
      // mette il pivot al posto giusto j
      v[inf] = v[j];
      v[j] = pivot;
    }
    if (inf < j-1)
      quickSort_r(v,inf,j-1);
    if (sup > j+1)
      quickSort_r(v,j+1,sup);
  }
}

// funzione principale
void quickSort(int* v, int n) {
  quickSort_r(v,0,n-1);
}
```

#### 11.3.4.1. Ordinamento veloce: costo

Nell'algoritmo di ordinamento Quick Sort il caso migliore si presenta quando ad ogni passo si riesce a partizionare il vettore in due vettori di dimensioni $n/2$. In questo caso il costo è $O(nlogn)$.

Nel caso peggiore, che si verifica quando ad ogni passo viene scelto come pivot il valore minimo o massimo contenuto nell'array[1], si ha un costo di $O(n^2)$, in quanto ad ogni passo si esamina un nuovo vettore di dimensione inferiore di uno rispetto alla dimensione del vettore precedente.

[1] Nel caso in esame, ovvero quando viene preso come pivot il primo elemento dell'array, questa situazione si verifica se l'array è già ordinato (crescente o decrescente).

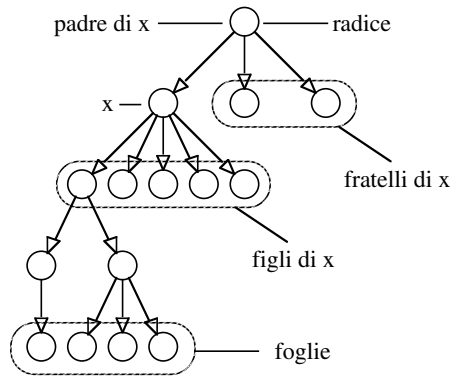
Pur avendo un costo asintotico peggiore del Merge Sort, il Quick Sort si comporta particolarmente bene quando il vettore è mediamente disordinato, ed è per questo motivo che è noto come ordinamento veloce. Infatti rispetto al Merge Sort, il Quick Sort ha il vantaggio di non effettuare l'operazione di fusione e l'allocazione di memoria aggiuntiva per memorizzare il risultato intermedio della fusione.

# 12. Alberi binari

## 12.1. Alberi

In questo capitolo viene presentata la struttura dati *albero*, che rappresenta un esempio particolarmente significativo di struttura non lineare. Infatti, prendendo a modello la definizione induttiva della struttura dati, la definizione delle funzioni che operano su alberi binari risulta immediata, mentre le implementazioni con i costrutti di ciclo richiedono un uso di strutture dati di appoggio.

L'albero è un tipo di dato non lineare utilizzato per memorizzare informazioni in modo gerarchico. Un esempio di albero è il seguente:



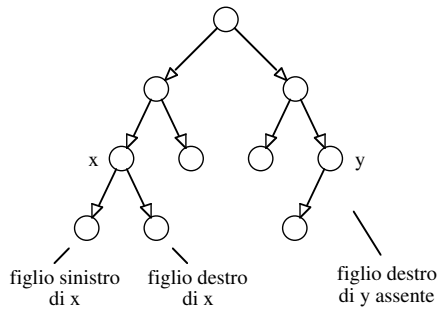

Osserviamo che un albero è costituito da **nodi** (indicati graficamente come cerchi), a cui è tipicamente associato un contenuto informativo, e **archi** (indicati graficamente come frecce), che rappresentano relazioni di discendenza diretta tra nodi. Useremo la seguente terminologia standard:

- **padre** di un nodo $x$: nodo che ha un collegamento in uscita verso il nodo $x$. In un albero ogni nodo ha al più un padre;
- **nodo radice** dell'albero: nodo che ha solo collegamenti in uscita verso altri nodi, cioè non ha nessun padre;
- **fratelli** di un nodo $x$: nodi che hanno lo stesso padre del nodo $x$;
- **figli** di un nodo $x$: nodi verso cui $x$ ha collegamenti in uscita;
- **foglie** dell'albero: nodi che non hanno nessun collegamento in uscita verso altri nodi.

Diremo inoltre che la radice è a **livello** zero nell'albero e che ogni altro nodo si trova ad un livello uguale al livello del padre più uno. Ad esempio, il nodo $x$ è a livello 1 nell'albero della figura precedente. La **profondità** di un albero è pari al massimo livello su cui si trova almeno un nodo dell'albero. Ad esempio, l'albero della figura precedente ha profondità 4.

## 12.2. Alberi binari

In questo capitolo tratteremo il caso particolare di alberi binari, dove ogni nodo ha al più due figli: un **figlio sinistro** ed un **figlio destro**, come mostrato in figura:



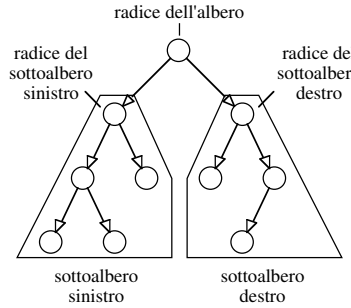

Si noti che il nodo $x$ ha un figlio sinistro ed un figlio destro, mentre il nodo $y$ ha solo un figlio sinistro.

### 12.2.1. Definizione induttiva di albero binario

Una classica definizione induttiva di albero binario è la seguente:

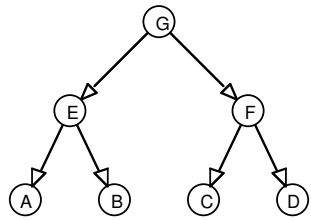
- l'albero vuoto è un albero binario;
- se $T_s$ e $T_d$ sono due alberi binari, allora l'albero che ha un nodo radice e $T_s$ e $T_d$ come sottoalberi è un albero binario;
- nient'altro è un albero binario.

I due sottoalberi $T_s$ e $T_d$ di un albero binario non vuoto $T$ vengono detti rispettivamente **sottoalbero sinistro** e **sottoalbero destro** di $T$.




### 12.2.2. Alberi binari completi

Un albero binario viene detto **completo** se ogni nodo che non sia una foglia ha esattamente due figli e le foglie sono tutte allo stesso livello nell'albero:




È facile verificare che un albero binario completo non vuoto di profondità $k$ ha esattamente $2^{k+1} - 1$ nodi: infatti, vi sono esattamente $2^k$ foglie a livello $k$ e $2^k - 1$ nodi che non sono foglie, e la somma di queste due quantità dà proprio $2^{k+1} - 1$.

## 12.3. Tipo astratto AlberoBin

Il tipo di dato astratto AlberoBin può essere definito come segue.

**TipoAstratto AlberoBin(T)**

**Domini**

**AlberoBin** : dominio di interesse del tipo

**T** : dominio dei valori associati ai nodi degli alberi binari

**Funzioni**

**albBinVuoto() $\mapsto$ AlberoBin**
pre: nessuna
post: RESULT è l'albero binario vuoto (ossia senza nodi)

**creaAlbBin(T r, AlberoBin s, AlberoBin d) $\mapsto$ AlberoBin**
pre: nessuna
post: RESULT è l'albero binario che ha r come radice, s come sottoalbero sinistro, e d come sottoalbero destro

**estVuoto(AlberoBin a) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se a è vuoto, false altrimenti

**radice(AlberoBin a) $\mapsto$ T**
pre: a non è vuoto
post: RESULT è il valore associato al nodo che è radice di a

**sinistro(AlberoBin a) $\mapsto$ AlberoBin**
pre: a non è vuoto
post: RESULT è il sottoalbero sinistro di a

**destro(AlberoBin a) $\mapsto$ AlberoBin**
pre: a non è vuoto
post: RESULT è il sottoalbero destro di a

**FineTipoAstratto**

## 12.4. Rappresentazione indicizzata di alberi binari

Vi sono diversi modi per rappresentare alberi binari nei linguaggi di programmazione. Uno dei più semplici è la rappresentazione indicizzata, realizzabile mediante array.

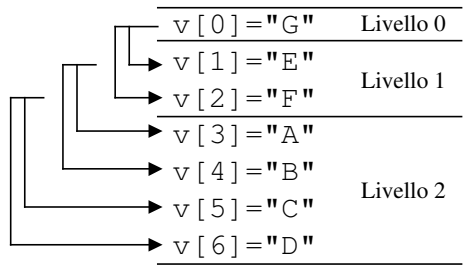
### 12.4.1. Rappresentazione indicizzata di alberi binari completi

La rappresentazione indicizzata mediante array è particolarmente adatta per rappresentare alberi binari completi. Dato un array $v$ di dimensione $n$:

- se $n = 0$, l'array $v$ rappresenta l'albero vuoto;

- se $n > 0$, l'array $v$ rappresenta un albero binario completo i cui nodi corrispondono agli indici nell'intervallo $[0..n-1]$ di $v$. In particolare si ha che:
  - l'albero rappresentato ha $n$ nodi;
  - il contenuto informativo del nodo $i$ è $v[i]$;
  - la radice corrisponde all'indice 0;
  - il figlio sinistro di $i$ ha indice $2i + 1$;
  - il figlio destro di $i$ ha indice $2i + 2$.

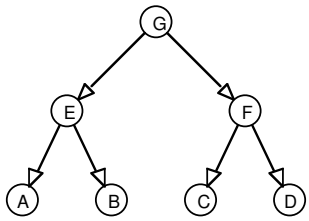
Ad esempio, l'albero binario completo mostrato nella figura precedente può essere rappresentato mediante l'array $v$ di dimensione $n = 7$ come mostrato sotto:




Si noti che i figli del nodo 'F', che corrisponde all'indice 2, sono 'C' e 'D', e questi corrispondono agli indici $5 = 2 \cdot 2 + 1$ e $6 = 2 \cdot 2 + 2$.

Un modo pratico per costruire una rappresentazione indicizzata di un albero binario consiste nel disporre nell'array, a partire da $v[0]$, i valori dei nodi presi da sinistra verso destra su ogni livello a partire dal livello zero fino al livello massimo: questo appare evidente se si osservano le due figure precedenti.

*Esempio:* L'albero binario completo mostrato nella figura seguente:



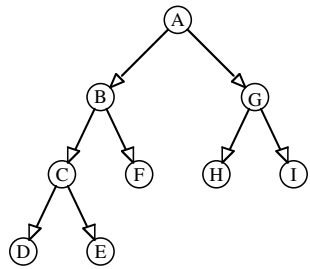
può essere rappresentato in modo indicizzato come array di caratteri:

```
char [] v = { 'G', 'E', 'F', 'A', 'B', 'C', 'D' };
```

### 12.4.2. Rappresentazione indicizzata di alberi binari non completi

È possibile utilizzare la rappresentazione indicizzata anche per alberi binari non completi, assumendo sia indicato esplicitamente se un nodo $i$ è presente nell'albero. Tuttavia, in questo modo è necessario fare spazio nell'array anche per nodi non effettivamente presenti nell'albero, e questo può essere estremamente dispendioso in termini di memoria richiesta.

*Esempio:* L'albero binario non completo mostrato nella figura seguente:




può essere rappresentato in C in modo indicizzato come segue:

```
char[] v = { 'A', 'B', 'G', 'C', 'F', 'H', 'I', 'D',
             'E', '0', '0', '0', '0', 'L', '0' };
```

Si dove '0' viene usato per indicare che il nodo non esiste.

### 12.4.3. Rappresentazione indicizzata generale di alberi binari

La rappresentazione generale di un albero binario sotto forma di array richiede la dichiarazione di un tipo array di nodi dell'albero:

```
#define MaxNodiAlbero 100

typedef  ...  TipoInfoAlbero;

struct StructAlbero {
  TipoInfoAlbero info;
  bool esiste;
};

typedef struct StructAlbero TipoNodoAlbero;

typedef TipoNodoAlbero TipoAlbero[MaxNodiAlbero];
```

La scelta in questo caso è stata quella di definire staticamente la dimensione dell'array e di usare un campo booleano esiste per indicare se un elemento dell'array rappresenta un nodo dell'albero o meno.

*Esempio:* Il seguente frammento di codice indica come stampare i figli di un nodo i assumendo di avere un albero binario rappresentato in modo indicizzato:

```
TipoAlbero albero;
...
// i: nodo di cui stampare i figli
if (i<0 || i>=MaxNodiAlbero)
  printf("Indice errato...");
else {
  if (2*i+1 < MaxNodiAlbero && albero[2*i+1].esiste)
    printf("sinistro di %d -> %d\n", i, albero[2*i+1].
    info);
  else
    printf("%d non ha figlio sinistro\n", i);
  if (2*i+2 < MaxNodiAlbero && albero[2*i+2].esiste)
    printf("destro di %d -> %d\n", i, albero[2*i+2].
    info);
  else
    printf("%d non ha figlio destro\n", i);
}
```

Si noti che un sottoalbero per essere definito deve avere un indice all'interno del massimo consentito ed, in tal caso, il campo esiste deve essere true.

## 12.5. Rappresentazione collegata di alberi binari

Gli alberi binari possono essere rappresentati anche mediante la **rappresentazione collegata**. Usando questo metodo, ogni nodo viene

rappresentato mediante una struttura avente i seguenti campi:

1. informazione associata al nodo;
2. riferimento al figlio sinistro del nodo;
3. riferimento al figlio destro del nodo.

È possibile astrarre questo concetto in C definendo un tipo TipoAbero che è un puntatore ad un nodo di un albero binario:

```
struct StructAlbero {
  TipoInfoAlbero info;
  struct StructAlbero *destro, *sinistro;
};

typedef struct StructAlbero TipoNodoAlbero;

typedef TipoNodoAlbero* TipoAlbero;
```

Osserviamo che:

- il campo info memorizza il contenuto informativo del nodo. Il tipo TipoInfoAlbero verrà definito di volta in volta a seconda della natura delle informazioni che si vogliono memorizzare nei nodi dell'albero binario, analogamente a quanto visto per le strutture collegate lineari.
- i campi sinistro e destro sono riferimenti ai nodi figlio sinistro e destro del nodo, rispettivamente. Per indicare che il nodo non ha uno o entrambi i figli useremo il valore NULL. Un nodo foglia dovrà pertanto avere entrambi i campi sinistro e destro posti a NULL.

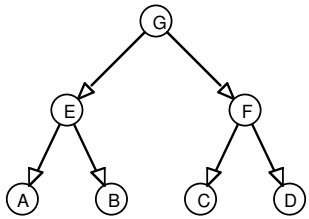
*Nota:* il tipo TipoNodoAlbero è simile al tipo TipoNodoSCL visto a proposito delle strutture collegate lineari. Il fatto che ogni nodo dell'albero binario contiene due puntatori indica che si ha a che fare con una **struttura collegata non lineare**.

Un albero binario nella rappresentazione collegata sarà rappresentato quindi nel seguente modo:

- L'albero binario *vuoto* viene rappresentato usando il valore NULL.

- Un albero binario *non vuoto* viene rappresentato mantenendo il riferimento al nodo radice dell'albero. Osserviamo che partendo dal riferimento alla radice dell'albero è possibile utilizzare i collegamenti ai nodi figli per raggiungere ogni altro nodo dell'albero. In base a questa assunzione, *ogni valore di tipo puntatore a TipoNodoAlbero consente di accedere a tutto il sottoalbero alla cui radice esso punta*.

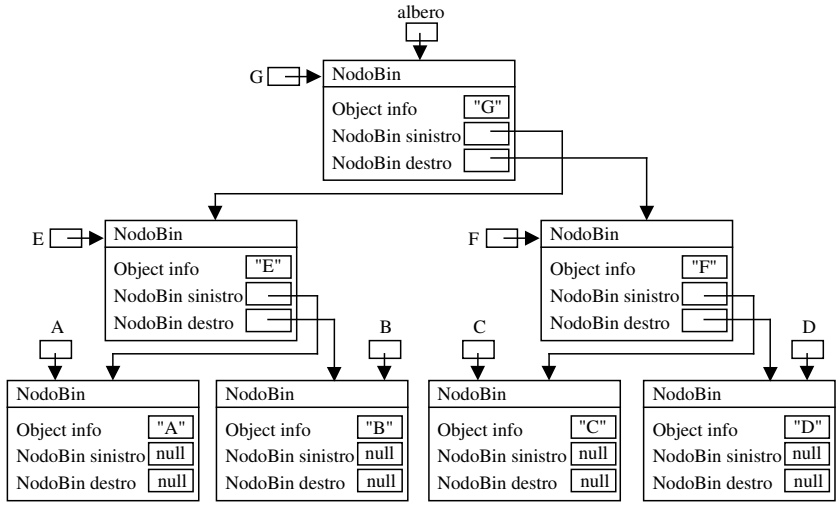
*Esempio:* L'albero binario completo mostrato nella figura seguente:




può essere rappresentato in C in modo collegato come illustrato nel seguente frammento di programma:

```
TipoNodoAlbero* nodoalb_alloc(TipoInfoAlbero e) {
  TipoNodoAlbero* r = (TipoNodoAlbero*) malloc(sizeof(
    TipoNodoAlbero));
  r->info = e; r->destro = NULL; r->sinistro = NULL;
  return r;
}

TipoAlbero A = nodoalb_alloc('A');
TipoAlbero B = nodoalb_alloc('B');
TipoAlbero C = nodoalb_alloc('C');
TipoAlbero D = nodoalb_alloc('D');
TipoAlbero E = nodoalb_alloc('E');
TipoAlbero F = nodoalb_alloc('F');
TipoAlbero G = nodoalb_alloc('G');
E->sinistro = A;    E->destro = B;
F->sinistro = C;    F->destro = D;
G->sinistro = E;    G->destro = F;
TipoAlbero albero = G;
```

Dapprima vengono creati i nodi dell'albero, e poi vengono inizializzati i campi corrispondenti ai sottoalberi, come illustrato nella seguente figura:




La variabile albero rappresenta quindi l'albero binario e contiene un riferimento alla radice dell'albero.

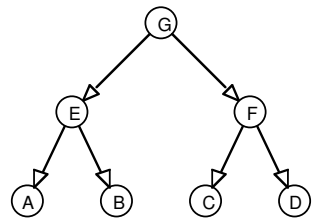
### 12.5.1. Implementazione del tipo astratto AlberoBin

Una realizzazione del tipo astratto AlberoBin usando la rappresentazione collegata è mostrata di seguito. Il tipo astratto AlberoBin viene implementato mediante la struttura C TipoAlbero, il tipo degli elementi T è definito in C con il tipo TipoInfoAlbero.

```
// l'albero vuoto e' rappresentato dal valore NULL
TipoAlbero albBinVuoto() {
  return NULL;
}

// costruttore di un albero a partire dai sottoalberi
    e info radice
TipoAlbero creaAlbBin(TipoInfoAlbero infoRadice,
    TipoAlbero sx, TipoAlbero dx) {
  TipoAlbero a = (TipoAlbero) malloc(sizeof(
    TipoNodoAlbero));
  a->info=infoRadice;
  a->sinistro=sx;
  a->destro=dx;
  return a;
}
```

```
// predicato che verifica se l'albero e' vuoto
bool estVuoto(TipoAlbero a) {
  return (a == NULL);
}

// funzione che restituisce il contenuto della radice
TipoInfoAlbero radice(TipoAlbero a) {
  if (a==NULL) {
    printf("ERRORE accesso albero vuoto\n");
    return ERRORE_InfoAlbero;
  }
  else
    return a->info;
}

// funzione che restituisce il sottoalbero sinistro
TipoAlbero sinistro(TipoAlbero a) {
  if (a==NULL) {
    printf("ERRORE accesso albero vuoto\n");
    return NULL;
  }
  else
    return a->sinistro;
}

// funzione che restituisce il sottoalbero destro
TipoAlbero destro(TipoAlbero a) {
  if (a==NULL) {
    printf("ERRORE accesso albero vuoto\n");
    return NULL;
  }
  else
    return a->destro;
}
```

*Esempio:* L'albero binario:




può essere costruito come mostrato nel seguente programma:

```
TipoAlbero A = creaAlbBin('A', albBinVuoto(),
    albBinVuoto());
TipoAlbero B = creaAlbBin('B', albBinVuoto(),
    albBinVuoto());
TipoAlbero C = creaAlbBin('C', albBinVuoto(),
    albBinVuoto());
TipoAlbero D = creaAlbBin('D', albBinVuoto(),
    albBinVuoto());
TipoAlbero E = creaAlbBin('E', A, B);
TipoAlbero F = creaAlbBin('F', C, D);
TipoAlbero G = creaAlbBin('G', E, F);
```

## 12.6. Rappresentazione parentetica di alberi binari

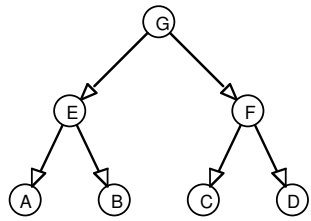
Un'altra possibile rappresentazione di un albero binario è la **rappresentazione parentetica**, basata su stringhe di caratteri. Assumiamo che $f(T)$ sia la stringa che denota l'albero binario $T$;

1. l'albero vuoto è rappresentato dalla stringa `"( )"`, cioè $f(\text{albero vuoto})$=`"( )"`;

2. un albero binario con radice $x$ e sottoalberi sinistro e destro $T_1$ e $T_2$ è rappresentato dalla stringa `"( "` + $x$ + $f(T_1)$ + $f(T_2)$ + `" )"`.

*Esempio:* L'albero mostrato nella seguente figura può essere rappresentato mediante la stringa:

```
"( G ( E ( A ( ) ( ) ) ( B ( ) ( ) ) ) "  +
    "( F ( C ( ) ( ) ) ( D ( ) ( ) ) ) )".
```



## 12.7. Costruzione di un albero binario da rappresentazione parentetica

Vediamo ora un modo conveniente per costruire un albero binario data una sua rappresentazione parentetica. In particolare, vediamo come progettare una funzione LeggiAlbero che fornisca questa funzionalità.

```
TipoAlbero LeggiAlbero(char *nome_file) {
  TipoAlbero result;
  FILE *file_albero;
  file_albero = fopen(nome_file, "r");
  result = LeggiSottoAlbero(file_albero);
  fclose(file_albero);
  return result;
}
```

La funzione ha come parametro in ingresso una stringa che rappresenta il nome del file che contiene l'albero e, oltre a gestire le operazioni di apertura e chiusura chiama la funzione ausiliaria LeggiSottoAlbero, che restituisce un puntatore al sottoalbero letto.

```
TipoAlbero LeggiSottoAlbero(FILE *file_albero) {
  char c;
  TipoInfoAlbero i;
  TipoAlbero r;
  c = LeggiParentesi(file_albero);   /* legge la
    parentesi aperta */
  c = fgetc(file_albero);   /* carattere successivo */
  if (c == ')')
    return NULL;   /* se legge () allora l'albero e'
    vuoto */
  else {
    fscanf(file_albero, "%c", &i);   /* altrimenti
    legge la radice */
    r = (TipoAlbero) malloc(sizeof(TipoNodoAlbero));
    r->info = i;
    /* legge i sottoalberi */
    r->sinistro = LeggiSottoAlbero(file_albero);
    r->destro = LeggiSottoAlbero(file_albero);
    c = LeggiParentesi(file_albero);   /* legge la
    parentesi chiusa */
    return r;
  }
}
```

Si noti che:

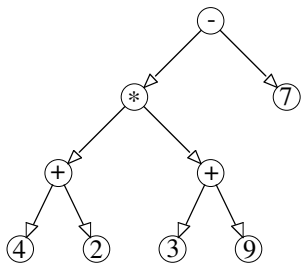
- quando l'albero è vuoto, il carattere che segue la parentesi aperta '(' deve necessariamente essere la parentesi chiusa ')' (senza spazi);
- al contrario quando il nodo non è l'albero vuoto, l'informazione di tipo carattere da associare al nodo deve essere preceduta da uno spazio ' '.
- Se le informazioni sono di tipo int, o altro tipo primitivo numerico il tipo della variabile usata per leggere l'informazione ed il formato della istruzione fscanf devono essere aggiornate coerentemente;
- se le informazioni sono di altro tipo, in particolare di tipo non primitivo per leggere l'informazione associata diventa necessario progettare una specifica funzione.

## 12.8. Rappresentazione di espressioni aritmetiche mediante alberi binari

Una espressione aritmetica può essere rappresentata mediante un albero binario in cui:

- i nodi non foglie rappresentano operazioni (es. +, -, *, / );
- le foglie rappresentano valori numerici (es. 2, 7, ecc.)

*Esempio:* L'espressione aritmetica $(4+2)\cdot(3+9)-7$ può essere rappresentata mediante l'albero binario:




L'albero, a sua volta, può essere rappresentato in modo parentetico come:

```
( -
    ( *
        ( +
            ( 4 ( ) ( ) )
            ( 2 ( ) ( ) )
        )
        ( +
            ( 3 ( ) ( ) )
            ( 9 ( ) ( ) )
        )
    )
    ( 7 ( ) ( ) )
)
```

Sebbene la corrispondenza tra la struttura dell'espressione e quella dell'albero binario sia immediata, la definizione della struttura dati da utilizzare per memorizzare le informazioni nel nodo richiede qualche accorgimento dato che alcuni nodi contengono degli operandi numerici ed altri contengono invece l'indicazione dell'operatore da applicare. Si possono sviluppare diverse alternative (ad esempio utilizzare una struttura con due campi, char ed int usando un carattere convenzionale per il campo char, quando il nodo è una foglia e contiene una informazione di tipo numerico. Lo pseudo-codice di una funzione valutaEspressione che, data una espressione rappresentata come albero binario, ne restituisce il valore calcolato può essere definito come:

```
int valutaEspressione(TipoAlbero a) {
  // caso albero vuoto: errore
  if (a==NULL) Errore albero vuoto;
  // caso nodo foglia: restituisce valore
  if (a->sinistro==NULL && a->destro==NULL)
     return  valore numerico;
  else { // caso nodo non foglia: calcola ricorsivamente
         // le sottoespressioni destra e sinistra
    int sin = valutaEspressione(a->sinistro);
    int des = valutaEspressione(a->destro);
    // applica operazione
    if (nodo-addizione) return sin + des;
    else ... analogamente per gli altri casi
    else Operazione non valida
  }
```

```
}
```

## 12.9. Visita in profondità di alberi binari

Diversamente dal caso delle strutture lineari, dove è possibile visitare tutti gli elementi in sequenza nell'ordine in cui sono rappresentati, nel caso di alberi binari vi sono molti modi diversi di esplorare i nodi. In particolare, vedremo alcuni algoritmi di visita che consentono di esplorare i nodi di un albero binario. Gli scopi di una visita possono essere molteplici: verificare la presenza di un dato elemento in un albero, calcolare proprietà globali di un albero, come profondità o numero totale di nodi, ecc.

La **visita in profondità** di un albero T può essere realizzata ricorsivamente come il seguente pseudo codice mostra:

```
if T non è vuoto {
   analizza il nodo radice di T
   visita il sottoalbero sinistro di T
   visita il sottoalbero destro di T
}
```

L'operazione di "analisi" di un nodo può essere semplicemente la stampa dell'informazione associata al nodo, ma può essere qualunque elaborazione di quell'informazione.

### 12.9.1. Proprietà della visita in profondità

L'algoritmo di visita in profondità visto precedentemente ha le seguenti proprietà fondamentali:

- termina in un numero finito di passi: infatti ogni chiamata ricorsiva viene applicata su un sottoalbero che è strettamente più piccolo dell'albero corrente;

- visita tutti i nodi di T: per ipotesi induttiva, l'algoritmo visita tutti i nodi dei sottoalberi sinistro e destro di T, se non sono vuoti. Poiché la radice di T viene anch'essa visitata, allora tutti i nodi di T vengono visitati;

- richiede un numero totale di passi proporzionale ad $n$, dove $n$ è il numero di nodi di T: infatti ogni nodo viene considerato al più una volta durante la visita (e questo avviene quando il nodo è la radice del sottoalbero corrente) e l'algoritmo effettua un numero costante di passi per passare da un nodo all'altro.

### 12.9.2. Tipi diversi di visite in profondità: in preordine, simmetrica, in postordine

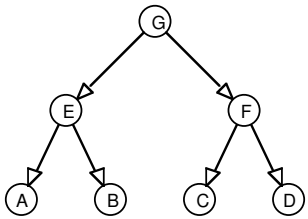
Le tre operazioni della visita in profondità:

- visita della radice;
- chiamata ricorsiva sul sottoalbero sinistro;
- chiamata ricorsiva sul sottoalbero destro;

possono essere realizzate in qualunque ordine. In particolare, consideriamo tre particolari ordini che danno luogo a tre diverse versioni di visita in profondità:

- **visita in preordine**: radice, sottoalbero sinistro, sottoalbero destro;
- **visita simmetrica**: sottoalbero sinistro, radice, sottoalbero destro;
- **visita in postordine**: sottoalbero sinistro, sottoalbero destro, radice.

*Esempio:* Consideriamo l'albero seguente:




Se l'operazione di visita è la stampa dell'informazione del nodo, si hanno i seguenti tre risultati applicando i diversi tipi di visita:

- visita in preordine: G E A B F C D
- visita simmetrica: A E B G C F D
- visita in postordine: A B E C D F G

Si noti che la radice G è la prima ad essere visitata nel caso di visita in preordine, è visitata a metà della visita nel caso di visita simmetrica, ed è visitata per ultima nel caso di visita in postordine.

## 12.10. Realizzazione della visita in profondità

Mostriamo ora come realizzare in C le operazioni di visita in profondità di un albero binario.

### 12.10.1. Rappresentazione tramite array

Consideriamo prima il caso della visita in preordine con la rappresentazione tramite array.

La visita inizia con la chiamata ad una funzione ausiliaria che ha come parametro ulteriore l'indice del nodo da visitare.

```
/* visita in preordine di un albero binario
   rappresentato con array */

void VisitaPreordine(TipoAlbero a)
{
  /* visita a partire dalla radice */
  VisitaIndicePre(a, 0);
}
```

La funzione ausiliaria è ricorsiva e l'ordine di esecuzione delle operazioni corrisponde a quello della visita in preordine.

```
/* visita in preordine */
void VisitaIndicePre(TipoAlbero a, int i) {
  if (i>=0 && i<MaxNodiAlbero && a[i].esiste) {
    /* albero non vuoto */
    Analizza(a[i].info);  // analizza la radice
    VisitaIndicePre(a, i * 2 + 1 ); // visita sinistro
    VisitaIndicePre(a, i * 2 + 2 ); // visita destro
  }
}
```

### 12.10.2. Rappresentazione tramite puntatori

Consideriamo ora la visita in profondità in postordine nella rappresentazione tramite struttura collegata:

```
void VisitaPostordine(TipoAlbero a) {
  if (a != NULL) {
    /* albero non vuoto */
    VisitaPostordine(a->sinistro); // visita sinistro
    VisitaPostordine(a->destro);   // visita destro
    Analizza(a->info);   // analizza la radice
  }
}
```

*Nota:* Semplicemente spostando la posizione dell'operazione di analisi si ottengono le tre varianti preordine, simmetrica, postordine.

## 12.11. Applicazioni delle visite

Mostriamo ora come adattare l'algoritmo di visita in profondità per realizzare alcune operazioni di interesse generale su alberi binari.

### 12.11.1. Calcolo della profondità di un albero

Si vuole definire una funzione che restituisce la profondità dell'albero passato come parametro:

```
/* calcola la profondita' di un albero binario */
int Profondita(TipoAlbero a)
{
  int s, d;
  if (a == NULL)
    return -1;   // l'albero vuoto ha profondita' -1
  else {
    // calcola la profondita' dei sottoalberi
    s = Profondita(a->sinistro);
    d = Profondita(a->destro);
    // l'albero complessivo ha profondita' max(s,d) +
    1
    return (Massimo(s, d) + 1);
  }
}
```

La funzione verifica dapprima se l'albero (o sottoalbero) passato come parametro è vuoto, nel qual caso restituisce come profondità $-1$. Questo è il passo base della ricorsione. Altrimenti, vengono calcolate ricorsivamente la profondità del sottoalbero sinistro e destro, e viene restituito il massimo di queste due quantità più uno.

```
/* determina il massimo fra due interi */
int Massimo(int a, int b) {
  if (a > b)
    return a;
  else
    return b;
}
```

### 12.11.2. Verifica della presenza di un elemento nell'albero

Una operazione di grande rilevanza applicativa è la ricerca di un dato valore in un albero binario. Si vuole definire una funzione che, dato un valore, restituisce true se il valore è presente nell'albero su cui è invocato, e false altrimenti. Nel caso in cui il valore venga trovato, la funzione deve restituire anche il puntatore al nodo corrispondente.

```
bool RicercaAlbero(TipoAlbero a, TipoInfoAlbero elem,
    TipoAlbero *posiz) {
  bool trovato = false; *posiz=NULL;
  if (a == NULL)
    trovato = false;
  else if (elem == a->info) { // elemento trovato
    *posiz = a;  trovato = true;
  }
  else {
    /* cerca nel sottoalbero sinistro */
    trovato = RicercaAlbero(a->sinistro, elem, posiz);
    if (!trovato)
      /* cerca nel sottoalbero destro */
      trovato = RicercaAlbero(a->destro, elem, posiz);
  }
  return trovato;
}
```

La funzione RicercaAlbero verifica dapprima il caso base in cui l'albero è vuoto, nel qual caso la risposta è false. Se invece l'albero non è vuoto, il valore è presente se e solo se è verificato almeno uno dei tre casi seguenti ed assegna il valore true alla variabile trovato, che altrimenti rimane con il valore false:

1. il valore è contenuto nella radice correntemente visitata (ovvero, elem == a->info esegue trovato=true);
2. il valore è contenuto nel sottoalbero sinistro della radice (ovvero, RicercaAlbero(a->sinistro, elem, posiz) restituisce true);

3. il valore è contenuto nel sottoalbero destro della radice (ovvero RicercaAlbero(a->destro, elem, posiz) restituisce true).

*Nota:* La funzione RicercaAlbero potrebbe dover visitare *tutti* i nodi prima di stabilire con certezza se un valore dato è presente o meno nell'albero. Il costo computazionale è quindi $O(n)$. Vedremo più avanti che usando alberi con determinate proprietà (alberi binari di ricerca) è possibile usare un metodo diverso e molto più efficiente per ricercare un valore.

*Nota:* Se il tipo delle informazioni memorizzate nel nodo dell'albero non è un tipo primitivo occorre definire una opportuna funzione che verifica l'uguaglianza di due elementi del tipo delle informazioni contenute nell'albero.

Analoghe funzioni che calcolano risultati relativi alle informazioni presenti nei nodi dell'albero senza modificare la struttura dell'albero possono essere implementate estendendo le relative funzioni ricorsive su strutture dati lineari, considerando la necessità di effettuare una ricorsione doppia per ogni nodo su entrambi i suoi sottoalberi.

Le funzioni che modificano il contenuto di un albero binario, ma non la sua struttura, possono essere implementate in maniera analoga alle relative funzioni che modificano il contenuto, ma non la struttura, di strutture dati lineari, tenendo conto della necessità in generale di effettuare l'operazione su entrambi i sottoalberi di ciascun nodo.

La struttura di una funzione che modifica il contenuto di un albero binario è la seguente.

```
void modifica(TipoAlbero a, ... ) {
  if (a != NULL) {
    a->info = ...;
    modifica(a->sinistro, ...);
    modifica(a->destro, ...);
  }
}
```

## 12.12. Alberi binari di ricerca

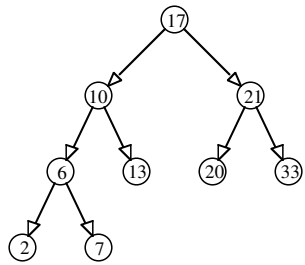
Concludiamo la trattazione sugli alberi binari discutendo un sottoinsieme degli alberi binari: gli **alberi binari di ricerca**. Gli alberi binari di ricerca sono strutture dati che consentono di cercare informazioni in modo molto efficiente. In un albero binario di ricerca, il dominio delle

informazioni associabili ai nodi deve essere un insieme su cui è definita una relazione di ordine totale.

Un albero binario di ricerca può essere:

- un albero vuoto, oppure
- un albero binario non vuoto tale che:
  - il valore di ogni nodo nel sottoalbero sinistro è minore o uguale al valore della radice rispetto alla relazione d'ordine;
  - il valore di ogni nodo nel sottoalbero destro è maggiore o uguale al valore della radice rispetto alla relazione d'ordine;
  - i sottoalberi sinistro e destro sono a loro volta alberi binari di ricerca.

*Esempio:* L'albero mostrato nella figura seguente è un albero binario di ricerca dove il dominio delle informazioni associate ai nodi è l'insieme dei numeri interi.




### 12.12.1. Verifica della presenza di un elemento in un albero binario di ricerca

Mostriamo ora una variante della funzione vista precedentemente che sfrutta le proprietà peculiari degli alberi binari di ricerca per verificare la presenza di un elemento nell'albero in modo molto efficiente. Assumiamo che i nodi dell'albero contengano numeri interi come nell'esempio precedente.

```
bool presenteAlberoRicerca(TipoAlbero a,
    TipoInfoAlbero elem) {
  if (a==NULL)
    return false;
  else
    if (a->info == elem) // elemento nella radice
      return true;
    else if (a->info > elem) // l'elemento puo' essere
                        // solo nel sottoalbero sin
      return presenteAlberoRicerca(a->sinistro, elem);
    else // l'elemento puo' essere solo nel
    sottoalbero des
      return presenteAlberoRicerca(a->destro, elem);
}
```

Se l'albero è vuoto, il metodo restituisce subito false. Altrimenti, l'elemento elem viene confrontato con la radice (a->info == elem):

- se il confronto ha successo, l'elemento cercato è nella radice (return true);
- se il valore nella radice dell'albero è maggiore di quello cercato, allora elem precede strettamente `radice()`, e quindi può solo essere presente nel sottoalbero sinistro (return presenteAlberoRicerca(a->sinistro, elem));
- se il valore nella radice dell'albero è minore di quello cercato, allora elem segue strettamente `radice()`, e quindi può solo essere presente nel sottoalbero destro (return presenteAlberoRicerca(a->destro, elem)).

*Nota:* La funzione presenteAlberoRicerca effettua al più una chiamata ricorsiva, e non due come nella versione su alberi binari generali. In questo modo, scartando ad ogni chiamata ricorsiva un intero sottoalbero, il metodo esplorerà al massimo tanti nodi quanto è la profondità dell'albero, e non tutti i nodi dell'albero. Se il metodo viene applicato ad un albero binario di ricerca completo con $n$ nodi, esso richiede un numero di passi proporzionale alla profondità dell'albero, che è $\log_2 n$. Questo metodo è pertanto sostanzialmente più efficiente ($(O(log(n))$), rispetto al procedimento generale che ha costo $O(n)$.

## 12.13. Accesso all'albero per livelli

In diverse situazioni è necessario accedere ai nodi di un albero considerando il livello a cui si trovano.

### 12.13.1. Stampa dei nodi di un livello

Mostriamo una funzione che, presi in input un albero $a$ (contenente valori interi) ed un livello $l$, stampa il contenuto dei nodi di $a$ che si trovano al livello $l$. La funzione implementa una visita in profondità ricorsiva in cui il livello d'interesse viene decrementato ogni volta che si scende di livello. Il contenuto della radice viene stampato solo quando tale livello è pari a 0.

```
void stampaNodiLivello(TipoAlbero a, int livello){
    if (estVuoto(a))
      return;
    else if (livello == 0) {
      printf("%d ", radice(a));
    }
    else {
      stampaNodiLivello(sinistro(a),livello-1);
      stampaNodiLivello(destro(a),livello-1);
    }
}
```

### 12.13.2. Insieme dei nodi di un livello

Mostriamo a seguire una funzione che restituisce l'insieme degli elementi presenti nei nodi di un dato livello usando una implementazione dell'insieme con side-effect.

```
Insieme* elementiLivello(TipoAlbero a, int livello) {
    Insieme* ris = insiemeVuoto();
    elementiLivelloRic(a,livello,ris);
    return ris;
}

void elementiLivelloRic(TipoAlbero a, int livello,
    Insieme* ins) {
    if (estVuoto(a)) return;
    else if (livello == 0)
      inserisci(ins,radice(a));
    else {
      elementiLivelloRic(sinistro(a),livello-1,ins);
      elementiLivelloRic(destro(a),livello-1,ins);
    }
}
```

In questo caso viene usata una funzione ausiliaria per effettuare la ricorsione sui sottoalberi del nodo corrente. Anche questa funzione è

essenzialmente una visita in profondità ricorsiva in cui viene usato un parametro per individuare il livello della radice del sottoalbero corrente.

La seguente variante usa un'implementazione funzionale dell'insieme e non fa uso della funzione ausiliaria.

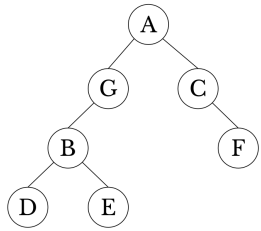
```
Insieme elementiLivelloB(TipoAlbero a, int livello) {
  if (estVuoto(a) || livello < 0)
    return insiemeVuoto();
  else if (livello == 0)
    return inserisci(insiemeVuoto(),radice(a));
  else {
    Insieme s = elementiLivelloB(sinistro(a),livello
    -1);
    Insieme d = elementiLivelloB(destro(a),livello-1);
    return unione(s,d);
  }
}
```

## 12.14. Operazioni di modifica di alberi

Le operazioni del tipo astratto AlberoBin sopra introdotte permettono la costruzione e la visita di un albero binario. Mostriamo ora, a titolo esemplificativo, l'implementazione di alcune operazioni di inserimento e cancellazione.

### 12.14.1. Inserimento di un nodo come foglia in posizione data

Dato un albero $a$ vogliamo inserire un nuovo nodo come foglia, in posizione specificata tramite una stringa opportuna. La stringa è una sequenza di caratteri 'd' ed 's', dove 's' indica il figlio sinistro e 'd' il figlio destro di un nodo. Partendo dalla radice dell'albero e dal primo carattere della stringa, ci si sposta sul figlio destro o sinistro a seconda del carattere incontrato, finché la stringa finisce o si è raggiunto un nodo privo del figlio specificato. Nel primo caso, l'albero non viene modificato, nel secondo si inserisce un nuovo nodo con l'elemento fornito in input nella posizione in cui ci si dovrebbe spostare. Ad esempio, dato il seguente albero:




la stringa "ssd" rappresenta il nodo E, mentre la stringa "sd" rappresenta il figlio destro (non presente) del nodo G.

Nel primo caso, non deve essere effettuata nessuna modifica all'albero, mentre nel secondo caso, deve essere inserito un nuovo nodo come figlio destro di G.

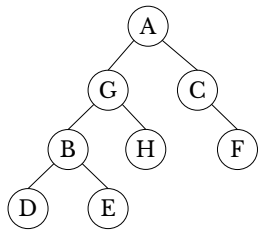
La funzione seguente inserisce un elemento come nodo in posizione data.

```
void insFogliaInPosizione(TipoAlbero* a,
                  TipoInfoAlbero e, char* p) {
    if (a == NULL)
      printf("ERRORE: puntatore NULL")
    else if (*a == NULL)
      *a = creaAlbBin(e,NULL,NULL);
    else if (*p=='\0')
      return;
    else if (*p=='s')
      insFogliaInPosizione(&((*a)->sinistro), e, p+1);
    else if (*p=='d')
      insFogliaInPosizione(&((*a)->destro), e, p+1);
}
```

Ad esempio se a è l'albero mostrato sopra, l'invocazione:

```
inserisciFogliaInPosizione(a, 'H', "sd")
```

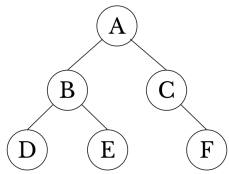
modifica a producendo il seguente risultato:



Si osservi che fornendo in input qualunque estensione della stringa "sd", as esempio "sds" o "sdsssd", etc. si ottiene lo stesso risultato, in quanto il nodo viene aggiunto nel momento in cui non è pi'au possible proseguire lungo il percorso rappresentato dalla stringa.

### 12.14.2. Inserimento di un nodo interno in posizione data

L'inserimento di un nodo interno pone difficoltà aggiuntive rispetto all'inserimento come foglia, in quanto occorre gestire il posizionamento del sottoalbero che viene "staccato" per fare posto al nuovo nodo. Per vedere ciò, si consideri nuovamente il seguente albero:




Si supponga di voler inserire un nuovo nodo G nella posizione attualmente occupata dal nodo B. In che relazione deve essere il nuovo nodo con il sottoalbero avente B per radice? In generale, tale relazione dipende dal caso in esame.

Mostriamo a seguire un'estensione della funzione inserisciFogliaInPosizione che inserisce un nuovo nodo nella posizione identificata da una stringa fornita in input (secondo la convenzione mostrata sopra). La funzione prende in input un parametro aggiuntivo di tipo carattere, usato per determinare se il sottoalbero disconnesso per far posto al nuovo nodo debba essere figlio destro (carattere 'd') o sinistro ('s') del nuovo nodo. Nel caso la stringa identifichi una foglia, la funzione si comporta in maniera analoga a inserisciFogliaInPosizione.

```
void insInPosizione(TipoAlbero* a,
          TipoInfoAlbero e, char* p, char f) {
  if (a == NULL)
    printf("ERRORE: puntatore NULL");
  else if (*a == NULL)
    *a = creaAlbBin(e,NULL,NULL);
  else if (*p=='\0') {
    if (f == 'd')
      *a = creaAlbBin(e,NULL,*a);
    else if (f == 's')
      *a = creaAlbBin(e,*a,NULL);
  }
  else if(*p=='s')
    insInPosizione(&((*a)->sinistro), e, p+1, f);
  else if(*p=='d')
    insInPosizione(&((*a)->destro), e, p+1, f);
}
```

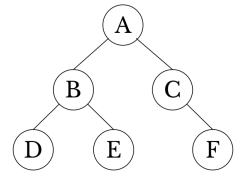
### 12.14.3. Cancellazione di un nodo

Quando si vuole cancellare un nodo $n$ da un albero binario, possono verificarsi due casi:

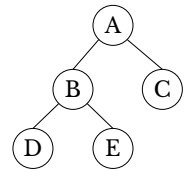
- $n$ ha uno o nessun figlio
- $n$ ha entrambi i figli

Nel primo caso si procede semplicemente eliminando il nodo $n$. Nel caso questo sia una foglia, nessuna ulteriore operazione è necessaria. Se invece $n$ ha un figlio, esso viene usato per rimpiazzare $n$.

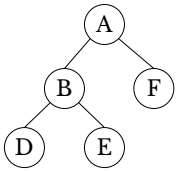
Nel seguente albero di esempio:




L'eliminazione del nodo F deve produrre il seguente risultato:

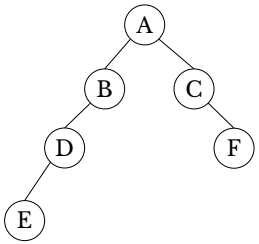


L'eliminazione del nodo C deve produrre il seguente risultato:

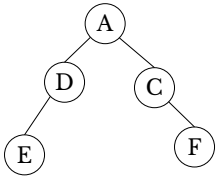



Nel secondo caso, invece, non esiste una soluzione unica e la scelta di come gestire i sottoalberi del nodo eliminato dipende dal caso in esame. A scopo illustrativo, mostriamo una possible soluzione in cui il sottoalbero destro del nodo da eliminare viene prima reso sottoalbero sinistro della foglia più a sinistra del sottoalbero con radice il nodo da eliminare. Questo comporta che il nodo selezionato abbia un solo figlio, per cui l'eliminazione ricade in uno dei casi precedenti.

Ad esempio, l'eliminazione del nodo B dal primo albero viene effettuata rendendo dapprima il nodo E figlio sinistro della foglia D, ottenendo il seguente albero:




e successivamente eliminando il nodo B, ottenendo pertanto il seguente risultato:




Mostriamo di seguito una possibile implementazione della funzione. La funzione eliminaNodo elimina la prima occorrenza del nodo che incontra. Si osservi che la funzione restituisce un valore booleano per indicare se l'eliminazione abbia avuto luogo (valore *true*) o meno (*false*). Inoltre, viene usata la funzione ausiliaria fogliaSinistra per individuare la foglia più a sinistra di un albero.

```
void aggiungiSinistra(TipoAlbero *a, TipoAlbero b) {
    if (*a == NULL)
        *a = b;
    else
        aggiungiSinistra(&((*a)->sinistro), b);
}
```

```
bool eliminaNodo(TipoAlbero* a, TipoInfoAlbero e){
  if (*a == NULL)
    return false;
  else if ((*a)->info == e) {
    TipoAlbero dx = (*a)->destro;
    if (dx != NULL)
      aggungiSinistra(a,(*a)->destro);
    TipoAlbero n = *a; // Elimina radice di *a
    *a = (*a)->sinistro;
    free(n); // Dealloca il nodo eliminato
    return true;
  }
  else
    return (eliminaNodo(&((*a)->sinistro),e) ||
            eliminaNodo(&((*a)->destro),e));
}
```

Si osservi che la funzione mostrata gestisce correttamente anche il caso in cui il nodo da eliminare sia una foglia o abbia un solo figlio.

## 12.15. Visita in profondità iterativa

La visita in profondità sfrutta una pila per mantenere l'ordine di visita dei nodi. Nell'implementazione ricorsiva ciò avviene in maniera implicita in quanto ad ogni invocazione ricorsiva corrisponde l'allocazione di un nuovo record di attivazione in cima alla pila. Lo stesso approccio può essere implementato in maniera esplicita, tramite l'uso di una pila, procedendo iterativamente.

```
sia P una pila vuota di nodi di un albero T
inserisci la radice di T in P
while (P non è vuota) {
    estrai il nodo n in testa alla pila P
    visita n
    inserisci il figlio destro di n in P
```

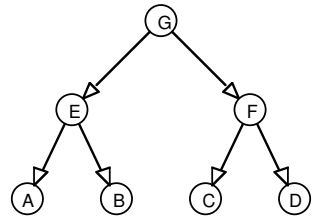
```
    inserisci il figlio sinistro di n in P
}
```

Mostriamo a seguire tale implementazione.

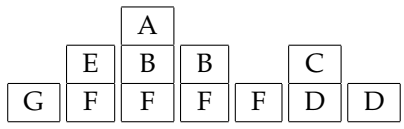
```
void visitaProfonditaIterativa(TipoAlbero a) {
    if (a == NULL) return;
    Pila* p = pilaVuota();
    push(p,a);
    while (!estVuota(p)) {
        TipoAlbero n = top(p);
        analizza(n);
        pop(p);
        if (n->destro != NULL)
          push(p,n->destro);
        if (n->sinistro != NULL)
          push(p,n->sinistro);
    }
}
```

Si osservi che l'ordine di inserimento dei nodi nella pila è invertito rispetto all'ordine con cui vengono visitati i sottoalberi destro e sinistro nella visita ricorsiva.

Consideriamo ora l'evoluzione della pila p a seguito dell'invocazione della funzione visitaProfonditaIterativa sull'albero seguente:




Mostriamo lo stato della pila all'inizio di ogni iterazione, quando viene effettuato il test del ciclo while:




Poiché ad ogni iterazione la funzione visita l'elemento affiorante della pila (rimpiazzandolo con i suoi figli), è immediato verificare che l'ordine di visita dei nodi è il seguente:

G E A B F G D

## 12.16. Visita in ampiezza di alberi binari

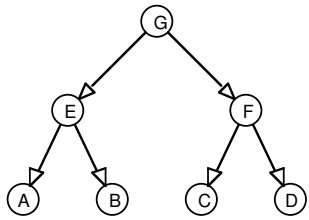
Una modalità alternativa di visita dei nodi di un albero è la cosiddetta visita in "ampiezza". Con questo termine si intende una visita in cui un nodo al livello $l$ viene visitato solo dopo che tutti i nodi al livello $l-1$ sono stati visitati. Contrariamente a quanto accade per la visita in profondità, l'implementazione della visita in ampiezza risulta più agevole nella variante iterativa; questa variante sfrutta una coda per memorizzare i nodi da visitare in maniera tale da garantire che l'ordine di visita corrisponda a quello di inserimento.

La **visita in ampiezza** di un albero T può essere realizzata iterativamente come mostrato dal seguente pseudocodice:

```
sia C una coda vuota di nodi di un albero
inserisci la radice di T in C
while (C non è vuota) {
    estrai il nodo n in testa alla coda C
    visita n
    inserisci il figlio sinistro di n in C
    inserisci il figlio destro di n in C
}
```

### 12.16.1. Esempio di visita in ampiezza di alberi binari

*Esempio:* Consideriamo l'albero seguente:




I nodi dell'albero vengono visitati nel seguente ordine:

G E F A B C D

### 12.16.2. Proprietà della visita in ampiezza

L'algoritmo di visita in ampiezza visto precedentemente ha le seguenti proprietà fondamentali:

- termina in un numero finito di passi: in un albero binario, nessun nodo è padre dei suoi antenati (incluso il nodo stesso) quindi, una volta estratto, non viene più inserito nella coda. Questo, insieme all'osservazione che un albero binario contiene un numero finito di nodi e ad ogni iterazione viene estratto esattamente un nodo, implica che la coda si svuota in un numero finito di passi, quindi l'algoritmo termina.

- visita tutti i nodi di T: è facile vedere che se un nodo viene inserito nella coda, sarà necessariamente visitato e tutti i suoi figli saranno inseriti nella coda a loro volta. Questo processo termina solo quando tutti i nodi sono stati inseriti, quindi estratti.

- richiede un numero totale di passi proporzionale ad $n$, dove $n$ è il numero di nodi di T: ogni nodo viene inserito esattamente una volta nella coda, e l'algoritmo effettua un numero costante di passi per inserirne i figli.

### 12.16.3. Realizzazione della visita in ampiezza

Mostriamo ora l'implementazione in C della visita in ampiezza di un albero binario. Assumiamo che siano disponibili:

- un'implementazione con side-effect del tipo astratto Coda i cui elementi siano di tipo TipoAlbero (si faccia riferimento alle implementazioni dei capitoli precedenti). Non è necessaria alcuna assunzione circa la modalità di rappresentazione della coda.

- un'implementazione con rappresentazione collegata del tipo astratto Albero Binario contenente elementi di tipo T (si faccia riferimento all'implementazione sopra riportata).

```
void visitaAmpiezza(TipoAlbero a){
    if (estVuoto(a)) return;
    Coda* c = codaVuota();
    inCoda(c,a);
    while(!estVuota(c)){
        TipoAlbero n = primo(c);
        analizza(n);
        outCoda(c);
        if (!estVuoto(sinistro(n)) inCoda(c,sinistro(n
    ));
        if (!estVuoto(destro(n)) inCoda(c,destro(n));
    }
}
```

Nel caso in cui si dovesse realizzare la funzione di visita accedendo alla rappresentazione, sarebbe sufficiente rimpiazzare le funzioni del tipo astratto con le operazioni sulla rappresentazione. A titolo esemplificativo mostriamo il codice per la realizzazione della visita nel caso di rappresentazione collegata (riportata per comodità).

```
typedef struct StructAlbero {
  TipoInfoAlbero info;
  struct StructAlbero* destro = NULL;
  struct StructAlbero* sinistro = NULL;
} TipoNodoAlbero;

typedef TipoNodoAlbero* TipoAlbero;

void visitaAmpiezza(TipoAlbero a){
    if (a == NULL) return;
    Coda* c = codaVuota();
    inCoda(c,a);
    while(!estVuota(c)){
        TipoAlbero n = a;
        analizza(n);
        outCoda(c);
        if (n->sinistro != NULL) inCoda(c,n ->
    sinistro);
        if (n->destro != NULL) inCoda(c,n -> destro);
    }
}
```

## 12.17. Applicazioni della visita in ampiezza

In molte situazioni le visite in profondità e in ampiezza possono essere utilizzate in maniera equivalente. Si pensi ad esempio al conteggio del numero di nodi di un albero o alla verifica della presenza di un

elemento in un albero. Esistono tuttavia situazioni in cui la visita in ampiezza si presta in maniera più naturale a realizzare la funzione desiderata; ne mostriamo a seguire un esempio.

### 12.17.1. Ricerca di un elemento a profondità minima

Si vuole definire una funzione che, dati un albero binario $a$ ed un elemento $e$, restituisca il riferimento ad un nodo di $a$ di livello minimo contenente $e$ ($NULL$ se non presente). Mostriamo l'implementazione della funzione nel caso di rappresentazione collegata dell'albero.

```
TipoNodoAlbero* nodoProfonditaMin(TipoAlbero a,
    TipoInfoAlbero e){
  if (a == NULL) return;
    TipoNodoAlbero* n;
    Coda* c = codaVuota(); // Coda di TipoAlbero (
    ovvero TipoNodoAlbero*)
    inCoda(c,a);
    while(!estVuota(c)){
        n = primo(c);
        if (n->info == e) return n;
        outCoda(c);
        if (n->sinistro!=NULL) inCoda(c,n->sinistro);
        if (n->destro!=NULL) inCoda(c,n->destro);
    }
    return NULL;
}
```

La funzione restituisce il primo nodo visitato contenente $e$. Poiché essa implementa una visita in ampiezza (come si può vedere confrontando la struttura della funzione con quella della visita in ampiezza), tale nodo è necessariamente quello di profondità minima.

Si osservi che con questa implementazione non è sempre necessaria una visita esaustiva dell'albero, in quanto la funzione termina non appena viene trovato un nodo contenente $e$.

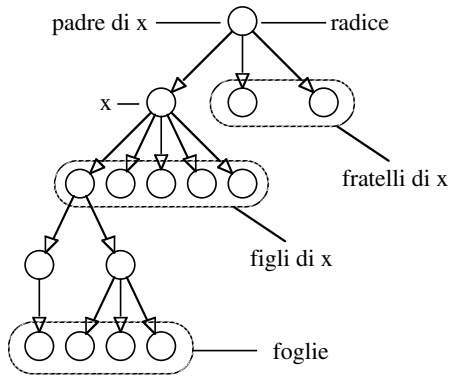
L'implementazione tramite visita in profondità, sebbene possibile, risulterebbe inutilmente complicata. Questa infatti richiederebbe di effettuare una visita esaustiva dell'albero, memorizzando tutti i nodi contenti $e$ e la relativa profondità (anch'essa da calcolare), a cui far seguire la selezione del nodo di profondità minima.

# 13. Alberi n-ari e grafi

In questo capitolo viene generalizzata la struttura *albero*, presentata nel capitolo precedente, inizialmente considerando alberi generici, ossia alberi nei quali ogni nodo può avere un numero arbitrario di figli. Successivamente, viene eliminato il vincolo che ogni nodo debba avere uno ed un solo nodo padre, ottenendo in questo modo una struttura dati denominata *grafo*. Di entrambe queste strutture dati vengono descritte diverse possibili implementazioni.

## 13.1. Alberi n-ari

Come visto nel capitolo precedente, l'albero n-ario è un tipo di dato non lineare utilizzato per memorizzare informazioni in modo gerarchico. Un esempio di albero è il seguente:




Naturalmente, la terminologia per denotare gli elementi dell'albero e le loro relazioni è la stessa del capitolo precedente.

### 13.1.1. Definizione induttiva di albero n-ario

Generalizziamo la definizione induttiva di albero binario:

- l'albero vuoto è un albero;
- se $T_1, \ldots, T_n$ $(n >= 1)$ sono alberi, allora l'albero che ha un nodo radice e $T_1, \ldots, T_n$ come sottoalberi è un albero;
- nient'altro è un albero.

I sottoalberi $T_1, \ldots, T_n$ $(n >= 1)$ di un albero non vuoto $T$ vengono detti **sottoalbero1** ... **sottoalberon** di $T$.

### 13.1.2. Tipo astratto AlberoN

Il tipo di dato astratto AlberoN può essere definito come segue.

**TipoAstratto AlberoN(T)**

**Domini**

**AlberoN** : dominio di interesse del tipo

**T** : dominio dei valori associati ai nodi dell'albero

**Funzioni**

**albVuoto() $\mapsto$ AlberoN**
pre: nessuna
post: RESULT è l'albero vuoto (ossia senza nodi)

**creaAlb(T r, AlberoN a1, ..., AlberoN an) $\mapsto$ AlberoN**
pre: nessuna
post: RESULT è l'albero n-ario che ha r come radice, a1,..., an come sottoalberi

**estVuoto(AlberoN a) $\mapsto$ Boolean**
pre: nessuna
post: RESULT è true se a è vuoto, false altrimenti

**radice(AlberoN a) $\mapsto$ T**
pre: a non è vuoto
post: RESULT è il valore del nodo radice di a

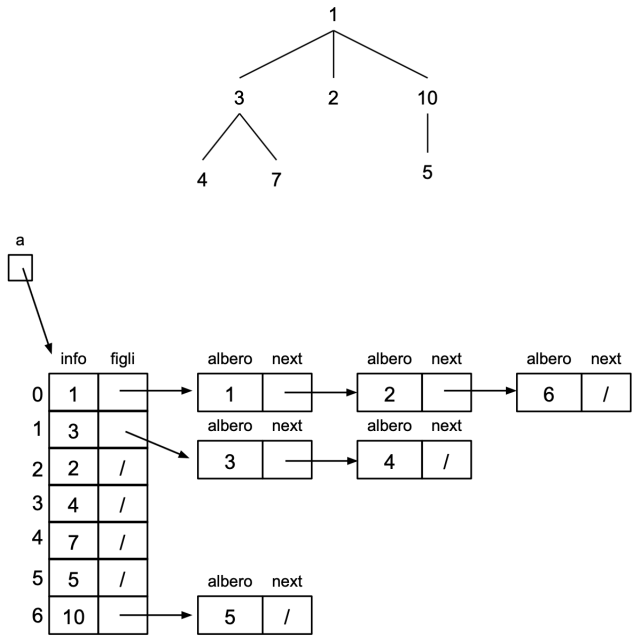
**figli(AlberoN a) $\mapsto$ Lista[AlberoN]**
pre: a non è vuoto
post: RESULT è una lista contenente i figli di a

**FineTipoAstratto**

### 13.1.3. Rappresentazione tramite lista dei successori

La rappresentazione indicizzata mediante array si generalizza al caso degli alberi n-ari. Tuttavia, a causa del fatto che la struttura non è regolare, nel senso che il numero dei figli di un nodo è arbitrario, occorre memorizzare per ogni nodo dell'albero in quale posizione dell'array vengono memorizzati i nodi figli. Questa rappresentazione prende quindi il nome di rappresentazione tramite *lista di successori*.

Il seguente albero n-ario può essere rappresentato tramite lista dei successori come illustrato




#### 13.1.3.1. Caratteristiche della rappresentazione

Si noti che:

- Gli elementi dell'array dovranno contenere un campo per memorizzare l'informazione associata al nodo ed un campo per memorizzare il puntatore ad una lista dei figli.
- Normalmente la radice si fa corrispondere al nodo in posizione 0, altrimenti si può indicare esplicitamente la posizione del nodo radice.

- I nodi foglia sono caratterizzati da una lista di successori vuota.
- L'ordine in cui vengono memorizzati i nodi nell'array non è in genere significativo, anche se si può forzare l'ordinamento in modo che la scansione sequenziale dell'array corrisponda alla visita per livelli.

### 13.1.4. Algoritmi di visita

Per visitare un albero n-ario occorre considerare che per ogni nodo occorre fare tante chiamate ricorsive quanti sono i figli del nodo.

```
if T non è vuoto {
     analizza il nodo radice di T
     while (primofiglio(T) !=NULL) {
            visita primofiglio(T)
            passa al figlio successivo
     }
}
```

- La visita in post-ordine richiede di effettuare l'analisi del nodo radice dopo l'esecuzione del ciclo.
- La visita simmetrica non è definita.
- La visita per livelli (in ampiezza) richiede l'utilizzo di una coda.

Per l'implementazione dell'albero n-ario tramite lista dei successori si rimanda alla sezione successiva sui grafi.

### 13.1.5. Rappresentazione collegata di alberi n-ari

Gli alberi n-ari possono essere rappresentati anche mediante la rappresentazione collegata. Tuttavia, a differenza del caso degli alberi binari, non si può definire a priori la dimensione del record, dato che il numero dei figli di un nodo non è definita a priori. Pertanto si utilizza una ulteriore struttura collegata per la memorizzazione dei figli.

La struttura del record per la memorizzazione di un nodo diventa semplicemente:

1. informazione associata al nodo;

2. riferimento ad una lista dei figli.

È possibile astrarre questo concetto in C definendo un tipo TipoAbero che è un puntatore ad un nodo di un albero:

```
typedef int TipoInfoAlbero;
struct StructNodoFiglio;
struct StructAlbero {
  TipoInfoAlbero info;
  struct StructNodoFiglio* figli;
};
struct StructNodoFiglio {
  struct StructAlbero* albero;
  struct StructNodoFiglio* next;
};
typedef struct StructAlbero* TipoAlbero;
typedef struct StructNodoFiglio* TipoFigli;
```

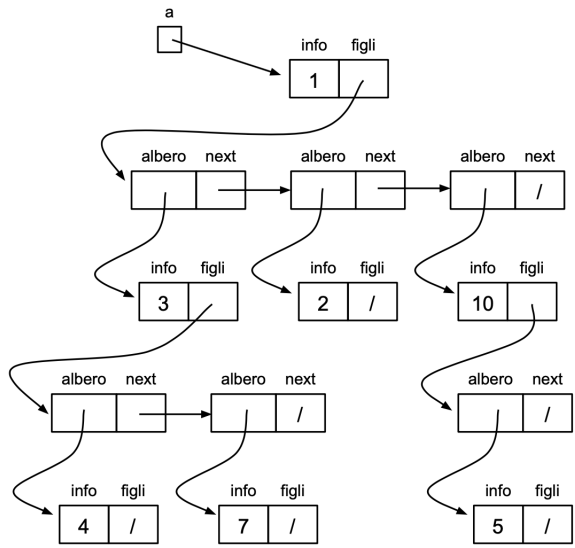
Osserviamo che:

- il campo info memorizza il contenuto informativo del nodo. Il tipo TipoInfoAlbero verrà definito di volta in volta a seconda della natura delle informazioni che si vogliono memorizzare nei nodi dell'albero, analogamente a quanto visto per le strutture collegate lineari.
- il campi figli contiene un riferimento ad una lista dei figli del nodo. Il valore NULL di questo campo indica che si tratta di un nodo foglia.

Un albero n-ario nella rappresentazione collegata sarà rappresentato quindi nel seguente modo:

- L'albero *vuoto* viene rappresentato usando il valore NULL.
- Un albero *non vuoto* viene rappresentato mantenendo il riferimento al nodo radice dell'albero. Osserviamo che partendo dal riferimento alla radice dell'albero è possibile utilizzare i collegamenti contenuti nella lista dei figli per raggiungere ogni altro nodo dell'albero. In base a questa assunzione, *ogni valore di tipo puntatore a TipoNodoAlbero consente di accedere a tutto il sottoalbero alla cui radice esso punta*.

*Esempio:* L'albero n-ario mostrato in precedenza può essere rappresentato in C in modo collegato come illustrato.




La variabile a rappresenta quindi l'albero e contiene un riferimento alla radice dell'albero.

#### 13.1.5.1. Implementazione del tipo astratto AlberoN

Una realizzazione del tipo astratto AlberoN usando la rappresentazione collegata è mostrata di seguito. Il tipo astratto AlberoN viene implementato mediante la struttura C TipoAlbero, il tipo degli elementi T è definito in C con il tipo TipoInfoAlbero.

Costruttori e predicati sono analoghi al caso dell'albero binario.

```
TipoAlbero alberoVuoto() {
  return NULL;
}

TipoAlbero creaAlbero(TipoInfoAlbero infoRadice,
          TipoFigli f) {
  TipoAlbero a = (TipoAlbero) malloc(sizeof(struct
    StructAlbero));
  a->info = infoRadice;
  a->figli = f;
  return a;
}
```

```
bool estVuoto(TipoAlbero a) {
  return (a == NULL);
}
```

Per quanto riguarda i selettori, la selezione dell'informazione nel nodo radice è analoga al caso dell'albero binario.

```
TipoInfoAlbero radice(TipoAlbero a) {
  if (a == NULL) {
    printf ("ERRORE accesso albero vuoto \n");
    return ERRORE_InfoAlbBin;
  } else {
    return a->info;
  }
}
```

La scelta di rappresentare i figli tramite una lista, richiede invece una opportuna funzione che selezioni dal nodo la lista dei figli. Una volta acquisito il puntatore a questa lista si possono utilizzare le funzioni sulle liste per estrarre il primo figlio `primofiglio`, e, più in generale gestire gli altri elementi della lista.

```
TipoFigli figli(TipoAlbero a) {
  if (a == NULL) {
    printf ("ERRORE accesso albero vuoto \n");
    return NULL;
  } else {
    return a->figli;
  }
}
```

Una volta realizzata la rappresentazione, la visita in pre-ordine dell'albero n-ario segue la formulazione generale dello schema di visita:

```
void stampaPreordine(TipoAlbero a) {
    if (a!=NULL) {
        printf(" %d ",a->info);
        TipoFigli p = a->figli;
        while (p!=NULL) {
            stampaPreordine(p->albero);
            p = p->next;
        }
    }
}
```

### 13.1.6. Rappresentazione parentetica di alberi n-ari

La **rappresentazione parentetica**, basata su stringhe di caratteri, si generalizza facilmente da quella dell'albero n-ario. Assumiamo che $f(T)$ sia la stringa che denota l'albero $T$; e che

1. l'albero vuoto è rappresentato dalla stringa `"()"`, cioè $f(\text{albero vuoto})$=`"()"`;
2. un albero con radice $x$ e sottoalberi $T_1 \ldots T_n$ è rappresentato dalla stringa `"( "` $+ x +$ `f`$(T_1) +\ldots+$ `f`$(T_n) +$ `"() )"`.

*Esempio:* L'albero mostrato nella seguente figura può essere rappresentato mediante la stringa:

```
"( 15 ( 3 ( -5 ( 0 () ) () ) "  +
         "( 2 () ) "  +
         "( 10 () ) "  +
         "() ) "  +
     "( 1 ( 12 () ) ( 5 () ) () ) "  +
     "( 8 () ) "  +
     "() )".
```

Le differenze con la rappresentazione parentetica degli alberi binari sono:

- il numero di figli dei nodi è variabile ed essi sono rappresentati dalla concatenazione delle rispettive rappresentazioni;
- viene inserito `"()"` per indicare che non vi sono altri figli in un nodo;
- la foglia dell'albero non richiede la presenza di due sottoalberi vuoti, ma semplicemente `"()"`, che indica l'assenza di figli.

### 13.1.7. Costruzione di un albero n-ario da rappresentazione parentetica

La funzione per costruire un albero n-ario data una sua rappresentazione parentetica segue lo schema iniziale di lettura da file della funzione LeggiAlbero vista per gli alberi binari.

```
TipoAlbero LeggiAlbero(char *nome_file) {
  TipoAlbero result;
  FILE *file_albero;
  file_albero = fopen(nome_file, "r");
  result = LeggiSottoAlbero(file_albero);
  fclose(file_albero);
  return result;
}
```

La funzione ha come parametro in ingresso una stringa che rappresenta il nome del file che contiene l'albero e, oltre a gestire le operazioni di apertura e chiusura chiama la funzione ausiliaria LeggiSottoAlbero, che restituisce un puntatore al sottoalbero letto.

```
TipoAlbero LeggiSottoAlbero(FILE *file_albero) {
  char c;
  TipoInfoAlbero i;
  TipoAlbero r;
  LeggiParentesi(file_albero);   // parentesi aperta
  c = fgetc(file_albero);   // carattere successivo
  if (c == ')')
    return NULL;   // se () l'albero e' vuoto
  else { // altrimenti legge la radice
    fscanf(file_albero, "%d", &i);
    r = creaAlbero(i, LeggiFigli(file_albero));
    LeggiParentesi(file_albero); // parentesi chiusa
    return r;
  }
}
```

Le convenzioni per la lettura dell'albero sono analoghe a quelle viste per gli alberi binari. Si riportano di seguito per completezza:

- quando l'albero è vuoto, il carattere che segue la parentesi aperta '(' deve necessariamente essere la parentesi chiusa ')' (senza spazi);
- al contrario quando il nodo non è l'albero vuoto, l'informazione di tipo carattere da associare al nodo deve essere preceduta da uno spazio ' '.

- Se le informazioni sono di tipo int, o altro tipo primitivo numerico il tipo della variabile usata per leggere l'informazione ed il formato della istruzione fscanf devono essere aggiornate coerentemente;

- se le informazioni sono di altro tipo, in particolare di tipo non primitivo per leggere l'informazione associata diventa necessario progettare una specifica funzione.

Seguono le funzioni ausiliarie per la lettura delle parentesi e per la lettura della lista dei figli.

```
void LeggiParentesi(FILE *file_albero) {
  char c = ' ';
  while (c!='(' && c!=')')
    c = fgetc(file_albero);
}

TipoFigli LeggiFigli(FILE *file_albero) {
    TipoAlbero f = LeggiSottoAlbero(file_albero);
    if (f==NULL)
        return NULL;
    else {
        // printf("aggiungi figlio %d\n", f->info);
        return add(LeggiFigli(file_albero), f);
    }
}
```

Ed infine si riportano le funzioni ausiliarie per la costruzione della lista dei figli.

```
TipoFigli add(TipoFigli f, TipoAlbero a) {
    TipoFigli n = (TipoFigli)malloc(sizeof(struct
    StructNodoFiglio));
    n->albero = a;
    n->next = f;
    return n;

}

void aggiungiFiglio(TipoAlbero a, TipoAlbero f) {
  if (a == NULL) {
    printf ("ERRORE accesso albero vuoto \n");
  } else {
    a->figli = add(a->figli, f);
  }
}
```

### 13.1.8. Esercizi

La variante della funzione che calcola la profondità dell'albero binario è immediata:

```
/* calcola la profondita' di un albero n-ario */
int Profondita(TipoAlbero albero)
{
  if (albero == NULL)
    return -1;   // l'albero vuoto ha profondita' -1
  else {
    // calcola la profondita' dei sottoalberi
    int pmax=-1;
    int p;
    TipoFigli l = a->figli;
    while (l!=NULL) {
            p=Profondita(l->albero);
            if (p>pmax) pmax=p;
            l = l->next;
            }
  return pmax+1;
  }
}
```

Anche l'estensione della ricerca si ottiene sostituendo alle due chiamate corrispondenti a sottoalbero sinistro e destro, la sequenza di chiamate ricorsive relative agli elementi della lista dei figli.

```
bool RicercaAlbero(TipoAlbero A, TipoInfoAlbero elem,
    TipoAlbero *posiz) {
  bool trovato = false; *posiz=NULL;
  if (A == NULL)
    trovato = false;
  else if (elem == A->info) { // elemento trovato
    *posiz = A;  trovato = true;
  }
  else {
    /* cerca nel sottoalbero sinistro */
    TipoFigli l = A->figli;
    while (p!=NULL and !trovato) {
        trovato = RicercaAlbero(l->albero, elem, posiz
    );
            l = l->next;
            }
  }
  return trovato;
}
```

Infine, anche per quanto riguarda la copia di un albero n-ario, l'estensione consiste nuovamente nel realizzare l'operazione su ciascun

elemento della lista di figli.

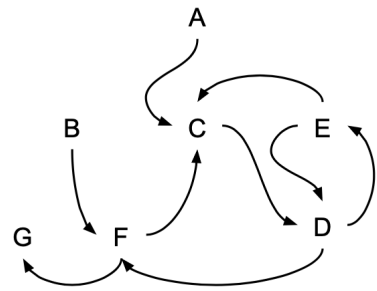
```
TipoFigli copiaFigli(TipoFigli f);

TipoAlbero copia(TipoAlbero a) {
    if (a==NULL)
        return NULL;
    else
        return creaAlbero(a->info,copiaFigli(a->figli)
    );
}

TipoFigli copiaFigli(TipoFigli f) {
    if (f==NULL)
        return NULL;
    else
        return add(copiaFigli(f->next),copia(f->albero
    ));
}
```

## 13.2. Grafi

Come anticipato, se rimuoviamo il vincolo che ogni nodo abbia uno ed un solo padre, la struttura dati che otteniamo prende il nome di *grafo*. In realtà, in molti testi si introduce prima il grafo, come struttura dati in grado di rappresentare *relazioni binarie* e successivamente si considera come caso particolare, l'albero n-ario.

In questa sezione, dopo una definizione della struttura dati grafo, faremo cenno ad alcune possibili implementazioni e rivisiteremo le nozioni di visita in profondità e visita in ampiezza nel caso del grafo.

Un esempio di grafo è il seguente:



#### 13.2.0.1. Definizione di grafo

Un grafo orientato è definito da una coppia di insiemi $\langle N, A \rangle$, in cui:

- $N$ è un insieme di nodi $N = \{n_0, \ldots, n_{s-1}\}$
- $A$ è un insieme di archi $N = \{a_1, \ldots, a_{m-1}\}$, in cui ciascun arco è a sua volta costituito da una coppia di nodi $\langle n_i, n_j \rangle$, che sta ad indicare che il nodo $n_i$ è collegato direttamente al nodo $n_j$.

Un grafo può essere visto coma la rappresentazione di una relazione $NxN$, con $N$ dominio finito.

#### 13.2.0.2. Un po' di nomenclatura

- *predecessore* è il primo nodo dell'arco $\langle n_i, n_j \rangle$;
- *successore* è il secondo nodo dell'arco $\langle n_i, n_j \rangle$;
- *sorgente* è un nodo che non ha archi entranti (e.g. la radice, ma in un grafo se ne possono avere più di una!);
- *pozzo* è un nodo che non ha archi uscenti (e.g. una foglia);
- *cammino* tra due nodi di un grafo è una sequenza di nodi $\{n_0, \ldots, n_k\}$, tale che per ogni coppia di elementi consecutivi della sequenza $\langle n_i, n_j \rangle$ vi è un arco nel grafo;
- *ciclo* è un cammino che contiene più volte lo stesso nodo del grafo.

La nozione di grafo si può generalizzare in varie maniere, considerando ad esempio che si può associare un peso agli archi o ammettendo più di un arco tra due nodi, ma questi argomenti esulano dagli scopi di questa dispensa, il cui scopo è quello di utilizzare delle rappresentazioni del grafo per lo svolgimento di esercizi relativi all'uso dei meccanismi di costruzione delle strutture dati resi disponibili dal linguaggio C.

#### 13.2.0.3. Rappresentazione del grafo tramite matrice delle adiacenze

Questa rappresentazione fa uso di una matrice di dimensione $NxN$, i cui elementi sono definiti come segue:

- $[i, j] = 1$, se nel grafo è presente l'arco $\langle n_i, n_j \rangle$;
- $[i, j] = 0$, altrimenti

Dopo aver associato a ciascun nodo del grafo un indice da 0 ad $n-1$ (dove $n$ è il numero di nodi nel grafo), ad esempio usando un array, il grafo mostrato in precedenza può essere rappresentato mediante matrice delle adiacenze come segue:

Associazione nodo-indice:

| nodo | indice |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |

Matrice delle adiacenze (le occorrenze del valore 0 non sono riportate per leggibilità):

| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | | | 1 | | | | |
| 1 | | | | | | 1 | |
| 2 | | | | 1 | | | |
| 3 | | | | | 1 | 1 | |
| 4 | | | 1 | 1 | | | |
| 5 | | | 1 | | | | 1 |
| 6 | | | | | | | |

L'implementazione di questa struttura si ottiene facilmente

```
#define NumeroNodi ...;
typedef int TipoInfoNodo;

struct Grafo{
      TipoInfoGrafo valori_nodi[NumeroNodi];
      int mat_adiacenza[NumeroNodi][NumeroNodi];
      };

typedef struct Grafo* TipoGrafo;
```

Si noti che la definizione della componente array `valori_nodi` non è necessaria, se l'informazione memorizzata nel nodo corrisponde all'etichetta numerica associata al nodo stesso.

Per esercizio si provi a verificare proprietà come:

- l'esistenza di nodi sorgente, nodi pozzo, presenza di nodi isolati (senza archi né ingresso né in uscita,
- esistenza di cammini tra coppie di nodi del grafo nodi del grafo,
- presenza di cicli etc. .

Questa rappresentazione offre una facile gestione delle operazioni fondamentali sulla struttura dati, ma ha l'inconveniente di richiedere molta memoria, che spesso viene usata solo in parte limitata. Infatti, dato che il numero degli archi è in genere paragonabile con il numero dei nodi la matrice delle adiacenze risulterà spesso *sparsa*, ossia avrà la maggioranza di valori pari a 0.

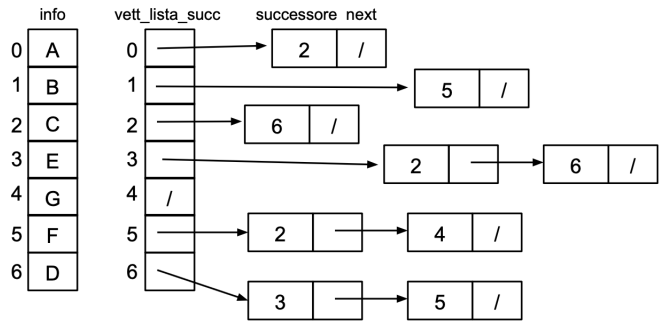
#### 13.2.0.4. Rappresentazione del grafo tramite liste dei successori

Questa rappresentazione è del tutto analoga a quella vista nel caso degli alberi n-ari.

Il grafo viene rappresentato da un array di nodi a ciascuno dei quali è associata:

- l'informazione memorizzata nel nodo;
- il riferimento ad una lista che contiene i puntatori ai nodi figli.

Il precedente grafo di esempio può essere rappresentato mediante liste di successori come di seguito rappresentato:

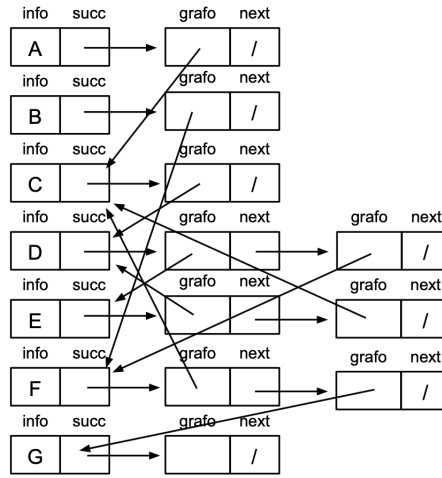



L'implementazione di questa struttura si ottiene creando una struttura collegata per la lista dei successori e definendo il grafo come array di puntatori a liste di successori. Anche in questo caso, se l'informazione da memorizzare nel nodo è rappresentabile direttamente tramite le etichette numeriche dei nodi non occorre definire l'array `info`.

```
#define NumeroNodi ...;
typedef int TipoInfoNodo;
struct ListaSucc {
  TipoNodo successore;
  struct ListaSucc *next;
};
typedef struct ListaSucc * TipoLS;
struct Grafo {
  TipoInfoNodo info [NumNodi];
  TipoLS vett_lista_succ[NumNodi];
};
```

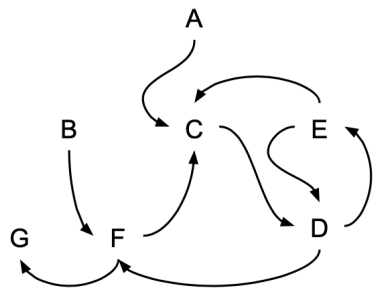
#### 13.2.0.5. Rappresentazione del grafo tramite strutture collegate

La rappresentazione tramite strutture collegate non presenta alcuna differenza, rispetto a quanto visto per gli alberi n-ari, in quanto la lista degli archi uscenti da un nodo corrisponde esattamente alla lista dei figli del nodo di un albero n-ario. La proprietà che nell'albero i nodi hanno un solo padre non vincola infatti in alcun modo la struttura definita.




#### 13.2.0.6. Tecniche di visita del grafo: in profondità

Consideriamo la visita in profondità del grafo mostrato in figura:




In questo caso, la presenza di un ciclo causa la non terminazione della visita, sia nel caso in cui questa venga realizzata tramite lo schema ricorsivo sia nel caso in cui essa sia realizzata usando la pila. Nel primo caso, la visita continuerà ad invocare se stessa a partire dallo stesso nodo, mentre nel caso della pila, il ciclo comporterà che lo stesso nodo rientrerà nella pilla un numero infinito di volte.

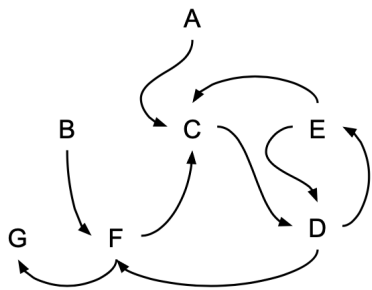
La tecnica per estendere i metodi di visita visti per gli alberi al caso del grafo consiste nel *marcare* i nodi durante la visita in modo che la visita prosegua per un nodo successore soltanto se esso non è già stato marcato (visitato in precedenza). In sintesi:

- si aggiunge all'informazione contenuta nel nodo un campo di tipo `bool`, che dovrà essere inizializzato a `false` prima dell'inizio della procedura di visita.

- prima di procedere con la visita in un nodo, si verifica se il suo campo marca è `false`: se non lo è si scarta e si passa al nodo successivo, se invece è `false` si visita il nodo, mettendo a `true` il suo campo marcatura.

#### 13.2.0.7. Tecniche di visita del grafo: ampiezza

Consideriamo la visita in ampiezza del grafo mostrato in figura:




anche nel caso della visita del grafo in ampiezza si presenta il problema di gestire l'eventuale presenza di cicli. Ed, anche in questo caso, si utilizza la marcatura dei nodi.