SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Matematica

Insegnamento di Informatica generale

Programmazione in C

Canale 2: Prof. Giancarlo Bongiovanni, Dott. Simone Silvestri

Questi appunti rispecchiano piuttosto fedelmente il livello di dettaglio che viene seguito durante le lezioni sul C, e costituiscono un ausilio didattico allo studio ed allo sviluppo dei programmi.

Tuttavia, è importante chiarire che gli appunti non vanno intesi come sostitutivi né della frequenza alle lezioni né dell'esercizio in prima persona nell'attività di programmazione, che rimangono fattori fondamentali per una buona preparazione dell'esame.

E' opportuno disporre, per qualunque necessità di approfondimento, di un manuale del linguaggio C (quale ad esempio Kernighan & Ritchie, The C Programming Language, Second Edition) o di un testo di riferimento per il C (quale ad esempio H.M. Deitel, P.J. Deitel, Corso completo di programmazione C, Apogeo).

Indice

1) Chiarezza dei programmi

Prima di qualunque altra considerazione è importante sottolineare che un aspetto fondamentale della programmazione, indipendentemente dal linguaggio utilizzato, è la chiarezza del programma che viene scritto.

Questo per varie ed importanti ragioni:

1. per poterne valutare la correttezza è necessario comprendere a fondo la struttura di un programma: tale requisito è molto facilitato se il programma è organizzato in modo chiaro, diventa estremamente difficile se non impossibile in caso contrario;
2. un programma può essere ripreso, a distanza di tempo, dall'autore stesso o da altre persone come base di partenza per svilupparne un altro; se esso è stato organizzato in modo chiaro questa attività può essere portata avanti in modo efficace, altrimenti no.

Vi sono due strumenti a disposizione del programmatore per migliorare la chiarezza di un programma:

- uso dell'indentazione;
- uso dei commenti.

1.1 Indentazione

L'indentazione (ossia la distanza dell'inizio di ogni riga del programma dal bordo sinistro) è una tecnica molto usata per fornire indicazioni sulla struttura di un programma:

- le istruzioni che appartengono allo stesso *blocco* (un blocco di istruzioni è una *sequenza di istruzioni racchiuse fra due parentesi graffe*) devono avere tutte la stessa indentazione;
- un blocco contenuto dentro un altro blocco deve avere una indentazione maggiore di quella del blocco che lo contiene.

In alcuni linguaggi di programmazione (ad esempio Python) e nello pseudocodice usato a lezione l'indentazione fa parte della sintassi del linguaggio, e quindi una sua errata gestione altera il funzionamento del programma.

In altri linguaggi non ne fa parte, come è il caso del C nel quale un blocco di istruzioni viene delimitato in modo esplicito mediante una coppia di parentesi graffe, ma rimane comunque uno strumento indispensabile per garantire la chiarezza del programma.

Ad esempio, è abbastanza facile comprendere che questo frammento di codice C calcola la somma dei valori positivi e quella dei valori negativi contenuti in una matrice quadrata:

```
for (i = 0; i < SIZE; i++) {
   for (j = 0; i < SIZE; j++) {
      if (M[i,j] < 0) {
         negativi = negativi + M[i,j];
      }
      if (M[i,j] > 0) {
         positivi = positivi + M[i,j];
      }
   }
}
```

Lo stesso frammento, senza indentazione, è più difficile da comprendere:

```
for (i = 0; i < SIZE; i++) {
for (j = 0; i < SIZE; j++) {
if (M[i,j] < 0) {
negativi = negativi + M[i,j];}
if (M[i,j] > 0){
positivi = positivi + M[i,j]}; }
}
}
```

Se poi si scrivono più istruzioni sulla stessa riga (il che è possibile in C poiché **ogni istruzione è terminata con un punto e virgola**) le cose diventano incomprensibili:

```
for (i = 0; i < SIZE; i++) { for (j = 0; i < SIZE; j++) {
if (M[i,j] < 0) {negativi = negativi + M[i,j];} if (M[i,j] > 0){
positivi = positivi + M[i,j]}; }}}
```

ATTENZIONE

In occasione degli esoneri e delle prove scritte i programmi C non indentati saranno considerati non correggibili e <u>pertanto non daranno alcun punteggio</u>.

1.2 Commenti

I commenti sono porzioni di testo che vengono ignorate durante le fasi di trasformazione del programma (compilazione e linking, che vedremo fra breve) necessarie alla creazione del programma eseguibile.

In C esistono due tipi di commenti:

- commenti su una singola riga;
- commenti su più righe.

I primi si inseriscono inziandoli con la doppia barra **//**, i secondi iniziano con la sequenza di caratteri **/*** e terminano con la sequenza di caratteri ***/**

```
/* ------------------------------------------------------------------------
Questo frammento di codice calcola due somme. La prima è la somma di tutti
gli elementi negativi della matrice M, la seconda è la somma di tutti gli
elementi positivi della matrice M
------------------------------------------------------------------------ */
for (i = 0; i < SIZE; i++) {           //ciclo sulle righe della matrice
   for (j = 0; i < SIZE; j++) {  //ciclo sulle colonne della matrice
      if (M[i,j] < 0) {    //se l'elemento è negativo
         negativi = negativi + M[i,j]; //aggiorna somma negativi
      }
      if (M[i,j] > 0) {    //se l'elemento è positivo
         positivi = positivi + M[i,j]; //aggiorna somma positivi
      }
   }
}
```

2) Compilazione ed esecuzione di programmi C

Un programma scritto in linguaggio C non può essere direttamente eseguito su un elaboratore.

Deve prima essere opportunamente trasformato in un *programma eseguibile*, scritto nel linguaggio macchina (ossia, un linguaggio le cui istruzioni sono direttamente eseguibili dal processore) dell'eleboratore su cui deve girare.

Una volta creato il programma eseguibile, esso può essere mandato in esecuzione.

La trasformazione consiste di due passi separati:

1. compilazione;
2. linking (collegamento).

2.1 Compilazione

Nei casi più semplici, come sono tutti quelli che ci interessano nel contesto di questo corso, entrambe queste operazioni si eseguono per mezzo di un singolo strumento, il compilatore C.

Sugli elaboratori del laboratorio di calcolo di Matematica il compilatore C si invoca dalla finestra del terminale (shell di comando), mediante il comando `gcc`.

Ad esempio, dato il seguente programma:

```
#include <stdio.h>

int main() {
   printf("hello, world\n");
   return 0;
}
```

contenuto in un file di testo di nome `helloworld.c`, posizioniamoci con la shell sul direttorio che contiene tale file ed impartiamo il seguente comando:

```
$ gcc helloworld.c
```

Dopo l'esecuzione di tale comando, nel direttorio contenente il file `helloworld.c` sarà presente un nuovo file di nome a.out che può essere mandato in esecuzione mediante il comando:

```
$ ./a.out
```

Nota: la sequenza di caratteri `./` che precede a.out significa "nel direttorio corrente" e su alcuni sistemi Unix è necessaria, su altri no. Per sicurezza è bene usarla sempre quando si lavora su Unix, Linux o Mac OS X.

Il comando gcc consente di specificare anche dei parametri, uno dei quali permette di assegnare uno specifico nome al programma eseguibile risultante dalla compilazione.

Ad esempio, il comando:

```
$ gcc -o helloworld.out helloworld.c
```

produce un file eseguibile di nome `helloworld.out` che può essere mandato in esecuzione mediante il comando:

```
$ ./helloworld.out
```

L'esecuzione del programma produce la scritta "hello, world" sullo schermo:

```
$ ./helloworld.out
hello, world
$
```

2.2 Linking

La trasformazione del programma C in programma eseguibile non consiste solo nella compilazione, ma anche in una successiva fase di "collegamento" (linking) con una o più *librerie*, ossia collezioni di funzionalità standard "preconfezionate" a disposizione dei programmi che ne necessitano.

Una serie di librerie vengono utilizzate automaticamente senza che il programmatore se ne debba preoccupare, mentre per altre librerie si deve esplicitamente richiamarne la necessità nel programma C, mediante opportune direttive come quella presente nell'esempio 1:

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

Questa direttiva istruisce:

- il compilatore (più precisamente il preprocessore C) ad utilizzare, in fase di compilazione, le definizioni delle funzioni presenti nella libreria `stdio.h` (scritta in C);
- il linker ad utilizzare la corrispondente libreria eseguibile in fase di linking.

L'effetto materiale delle direttive di tipo `include` è che il preprocessore C inserisce nel testo del programma C, nel punto in cui è presente la direttiva, il testo contenuto nel file il cui nome appare fra i simboli `<` e `>`. Nell'esempio specifico, il testo contenuto nel file `stdio.h`.

Il compilatore quindi compila il file prodotto dal preprocessore C, che a quel punto contiene sia il codice contenuto nel file che è stato incluso sia quello contenuto nel file che è stato passato al compilatore nella linea di comando .

Altre librerie importanti sono `stdlib` (necessaria ad esempio per poter allocare dinamicamente la memoria) e `math` (necessaria per utilizzare varie funzioni matematiche quali, ad es., la radice quadrata). Se servono vanno incluse con la direttiva `include`. Per compilare programmi che includono la libreria `math` si deve aggiungere nella riga di comando l'opzione `-lm`.

La direttiva `include`, inoltre, permette al programmatore di definire una volta sola del codice (di solito opportune strutture dati e funzioni) da utilizzare poi in diversi programmi, salvarlo in un file con estensione `.h` e includerlo con questo tipo di direttiva nei programmi C che necessitano di tali strutture dati e funzioni.

3) Nozioni fondamentali per la programmazione in C

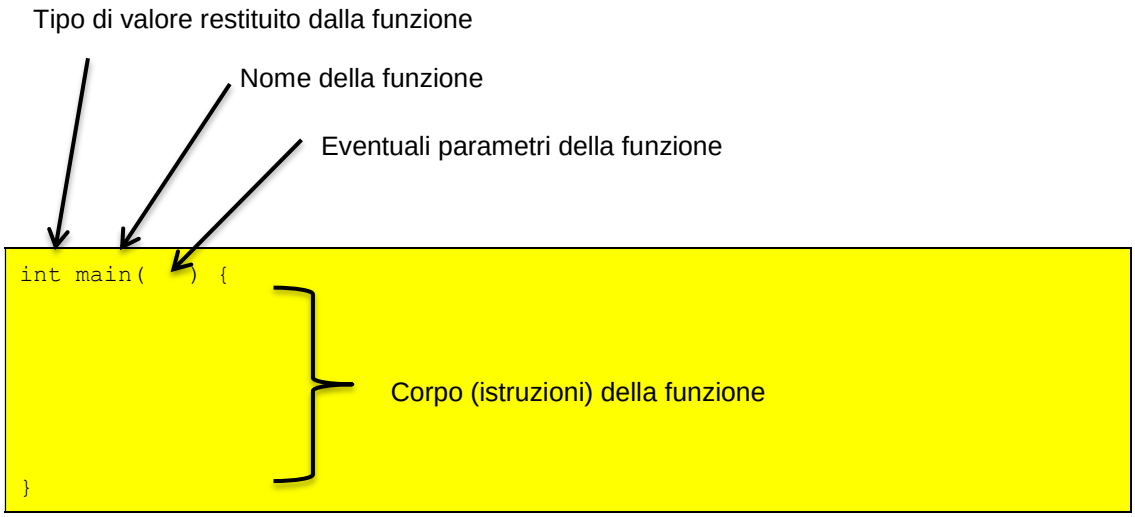
3.1 La funzione `main()`

Un programma C consiste di una o più funzioni.

Una di queste funzioni riveste un ruolo speciale, ed è la funzione `main()`.

La funzione `main()` deve essere obbligatoriamente presente e viene eseguita quando si manda in esecuzione il programma.

La funzione `main()` (ciò vale per tutte le funzioni C) ha la seguente sintassi:



La funzione `main()` a sua volta può chiamare altre funzioni, definite nel programma C stesso oppure nelle librerie incluse nel programma C. Approfondiremo questi aspetti quando parleremo più dettagliatamente delle funzioni.

3.2 Variabili

Come in tutti i linguaggi di programmazione, in C è possibile (e necessario) utilizzare delle *variabili* per poter memorizzare i valori risultanti via via dalla computazione.

Ogni variabile ha un nome univoco, che la distingue dalle altre variabili. Il nome:

- deve iniziare con una lettera;
- può essere composto da lettere, numeri ed altri caratteri ammessi (ad es. _);
- è case-sensitive (`Alfa` è un nome diverso da `alfa`).

Ogni variabile ha un tipo, che ne caratterizza la natura. Esso può essere un tipo “primitivo” come quelli elencati nel seguito oppure un tipo più complesso definito dall’utente. Tipi primitivi sono ad esempio:

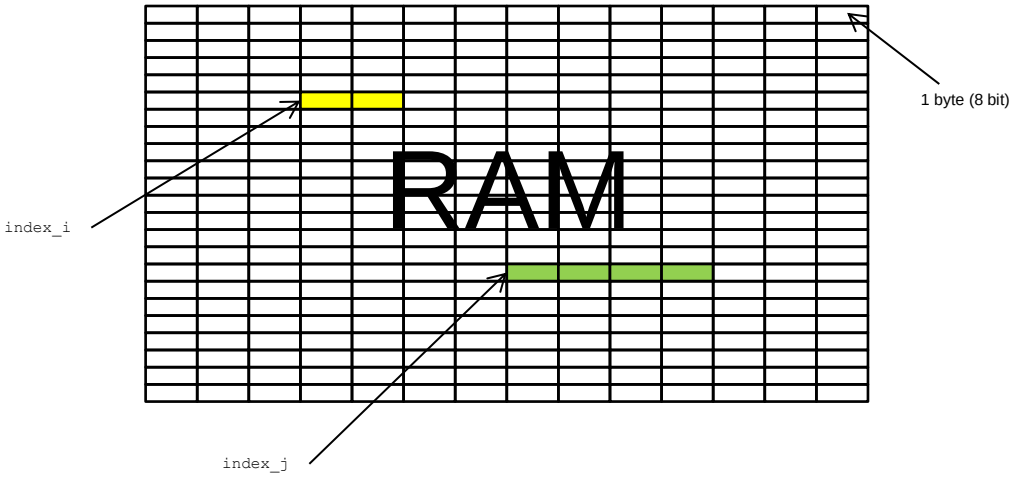
- `int`: per memorizzare numeri interi, di norma rappresentati con 16 bit (valori da -32768 a 32767 inclusi); va detto che in quasi tutte le attuali implementazioni ormai anche in questo caso si usano 32 bit;
- `long`: per memorizzare numeri interi di modulo più grande (di norma rappresentati con 32 bit, valori da -2,147,483,648 a 2,147,483,647 inclusi);
- `float`: per memorizzare numeri decimali in virgola mobile; di norma sono a 32 bit, di cui 8 per l’esponente e 24 per la mantissa e permettono di rappresentare valori da $1.175494351*10^{-38}$ a $3.402823466*10^{38}$.
- `double`: per memorizzare numeri decimali con precisione più grande (di norma rappresentati con 64 bit, valori da $2.2250738585072014 *10^{-308}$ a $1.7976931348623158*10^{308}$);
- `char`: per memorizzare un singolo carattere;
- ecc.

Ogni variabile deve essere dichiarata prima di poter essere utilizzata:

- una variabile si dichiara scrivento prima il tipo e poi il nome, ad es.

```
int index_i;
long index_j;
```

- la dichiarazione di una variabile produce l’allocazione in memoria di una quantità di celle di memoria sufficienti a contenere un valore del tipo dichiarato per la variabile (ad es.: due byte, ossia 16 bit, per una variabile di tipo `int`);
- attraverso il nome della variabile il programma accede a tale porzione di memoria per leggere il valore della variabile o assegnarle un nuovo valore.



Ogni variabile può venire inizializzata (cioè si può assegnarle un valore iniziale) già al momento della dichiarazione.

ATTENZIONE

Se nel programma si utilizza i valore di una variabile non inizializzata i risultati della computazione divengono imprevedibili.

Nel seguente programma:

```
#include <stdio.h>

int main() {
   int i = 100;
   int j;

   ...
}
```

la variabile intera `i` è dichiarata e contestualmente inizializzata al valore 100, mentre la variabile intera `j` è dichiarata ma non inizializzata. Se il programma stampasse i loro valori:

- per `i` si otterrebbe il corretto valore 100;
- per `j` si avrebbe un valore imprevedibile poiché il programma interpreterebbe come numero intero il valore contenuto in quel momento nella porzione di memoria assegnata a `j` e stamperebbe tale valore, che ovviamente non ha nulla a che vedere con l'esecuzione del programma in quanto dipende dall'uso che altri programmi hanno fatto di quella stessa porzione di memoria.

Ogni variabile ha un ambito di visibilità (detto *scope*), ossia un ambito nel quale essa è definita ed utilizzabile e al di fuori del quale non è definita né utilizzabile:

- una variabile definita dentro una funzione ha come ambito di visibilità tutta la successiva parte del corpo della funzione (variabile locale alla funzione); si consiglia di dichiarare tutte le variabili necessarie all'inizio della funzione (subito dopo la parentesi graffa che segna l'inizio del corpo della funzione);
- una variabile definita fuori da ogni funzione ha come ambito di visibilità tutte le funzioni la cui definizione segue il punto in cui la variabile è dichiarata (variabile globale); si sconsiglia l'utilizzo di tale pratica, che facilita i cosidetti *side-effect*;
- assegnare lo stesso nome a due variabili diverse è una pratica programmativa da evitare salvo nel caso in cui i relativi scopi di validità siano disgiunti (ad es. le due variabili sono definite in due funzioni diverse che non si chiamano a vicenda).

3.3 Assegnamento

L'operatore di assegnamento è l'operatore `=` (simbolo di uguale):

Esempi:

- `i = 100;` scrive il valore intero 100 nella porzione di memoria assegnata ad `i`;
- `a = b;` copia nella porzione di memoria assegnata ad `a` il valore presente nella porzione di memoria assegnata a `b`;

L'assegnamento può essere effettuato solo fra tipi di dati uguali o compatibili (in particolare, un valore di tipo `int` può essere assegnato a una variabile di tipo `float` o `double`).

3.4 Operatori aritmetici

Gli operatori aritmetici sono i seguenti.

+	Somma
−	Sottrazione
*	Moltiplicazione
/	Divisione
%	Modulo (resto della divisione intera)

Esempi:

```
a = 3*5;
b = c+d*50;
x = alfa%2;
```

Il valore del risultato: dipende sia dall'espressione che viene calcolata sia dal tipo della variabile che riceve il risultato. Alcuni casi sono riportati nella tabella seguente.

Istruzione	Valore di a risultante dopo l'assegnamento	Commento
`int a = 7/3;`	2	L'espressione 7/3 è una divisione fra interi che produce un valore intero
`int a = 7.0/3.0;`	2	L'espressione 7.0/3.0 è una divisione fra decimali che produce un valore decimale, che però diviene un valore intero (troncato) nel momento in cui viene assegnato a una variabile intera
`float a = 7/3;`	2,0	L'espressione 7/3 è una divisione fra interi che produce un valore intero, che però diviene un valore decimale (con zeri dopo la virgola) nel momento in cui viene assegnato a una variabile float
`float a = 7.0/3;` `float a = 7/3.0;`	2,333… 2,333…	Un decimale diviso un intero e un intero diviso un decimale producono valori decimali
`int a = 7.0/3;` `int a = 7/3.0;`	2	Che però divengono valori interi (troncati) nel momento in cui vengono assegnati a una variabile intera

Si consiglia comunque di sperimentare di persona tali situazioni sul sistema in uso, poiché il comportamento è dipendente dall'implementazione.

Gli operatori aritmetici seguono le classiche regole di precedenza. Scrivere:

```
d = a*b+c;
```

ha lo stesso effetto che scrivere:

```
d = (a*b)+c;
```

3.5 Input/output

In C sono disponibili le due funzioni `scanf()` e `printf()` il cui scopo è rispettivamente:

- acquisire i dati da utilizzare nel programma (input);
- fornire i risultati prodotti dal programma (output).

Tali funzioni operano interagendo con la finestra del terminale:

- `scanf()` acquisisce i dati inseriti nella finestra del terminale tramite la tastiera dal momento in cui la funzione viene chiamata al momento in cui viene premuto il tasto *return* (da *carriage return*; in italiano viene detto *invio* o *ritorno carrello*);
- `printf()` fornisce i risultati scrivendoli sulla finestra del terminale.

Queste funzioni sono definite nella libreria `stdio.h`, che deve quindi essere inclusa con l'opportuna direttiva.

Esiste anche la possibilità di leggere dati da file e scriverli su file, come vedremo più avanti.

3.5.1 Input

La funzione `scanf()` (nome che deriva da *scan format*) è definita in modo da permettere di specificare sia quanti e quali dati debbano essere letti, sia il formato di ciascuno dei dati che devono essere letti. Inoltre, essa restituisce il valore -1 se si è verificato un errore (noi ignoreremo questo aspetto).

La funzione si aspetta i seguenti parametri:

- una stringa di formato, che serve a specificare il tipo di ciascun dato che deve essere letto;
- l'indirizzo di memoria di ciascuna delle variabili alle quali vanno attribuiti i valori che vengono letti: i tipi specificati nella stringa di formato vengono messi ordinatamente in corrispondenza con le variabili di cui vengono forniti gli indirizzi.

Nel caso di valori numerici (gli unici che ci interessano nel contesto di questo corso):

- `scanf()` riceve tutti i caratteri inseriti nella finestra del terminale tramite la tastiera dal momento in cui la funzione viene chiamata al momento in cui viene premuto il tasto *return* (da *carriage return*; in italiano viene detto *invio* o *ritorno carrello*)
- li interpreta secondo quanto specificato dalla stringa di formato e li assegna ordinatamente alle variabili passate come parametri;
- il carattere prodotto dal tasto return viene scartato dalla funzione.

Vediamo come usare la funzione attraverso alcuni esempi.

```
scanf("%d", &index_i);
```

Il significato dell'operatore `&` che precede il nome della variabile `index_i` sarà chiarito quando si illustreranno i puntatori.

In questo esempio i parametri indicano che si deve leggere un singolo numero intero (la stringa di formato è `"%d"`) e che il valore immesso va assegnato alla variabile `index_i`.

Nella finestra del terminale va inserito un numero intero terminato dal return.

Questa istruzione cattura le cifre immesse da tastiera e le interpreta come un singolo numero intero il cui valore viene assegnato alla variabile di tipo intero `index_i`.

```
scanf("%d %d", &index_i, &index_j);
```

In questo esempio i parametri indicano che si devono leggere due numeri interi (`"%d %d"`), che tali valori devono essere separati da un singolo spazio (presente nella stringa di formato fra i due `%d`) e che i relativi valori immessi vanno assegnati ordinatamente alle variabili intere `index_i` e `index_j`.

Nella finestra del terminale vanno inseriti due numeri interi, separandoli con un singolo spazo, terminando il secondo (e l'intera operazione di lettura) mediante un return.

Questa istruzione cattura i caratteri immessi da tastiera fino al return (escluso), li interpreta come due numeri interi separati da uno spazio i cui valori vengono assegnati ordinatamente alle variabili di tipo intero `index_i` e `index_j`.

I valori più comuni da utilizzare nella stringa di formato sono i seguenti.

Tipo di dato	Tipo in C	Stringa di formato	Esempi di sequenze di caratteri da digitare sulla tastiera
Intero	`int`	`%d`	`2975` `-312`
Decimale	`float`	`%f`	`52.750` `-0.75`
Carattere	`char`	`%c`	`i` `z`
Stringa di caratteri	`char*`	`%s`	`buongiorno`

```
scanf("%f %d", &temperatura_media, &numero_misure);
```

In questo esempio i parametri indicano che si deve leggere per primo un numero decimale e per secondo un numero intero (`"%f %d"`), separati da un singolo spazio, e che i valori immessi vanno assegnati ordinatamente alle variabili `temperatura_media` e `numero_misure`, di cui la prima è `int` e la seconda è `float`.

Molta attenzione deve essere posta al momento dell'immissione di dati, che devono essere scritti in perfetta congruenza sia con i tipi di formati che la `scanf()` si aspetta di acquisire sia con la struttura complessiva della stringa di formato. Il separatore dei decimali è il punto e non la virgola. Una corretta immissione di dati per questo esempio è la seguente:

```
62.54 274
```

(nota: un singolo spazio separa i valori, tanto nella stringa di formato quanto nella sequenza immessa tramite tastiera). Premendo il return si esegue materialmente l'operazione di lettura.

Se non si rispettano i formati specificati la `scanf()` restituisce un errore (segnalato tramite il valore -1) e diventa difficile capire quali dati siano stati letti e quali no perché ciò dipende dall'implementazione.

3.5.2 Output

ATTENZIONE

Negli esempi riportati in questa dispensa sono spesso presenti stampe il cui scopo è guidare l'utente nell'immissione dati oppure fornire indicazioni sui valori forniti in output. Si tenga presente che in occasione della consegna degli homework l'output prodotto dai programmi deve essere TOTALMENTE CONFORME a quanto specificato negli homework stessi, per cui stampe di tale tipo NON DEVONO ESSERE PRESENTI a meno che non siano esplicitamente richieste.

La funzione `printf()` (nome che deriva da *print format*) è definita in modo da permettere di specificare:

- una stringa "esplicativa" di caratteri da stampare sulla finestra del terminale a corredo della stampa dei risultati;
- il formato di ciascuno dei dati che devono essere stampati sulla finestra del terminale;
- I dati che devono essere stampati sulla finestra del terminale.

Inoltre, essa restituisce il valore -1 se si è verificato un errore (ignoreremo questo aspetto).

La funzione si aspetta i seguenti parametri:

- una stringa di formato, che serve a specificare sia la stringa esplicativa sia il formato in cui ciascun dato deve essere stampato;
- le variabili delle quali si vogliono stampare i valori: i formati specificati nella stringa di formato (`%d`, `%f`, ecc. come visto per `scanf()`) vengono messi essere ordinatamente in corrispondenza con le variabili di cui si vogliono stampare i valori.

Vediamo come usare la funzione attraverso alcuni esempi.

```
printf("Il valore di index_i è: %d", index_i);
```

In questo esempio i parametri indicano che si deve stampare un singolo numero decimale (la stringa di formato contiene un solo `%d`) e che il valore da stampare è quello della variabile `index_i`. Inoltre, la stringa di formato specifica che si deve stampare la stringa esplicativa:

`"Il valore di index_i è: "`

(notare lo spazio dopo i due punti) prima della stampa del valore della variabile `index_i`. Il risultato è che sulla finestra del terminale appare, ad esempio:

```
Il valore di index_i è: 3271
```

e la prossima istruzione di stampa proseguirà sulla medesima linea della finestra del terminale.

Se si desidera che la successiva stampa avvenga su una nuova linea, si devono inserire alla fine della stringa di formato i caratteri `\n` (corrispondenti al carattere non stampabile *newline*):

```
printf("Il valore di index_i è: %d\n", index_i);
```

Il prossimo esempio stampa il valore di due variabili `a` e `b`, la prima intera e la seconda decimale, e poi va a capo:

```
printf("a = %d b = %f\n", a, b);
```

sulla finestra del terminale apparirà, ad esempio:

```
a = 102 b = 37.1
```

si noti che tutti gli spazi presenti nella stringa di formato vengono rispettati nella stampa.

3.5.3 Redirezione dell'input/output

E' possibile fare in modo che il programma C legga I dati da file e/o scriva I risultati su file, senza dover modificare il codice. In particolare, leggere i dati da un file anziché dalla finestra del terminale può essere estremamente utile quando il programma deve leggere una quantità rilevante di dati: si risparmia tempo e si evitano potenziali errori di immissione dati.

Ciò si ottiene con opportuni operatori dello shell che redirigono l'input e l'output dalla situazione standard (nella quale sia l'input che l'output sono collegati alla finestra del terminale) a quella voluta, di solito da e verso dei file:

- l'operatore < redirige l'input;
- l'operatore > redirige l'output.

Ad esempio, il seguente comando esegue il programma "prova.out" facendogli leggere i dati da un file di nome "numeri.txt" e stampando i risultati sulla finestra del terminale:

```
$ ./prova.out < numeri.txt
```

Si noti che tutte le chiamate di `printf()` contenute nel programma continuano a stampare nella finestra del terminale, anche quelle (superflue) pensate per indicare all'utente quali dati inserire: nel caso si intenda usare questa tecnica è bene commentare tali chiamate.

Il seguente comando acquisisce i valori dal file di nome "numeri.txt" e redirige le stampe sul file di nome "output.txt:

```
$ ./prova.out < numeri.txt > output.txt
$
```

Nulla viene visualizzato nella finestra del terminale. Anche in questo caso si deve ricordare che tutte le chiamate di `printf()` vengono redirette e scrivono sul file, anche quelle pensate per indicare all'utente quali dati inserire.

Infine, il seguente comando acquisisce i valori dalla finestra del terminale e redirige le stampe sul file di nome "output.txt:

```
$ prova.out > output.txt
```

Di nuovo, si ricordi che tutte le chiamate di `printf()` vengono redirette e scrivono sul file, anche quelle pensate per indicare all'utente quali dati inserire: quindi l'utente non avrà indicazioni sui dati da inserire.

3.5.4 Esempio 01: lettura e scrittura

Il programma seguente (esempio_01.c) legge due interi dalla finestra del terminale e stampa la somma dei due.

```
#include <stdio.h>

int main() {
   int i, j, k;

   printf("Inserisci il primo numero: ");
   scanf("%d", &i);
   printf("Inserisci il secondo numero: ");
   scanf("%d", &j);

   k = i + j;

   printf("La somma di i e j vale: %d\n", k);
   return 0;
}
```

Dopo la compilazione e l'esecuzione la finestra del terminale mostrerà, ad esempio:

```
$ gcc –o esempio_01.out esempio_01.c
$ ./esempio_01.out
Inserisci il primo numero: 21
Inserisci il secondo numero: 37
La somma di i e j vale: 58
$
```

4 La selezione

Il comando di selezione (istruzione `if`) permette di diversificare il flusso dell'esecuzione lungo due rami diversi a seconda del valore di verità di un'espressione logica.

4.1 Operatori relazionali e logici

Gli operatori relazionali sono i seguenti.

Relazione	`Operatore`	Esempio
E' uguale a	`==`	`a == b`
E' diverso da	`!=`	`a != b`
E' maggiore di	`>`	`a > b`
E' maggiore o uguale a	`>=`	`a >= b`
E' minore di	`<`	`a < b`
E' minore o uguale a	`<=`	`a <= b`

Il risultato dell'applicazione di un operatore relazionale a due operandi è un valore intero (in C non esiste il tipo booleano) pari a:

- uno se la relazione specificata dall'operatore, applicata ai due operandi, è soddisfatta;
- zero altrimenti.

Gli operatori logici, utili per costruire espressioni condizionali più complesse, sono i seguenti.

Operazione logica	`Operatore`	Esempio
Congiunzione (AND)	`&&`	`(a == b)&&(c == d)`
Disgiunzione (OR)	`||`	`(a == b)||(c == d)`
Negazione (NOT)	`!`	`(!a) && (b >= c)`

L'operatore NOT ha precedenza su AND e OR. Per stabilire precedenze fra AND e OR è necessario l'uso delle parentesi.

In C la valutazione di un'espressione condizionale complessa si arresta, senza essere completamente valutata, quando:

- le clausole più interne sono messe in AND e una di esse ha valore falso (il valore risultante è zero);
- le clausole più interne sono messe in OR e una di esse ha valore vero (il valore risultante è uno).

L'espressione logica è valutata scorrendola da sinistra verso destra, con ovvia precedenza alle clausole più interne.

4.2 Istruzione `if`

L'istruzione `if` in C ha la seguente sintassi.

```
if (condizione) {
   …
   …          blocco if: istruzioni eseguite se la condizione è vera
   …
} else {
   …
   …          blocco else: istruzioni eseguite se la condizione è falsa
   …
}
```

La `condizione` di norma è una espressione condizionale, costruita con gli operatori condizionali e logici visti sopra.

Esempi:

```
if (a > b) {
   c = a;
} else {
   c = b;
}
```

```
if (a > b) c = a;
else c = b;
```

```
if (a%2 == 0) && (b%2 == 0)
   tutti_pari = 1;
```

Note:

- se non sono necessari, l'`else` e il relativo blocco possono essere omessi;
- se un blocco (`if` oppure `else`) consiste di una singola istruzione, le relative parentesi graffe possono essere omesse;
- in C la condizione può essere anche una qualunque variabile: in tal caso se il valore della variabile è zero si esegue il blocco `else`, se è un qualunque valore diverso da zero si esegue il blocco `if`;
- può anche essere una assegnazione (ad es. `if (a = b*2) istruzione;`), nel qual caso la valutazione della condizione avviene sulla variabile a sinistra dell'uguale dopo aver eseguito l'assegnazione. Nota: questa pratica di programmazione è assolutamente sconsigliabile.

4.3 Esempi ed esercizi

Esempio 02: lettura, test e scrittura

Il programma seguente (esempio_02.c) legge due interi dalla finestra del terminale, verifica se sono uguali e stampa un messaggio conseguente.

```
#include <stdio.h>

int main() {
   int i, j, k;

   printf("Inserisci il primo numero: ");
   scanf("%d", &i);
   printf("Inserisci il secondo numero: ");
   scanf("%d", &j);

   if (i == j) printf("I due numeri sono uguali\n");
   else printf("I due numeri sono diversi\n");
   return 0;
}
```

Dopo la compilazione e l'esecuzione la finestra del terminale mostrerà, ad esempio:

```
$ gcc -o esempio_02.out esempio_02.c
$ ./esempio_02.out
Inserisci il primo numero: 25
Inserisci il secondo numero: 48
I due numeri sono diversi
$
```

Esercizio 01

Scrivere un programma che legga un numero intero e verifichi se il numero immesso è pari o dispari.

Esercizio 02

Scrivere un programma che legga due numeri interi e stampi:

- un opportuno messaggio se i numeri immessi sono uguali;
- il maggiore dei due nel caso siano diversi.

5 L'iterazione

I comandi (istruzioni) di iterazione permettono di definire sotto quali condizioni si deve ripetere l'esecuzione di una stessa porzione (blocco) del programma.

In C esistono diverse istruzioni per gestire l'iterazione:

- istruzione `for`;
- istruzione `while`;
- istruzione `do while`.

5.1 Istruzione `for`

L'istruzione `for` ha la seguente sintassi:

```
for (espressione_1; espressione_2; espressione_3) {
   …
   …
   …
}
```

Dove:

- `espressione_1` specifica il valore iniziale della variabile di scorrimento;
- `espressione_2` specifica la condizione che deve essere verificata per eseguire la prossima iterazione;
- `espressione_3` governa l'evoluzione dei valori della variabile di scorrimento.

Ad esempio, il seguente ciclo `for` viene eseguito 10 volte, una per ciascuno dei valori 0, 1, 2, …, 9:

```
for (i = 0; i < 10; i = i + 1) {
   …
   …
   …
}
```

Al posto di `i = i + 1` nella `condizione_3` è pratica comune utilizzare il *postincremento*: `i++`

NOTE:

- come nel caso dell'`if`, se il blocco di istruzioni da eseguire è costituito da una singola istruzione le parentesi graffe possono essere omesse;
- la variabile di scorrimento non deve necessariamente essere di tipo intero;

- l'incremento della variabile di scorrimento non è necessariamente positivo ed unitario: può essere una qualunque operazione di somma, sottrazione, moltiplicazione, divisione, e perfino una operazione non aritmetica (come ad esempio l'avanzamento di un puntatore se la variabile di scorrimento è di tale tipo); esso viene eseguito al termine dell'iterazione, subito prima di ritornare alla valutazione di `espressione_2` per la prossima iterazione.

5.2 Istruzione `while`

L'istruzione `while` ha la seguente sintassi:

```
while (condizione) {
   …
   …
   …
}
```

Dove:

- `condizione` è la espressione condizionale che viene valutata prima di eseguire il blocco di istruzioni: se essa è vera il blocco (e quindi l'iterazione) si esegue, altrimenti no;
- dopo l'esecuzione di ciascuna iterazione si ritorna automaticamente alla valutazione della condizione;
- il ciclo `while` termina non appena la condizione è falsa.

Si noti che l'iterazione del `while` potrebbe non essere eseguita nemmeno una volta (se la condizione è falsa fin dall'inizio) oppure potrebbe non terminare mai (se la condizione non diviene mai falsa).

Se il blocco di istruzioni da eseguire è costituito da una singola istruzione le parentesi graffe possono essere omesse.

Ad esempio, il seguente ciclo `while` viene eseguito finché la variabile `stop`, raddoppiata ad ogni iterazione, resta minore di 4096 (quindi si eseguono $\log_2 4096 = 12$ iterazioni):

```
stop = 1;
while (stop < 4096) {
   …
   …
   stop = stop*2;
}
```

5.3 Istruzione `do while`

L'istruzione `do while` ha la seguente sintassi:

```
do {
   …
   …
   …
} while (condizione);
```

Dove:

- `condizione` è la espressione condizionale che viene valutata dopo aver eseguito l'iterazione: se essa è vera la prossima iterazione si esegue, altrimenti no.

Si noti che l'iterazione del `do while` viene sempre eseguita almeno una volta (poiché la condizione si valuta dopo l'esecuzione dell'iterazione) e, come nel caso del while, potrebbe non terminare mai (se la condizione non diviene mai falsa).

Se il blocco di istruzioni da eseguire è costituito da una singola istruzione le parentesi graffe possono essere omesse.

Ad esempio, il seguente ciclo `do while` viene eseguito finché alla variabile `prosegui` non viene assegnato, dentro l'iterazione, il valore zero:

```
do {
   …
   if (…) prosegui = 0;
   …
} while (prosegui);
```

5.4 Istruzione `break` e `continue`

Le istruzioni `break` e `continue` servono entrambe per interrompere l'esecuzione dell'iterazione corrente, ma operano in due modi diversi:

- l'istruzione `break` causa l'immediata interruzione dell'esecuzione della iterazione corrente e l'uscita dall'intero costrutto iterativo (for, while, do while) nel quale viene eseguita;

- l'istruzione `continue` causa l'immediata interruzione dell'esecuzione della iterazione corrente ma non l'uscita dal costrutto iterativo: il flusso dell'esecuzione prosegue ritornando alla valutazione della condizione relativa alla prossima iterazione;
- NOTA: se l'iterazione corrente è annidata dentro altre iterazioni più esterne, queste due istruzioni non influenzano tali iterazioni esterne ma solo quella corrente.

Ad esempio, il seguente frammento di codice è un ciclo `while` dal quale non si esce per via della condizione (che resta sempre vera) ma per mezzo dell'istruzione `break`:

```
int i = 1;
while (1) {
   printf("%d\n", i);
   i++;
   if (i >= 100) break;
}
```

Mentre, nel prossimo esempio, alcune iterazioni del ciclo non si eseguono per intero (in modo da non produrre le stampe dei valori pari):

```
int i = 0;
while (i <= 100) {
   i++;
   if (i%2 == 0) continue;
   printf("%d\n", i);
}
```

Nel prossimo esempio, l'istruzione `break` interrompe l'esecuzione del `while` ma non influenza in alcun modo quella del `for`:

```
for (i = 0; i < 100; i++) {
   int j = 1;
   while (1) {
      printf("%d\n", j);
      j++;
      if (j >= 100) break;
   }
}
```

5.5 Esempi ed esercizi

Esempio 03: calcolo fattoriale con ciclo while

Il programma seguente (esempio_03.c) legge un intero dalla finestra del terminale e stampa il fattoriale di tale valore.

```
#include <stdio.h>

int main() {
   int n;
   long n_fatt = 1;
   int i = 1;

   printf("Inserisci valore: ");
   scanf("%d", &n);

   while (i <= n ) {
      n_fatt = n_fatt*i;
      i = i + 1; //oppure i++
   }
   printf("Il fattoriale di %d vale %ld\n", n, n_fatt);
   return 0;
}
```

Esempio 04: ciclo do..while per gestione input

Il programma seguente (esempio_04.c) legge una serie di interi non negativi dalla finestra del terminale e li stampa. Il procedimento si arresta immettendo un numero negativo (che non viene stampato).

```
#include <stdio.h>

int main() {
   int i;

   do {
      scanf("%d", &i);
      if (i >= 0) printf("%d\n", i);
   } while (i >= 0);
   return 0;
}
```

Esempio 05: cicli for innestati

Il programma seguente (esempio_05.c) legge un intero positivo n dalla finestra del terminale e stampa un quadrato di asterischi di lato n:

```
#include <stdio.h>

int main() {
   int i, j, n;

   printf("Inserisci il numero di asterischi: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         printf("*");
      }
      printf("\n");
   }
   return 0;
}
```

Esercizio 03

Scrivere un programma che legga un numero intero n, verifichi se il numero immesso è primo o no e stampi un messaggio conseguente.

Suggerimento: una prima idea è provare a dividere il numero n per tutti gli interi da 2 a (n – 1): se anche una sola di queste divisioni ha resto zero il numero non è primo. Ragionando un po' di più ci si accorge che si può fare di meglio.

6 Puntatori

Le variabili, oltre ad essere dei tipi già visti, possono anche essere di tipo *puntatore* (*pointer*).

Il valore di una variabile di tipo puntatore è un indirizzo di memoria, che si utilizza per accedere al contenuto della porzione di memoria che inizia con tale indirizzo.

6.1 Dichiarazione

Ogni variabile di tipo puntatore viene dichiarata in connessione al tipo di dato cui deve puntare.

Per dichiarare una variabile di tipo puntatore si usa il simbolo `*`, come nell'esempio seguente.

Il seguente frammento di codice:

```
int i;
int *ptr_1;
```

dichiara due variabili:

- i è una variabile di tipo intero;
- `ptr_1` è una variabile di tipo "puntatore ad intero", che può essere usata per puntare alla locazione di memoria che contiene una qualunque variabile di tipo intero.

NOTE:

Quando si vogliono dichiarare più variabili di tipo puntatore nella stessa istruzione è obbligatorio ripetere l'asterisco prima del nome di ciascuna variabile:

```
int *ptr_1, *ptr_2;
```

Infatti, la seguente dichiarazione:

```
int *ptr_1, ptr_2;
```

ha come effetto quello di dichiarare:

- `prt_1` come puntatore ad intero;
- `ptr_2` come intero, e non come puntatore ad intero.

6.2 Assegnazione di valori

A una variabile di tipo puntatore (nel seguito, per brevità, detta semplicemente *puntatore*) si può assegnare un valore in alcuni modi diversi:

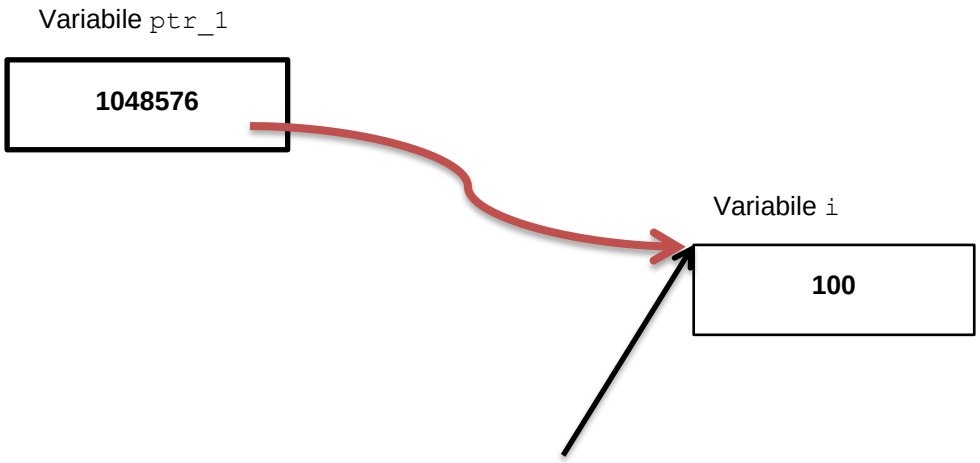
- se il puntatore non punta a nulla lo si inizializza con lo speciale valore `NULL` (definito nella libreria `stdlib`);
- al puntatore `ptr_1` si può assegnare il valore di un altro puntatore `ptr_2`: in tal caso dopo l'assegnazione sia `ptr_1` che `ptr_2` punteranno alla stessa locazione di memoria;

- a un puntatore si può assegnare l'indirizzo di memoria dove è già allocata una altra variabile, mediante l'operatore `&`; dopo tale assegnazione il valore del puntatore sarà l'indirizzo di memoria dell'altra variabile;
- si può assegnare un valore a un puntatore mediante la funzione `malloc()`, che vedremo fra breve.

Esempi di assegnazione di valori a puntatori:

```
int i = 100; //dichiarazione ed inizializzazione della variabile i
int *ptr_1; //dichiarazione di una variabile di tipo puntatore ad intero

ptr_1 = &i;
/* ora il valore di prt_1 è l'indirizzo di memoria dove è contenuta la
variabile i */
```

Dopo l'esecuzione di questo frammento di codice la situazione in memoria è la seguente:



Indirizzo di memoria del primo byte allocato alla variabile `i`: **1048576**

ATTENZIONE

Se nel programma si utilizza un puntatore non correttamente inizializzato la computazione di norma termina immediatamente con un errore di accesso alla memoria (segmentation fault) oppure, se il sistema di calcolo non è fornito di adeguati meccanismi di protezione della memoria, può portare a un crash di sistema.

Nell'esempio seguente si mostrano altri modi di assegnare valori al variabili di tipo puntatore.

```
int i = 100; //dichiarazione ed inizializzazione
int *ptr_1; //dichiarazione
int *ptr_2 = NULL; /dichiarazione ed inizializzazione
int *ptr_3 = NULL; /dichiarazione ed inizializzazione

ptr_1 = &i;
/* ora il valore di prt_1 è l'indirizzo di memoria dove è contenuta la
variabile i */

ptr_2 = ptr_1;
/* ora anche il valore di prt_2 è l'indirizzo di memoria dove è contenuta la
variabile i */

ptr_3 = (int*)malloc(sizeof(int));
/* la vedremo fra breve: si alloca una zona di memoria adatta a contenere un
intero e si assegna il relative indirizzo a ptr_3 */
```

6.3 Operatori

In relazione ai puntatori sono disponibili due operatori:

- l'operatore `&` (e commerciale o ampersand): operatore di indirizzamento;
- l'operatore `*` (da non confondere con l'operatore di moltiplicazione, nonostante il carattere sia il medesimo): operatore di deferenziazione o riferimento.

6.3.1 Operatore `&`

L'operatore `&` è un operatore unario (ossia si applica ad una singola variabile) che restituisce l'indirizzo iniziale della porzione di memoria nella quale è contenuta la variabile in questione.

Viene quindi spesso usato per assegnare a un puntatore l'indirizzo di memoria di un'altra variabile.

Si può comprendere ora la ragione per la quale viene utlilizzato nelle operazioni di lettura dati: la funzione `scanf()` si aspetta un puntatore a una porzione di memoria nella quale scrivere il dato letto dall'esterno.

Esempio:

```
int i = 100;
int j = 200;

int *ptr_1, *ptr_2;

ptr_1 = &i; //ptr_1 "punta" a i
ptr_2 = &j; //ptr_2 "punta" a j
```

6.3.1 Operatore *

L'operatore * è un operatore unario (ossia si applica ad una singola variabile di tipo puntatore) che restituisce il valore contenuto nella porzione di memoria il cui indirizzo iniziale è lo stesso del valore corrente del puntatore. In altre parole, applicando l'operatore * a un puntatore si ottiene il valore della variabile "puntata" dal puntatore. Questa operazione si chiama *deferenziazione del puntatore* (*pointer deferencing*).

Viene quindi usato ogniqualvolta si debba recuperare il valore (e non l'indirizzo!) di una variabile per mezzo di un puntatore ad essa agganciato.

```
int i = 100;
int j = 200;
int h, k;

int *ptr_1, *ptr_2;

ptr_1 = &i; //ptr_1 "punta" a i
h = *ptr_1; //h adesso vale 100
ptr_2 = &j; //ptr_2 "punta" a j
k = *ptr_2; //k adesso vale 200

ptr_1 = ptr_2; //adesso ptr_2 "punta" a j
h = *ptr_1; //h adesso vale 200
```

6.4 Allocazione dinamica della memoria

Utilizzando i puntatori diviene possibile allocare dinamicamente la memoria a tempo di esecuzione, senza dover dichiarare staticamente (ossia in anticipo, in fase di stesura del programma) la quantità di memoria necessaria.

Ciò è particolarmente utile, come vedremo fra breve, quando si vogliano utilizzare vettori o matrici senza che sia nota a priori la loro dimensione, ed è il modo più efficiente di gestire strutture inerentemente dinamiche quali liste, alberi e grafi.

L'allocazione dinamica della memoria si realizza mediante l'uso combinato di due funzioni definite nella libreria `stdlib.h` (che va quindi importata):

- la funzione `sizeof()`;
- la funzione `malloc()`.

La funzione `sizeof()` si utilizza per determinare la dimensione (in byte) di un tipo di dato che viene passato come parametro. Il tipo di dato può essere un tipo elementare (`int`, `long`, `float`, ecc.) ma può anche essere un tipo più complesso, come vedremo più avanti.

La funzione `malloc()` si utilizza per:

1. allocare una porzione di memoria, della quale si deve specificare la dimensione in byte (di norma tramite la `sizeof()`);
2. assegnare l'indirizzo (iniziale) di tale porzione di memoria ad un puntatore: il valore di ritorno restituito dalla funzione è l'indirizzo iniziale della porzione di memoria allocata dalla `malloc()`, ed è il valore che viene assegnato al puntatore.

Poiché `malloc()`, per come è definita, restituisce un "generico" puntatore, nella sua chiamata si deve usare la tecnica del cosidetto *type cast* (o *type coercion*), ossia una trasformazione del risultato restituito da un generico puntatore allo specifico tipo di puntatore della variabile puntatore che riceve il risultato della `malloc()`.

Ad esempio, questo frammento di codice alloca lo spazio di memoria necessario per un intero, assegna il relativo indirizzo ad un puntatore e, attraverso tale puntatore scrive il valore 100 in quella porzione di memoria:

```
int *ptr_1 = NULL;

ptr_1 = (int*)malloc(sizeof(int)); //allochiamo la memoria per un intero
*ptr_1 = 100; //scriviamo il valore 100 nella memoria "puntata" da ptr_1
```

6.5 Deallocazione della memoria

Quando una porzione di memoria, precedentemente allocata dinamicamente per mezzo della `malloc()`, non è più utilizzata si deve "liberarla", ossia renderla disponibile al programma per le prossime richieste di allocazione dinamica.

Ciò si ottiene con la funzione `free()`, che dealloca la memoria puntata dal puntatore che viene passato come parametro.

Nell'esempio seguente, dopo aver allocato una porzione di memoria, avervi scritto un valore ed averlo stampato, quella porzione di memoria viene deallocata:

```
int *ptr_1 = NULL;

ptr_1 = (int*)malloc(sizeof(int); //allochiamo la memoria per un intero
*ptr_1 = 100; //scriviamo il valore 100 nella memoria "puntata" da ptr_1
printf("%d\n", *ptr_1);
free(ptr_1);
```

Si noti che dopo l'esecuzione della `free()` il puntatore `ptr_1` punta alla stessa porzione di memoria cui puntava prima, nella quale però altri processi potrebbero scrivere, quindi non deve essere utilizzato se prima non gli viene riassegnato un valore (ossia un indirizzo di memoria) valido nel contesto del programma che si sta eseguendo.

6.6 Esempio 06: allocazione dinamica e deallocazione di interi

Il programma seguente (esempio_06.c) offre la stessa funzionalità dell'esempio 04, ma utilizza l'allocazione dinamica.

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

int main() {
   int *ptr_i;

   ptr_i = (int*)malloc(sizeof(int));
```

```
   do {
      scanf("%d", ptr_i); // si noti: non serve l'operatore &
      if (*ptr_i >= 0) printf("%d\n", *ptr_i);
   } while (*ptr_i >= 0);

   free(ptr_i);
   return 0;
}
```

L'uso della `free()` non è necessario nel caso in cui la memoria allocata dinamicamente serva fino alla fine dell'esecuzione del programma. In tal caso, infatti, la terminazione del programma ha come effetto la liberazione di tutta la memoria allocata al programma, comprensiva quindi di quella allocata dinamicamente. E' in effetti il caso dell'esempio precedente, nel quale tuttavia la presenza delle chiamate a `free()` ha lo scopo di illustrarne l'uso.

6.7 Puntatori a puntatori

Dato che il valore di una variabile `x` di tipo puntatore è un indirizzo di memoria, nulla vieta che essa "punti" ad una variabile `y` che è a sua volta un puntatore a una terza variabile `z`.

In questo caso si dice che la variabile `x` è un *puntatore a puntatore*: attraverso essa si può accedere direttamente, con l'opportuna sintassi, alla variabile `z` effettuando, quindi, una *doppia deferenziazione*.

I puntatori a puntatori sono fondamentali in due situazioni:

- allocazione dinamica di matrici (che vedremo fra breve);
- passaggio di parametri alle funzioni (che vedremi più avanti).

Le sintassi per dichiarare una variabile di tipo puntatore a puntatore e per accedere al valore della variabile "puntata" tramite doppia deferenziazione sono analoghe a quelle già viste per i puntatori, ma si adoperano i simboli `**` anziché il simbolo `*`.

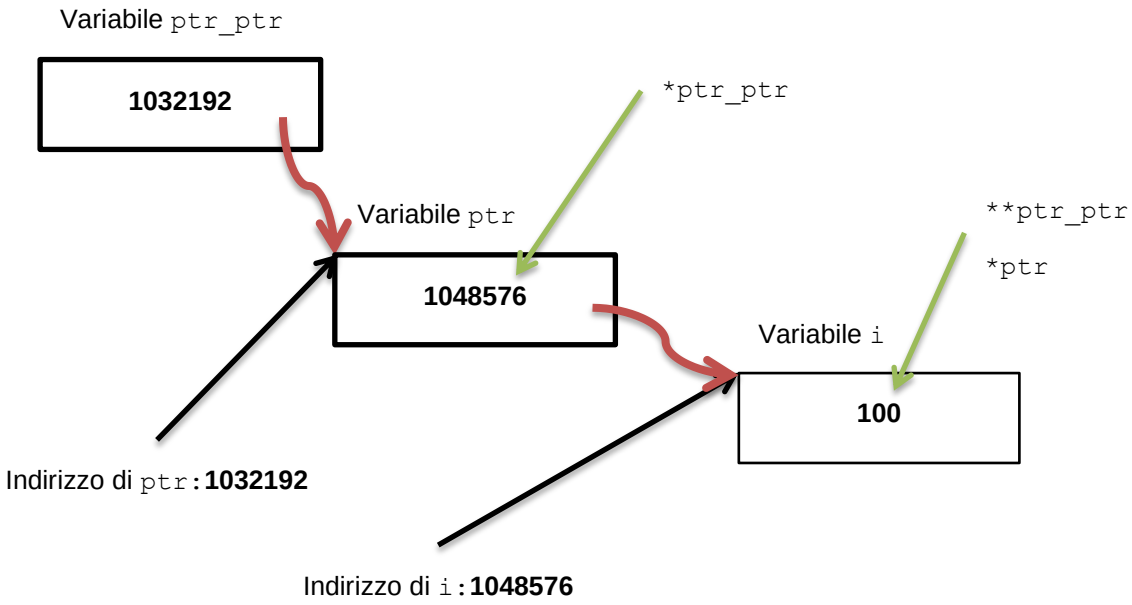
Vediamo un esempio. Supponiamo di volere:

- una variabile i di tipo int;
- una variabile `ptr` di tipo puntatore ad intero che (mediante deferenziazione) punti ad `i`;
- una variabile `ptr_ptr` di tipo puntatore a puntatore ad intero che (attraverso doppia deferenziazione) punti ad `i`.

Il codice che produce quanto desiderato mediante allocazione statica della memoria è il seguente:

```
int i = 100; //dichiarazione ed inizializzazione della variabile i
int *ptr; //variabile di tipo puntatore ad intero
int **ptr_ptr; //variabile di tipo puntatore a puntatore ad intero

ptr = &i;
/* ora il valore di prt è l'indirizzo di memoria dove è contenuta la variabile i */
ptr_ptr = &ptr;
/* ora il valore di prt_ptr è l'indirizzo di memoria dove è contenuta la variabile ptr */
```

e la situazione in memoria è la seguente:



Si noti che, relativamente all'esempio precedente:

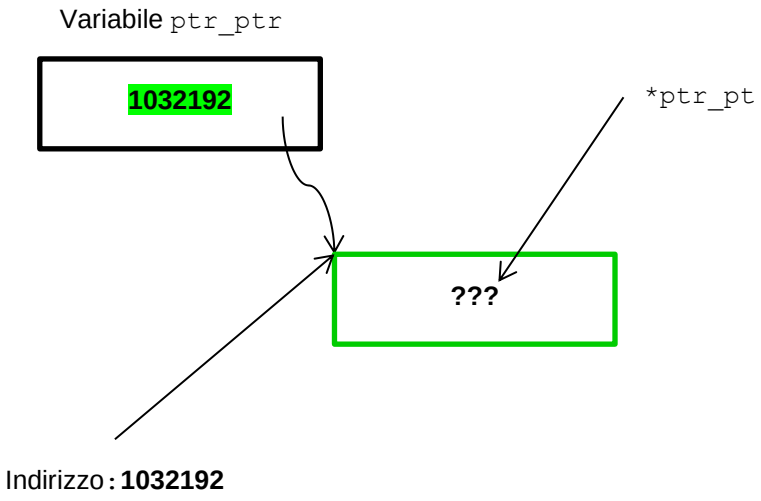
- l'istruzione `int h = **ptr_ptr` assegna ad `h` il valore **100**, ossia il valore della variabile accessibile da `ptr_ptr` tramite doppia deferenziazione;
- l'istruzione `long k = *ptr_ptr` assegna a `k` il valore **1048576**, ossia il valore della variabile accessibile da `ptr_ptr` tramite singola deferenziazione; tale variabile è `ptr` che, essendo una variabile di tipo puntatore, ha un valore che è un indirizzo di memoria.

Un'altra possibilità è utilizzare l'allocazione dinamica della memoria, come illustrato nel seguente frammento di codice:

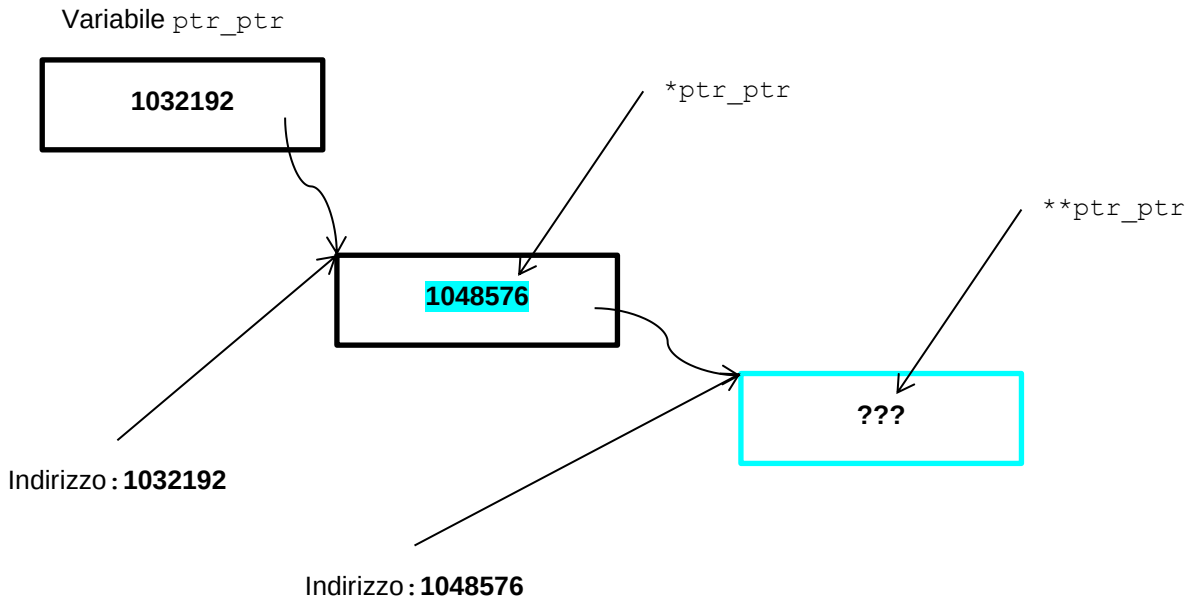
```
int **ptr_ptr = NULL; //dichiariamo un puntatore a puntatore

ptr_ptr = (int**)malloc(sizeof(int*));//1 allochiamo la memoria per un punt.
*ptr_ptr = (int*)malloc(sizeof(int)); //2 allochiamo la memoria per un intero
**ptr_ptr = 100; //3-scriviamo il valore 100 nella memoria "puntata"
                 //mediante doppia deferenziazione da ptr_ptr
```

Nelle figure seguenti i colori associano le singole istruzioni alle operazioni effettivamente svolte e il "???" indica la presenza di un valore preesistente in memoria, valore perciò non utilizzabile dal programma.

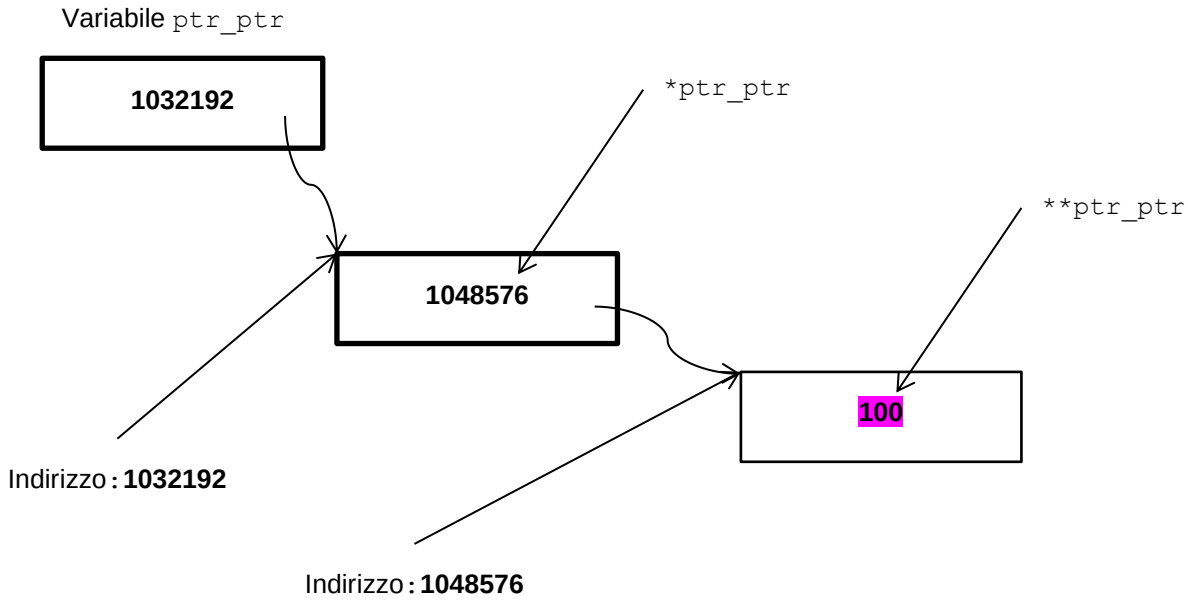
Dopo l'esecuzione della prima delle tre istruzioni la situazione in memoria è la seguente:



Dopo l'esecuzione della seconda delle tre istruzioni la situazione in memoria è la seguente



Dopo l'esecuzione della terza delle tre istruzioni la situazione in memoria è la seguente:



L'unica differenza rispetto all'allocazione statica è che non abbiamo più accesso al puntatore intermedio (che non abbiamo dichiarato perché non necessario) se non attraverso `*ptr_ptr`.

7 Vettori

I vettori sono strutture dati che consistono di una collezione di elementi tutti dello stesso tipo. Il tipo di dato può essere un tipo elementare (`int`, `long`, `float`, ecc.) ma può anche essere un tipo più complesso, come vedremo più avanti.

I singoli elementi del vettore si riferiscono per mezzo di un *indice*, di valore intero, racchiuso fra parentesi quadre:

Ad esempio, `A[10]` fornisce il valore dell'elemento di indice 10 del vettore `A`. Viceversa, `&A[10]` fornisce l'indirizzo di memoria dell'elemento di indice 10.

E' importante ricordare che in C gli indici degli elementi di ogni vettore partono da zero (e non da uno) ed arrivano a (numero degli elementi del vettore – 1). Ad esempio, per un vettore contenente 20 elementi gli indici vanno da zero a 19.

7.1 Dichiarazione e allocazione

Un vettore può essere dichiarato e allocato staticamente o dinamicamente:

- la prima opzione implica che sia fissata a priori la dimensione del vettore, e per ottenerla si usa la notazione parentetica:

  ```
  int A[10];
  ```

- la seconda, viceversa, permette di dichiarare il vettore senza specificarne le dimensioni e si ottiene dichiarando il vettore come un semplice puntatore:

  ```
  int *B;
  ```

  e poi, a tempo di esecuzione, allocando il vettore nella dimensione voluta (dimensione che può provenire da una variabile, `n` nell'esempio):

  ```
  B = (int*)malloc(sizeof(int)*n);
  ```

  dopo questa istruzione il puntatore `B` punta all'indirizzo del primo elemento del vettore.

In entrambi i casi, per accedere agli elementi si usa il nome del vettore seguito dall'indice dell'elemento racchiuso fra parentesi quadre.

Nell'esempio seguente si mostrano entrambe le opzioni relativamente a due vettori A e B contenenti ciascuno 10 numeri interi (`int`).

Il vettore A è dichiarato e allocato staticamente, il vettore B è dichiarato senza specifica della dimensione ed è poi allocato dinamicamente per mezzo della `malloc()`, alla quale si passa una

dimensione che è il prodotto di `sizeof(int)` per il numero di elementi che il vettore deve contenere.

Il programma quindi assegna valori a tutti gli elementi di entrambi i vettori, somma i valori di indice uguale dei due vettori e stampa tali somme.

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

int main() {

   int i, n;
   int A[10];
   int *B;

   n = 10; //ma può essere anche letto dall'esterno
   B = (int*)malloc(sizeof(int)*n);

   for (i = 0; i < 10; i++) {
      A[i] = 2*i;
   }
   for (i = 0; i < 10; i++) {
      B[i] = 3*i;
   }
   for (i = 0; i < 10; i++) {
      printf("%d ", A[i]+B[i]);
   }
   printf("\n");
   return 0;
}
```

Nell'ambito di questo insegnamento useremo sempre e solo l'allocazione dinamica, per le seguenti ragioni:

- è una tecnica più versatile dell'allocazione statica;
- permette di iniziare a familiarizzare con l'uso dei puntatori.

Per deallocare un vettore allocato staticamente si utilizza la funzione `free()` passandole il puntatore al vettore. Nel caso dell'esempio soprariportato:

```
free(B);
```

7.2 Esempi ed esercizi

Esempio 07: allocazione dinamica vettore

Il programma seguente (esempio_07.c) legge un intero dalla finestra del terminale e lo utilizza come dimensione di un vettore di interi da allocare dinamicamente. Quindi acquisisce da tastiera tutti i valori da inserire nel vettore ed infine stampa tali valori su una stessa riga, separandoli con uno spazio.

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

int main() {

   int i, n;
   int val;
   int *A;

   printf("Inserisci la dimensione del vettore: ");
   scanf("%d", &n);

   A = (int*)malloc(sizeof(int)*n);
   for (i = 0; i < n; i++) {
      printf("Inserisci il prossimo valore del vettore: ");
      scanf("%d", &val);
      A[i] = val;
   }
   for (i = 0; i < n; i++) printf("%d ", A[i]);
   printf("\n");

   return 0;
}
```

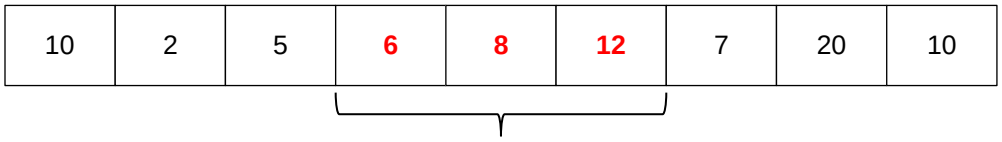
Esempio 08: massimo di un vettore

Il programma seguente (esempio_08.c), una volta acquisito il vettore, trova e stampa il suo elemento massimo.

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

int main() {

   int i, n, val, max;
   int *A;

   printf("Inserisci la dimensione del vettore: ");
   scanf("%d", &n);

   A = (int*)malloc(sizeof(int)*n);
   for (i = 0; i < n; i++) {
      printf("Inserisci il prossimo valore del vettore: ");
      scanf("%d", &val);
      A[i] = val;
   }
   max = A[0]; //inizializziamo il massimo: Θ(1)
   for (i = 1; i < n; i++) //scandiamo gli altri elementi
                           //del vettore: (n-1) iterazioni + Θ(1)
      if(A[i] > max) max = A[i]; //aggiorniamo il nuovo massimo: Θ(1)+Θ(1)
   printf("Il massimo vale: %d\n", max); // Θ(1)

   return 0;
}
```

Esempio 09: sottovettore di elementi pari più lungo

Il programma seguente (esempio_09.c), una volta acquisito il vettore, trova e stampa la lunghezza del più lungo sottovettore composto di numeri pari:




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

int main() {
   int i, n, val;
   int count = 0;//contatore del numero di pari contigui incontrati
   int max_len = 0;//lunghezza del massimo sottovettore incontrato sino ad ora
   int *A;

   printf("Inserisci la dimensione del vettore: ");
   scanf("%d", &n);

   A = (int*)malloc(sizeof(int)*n);
   for (i = 0; i < n; i++) {
      printf("Inserisci il prossimo valore del vettore: ");
      scanf("%d", &val);
      A[i] = val;
   }

   for (i   = 0; i < n; i++){//scorro il vettore
      if (A[i]%2 == 0){//incontro un pari
         //incremento il contatore della lunghezza del sottovettore corrente:
         count++;
         //se necessario aggiorno il massimo:
         if (max_len < count) max_len = count;
      }else{   //incontro un dispari, il sottovettore corrente e' terminato
               //quindi riazzero il contatore
         count = 0;
      }
   }
   printf("Il sottovettore di numeri pari di lungh. massima e' lungo %d\n", max_len);
   return 0;
}
```

Esercizio 04

Modificare il programma dell'esempio 09 in modo che stampi anche gli indici di inizio e di fine del più lungo sottovettore composto di numeri pari.

Suggerimento: modificare l'esempio 09 gestendo, con ulteriori variabili, gli indici di inizio e fine del più lungo sottovettore che via via viene trovato. Attenzione a un fatto: lo zero è un valore valido per gli indici degli elementi del vettore, quindi non deve essere restituito come indice di inizio (o fine) del sottovettore se esso non esiste (è il caso di un vettore contenente solo elementi dispari).

Esercizio 05

Scrivere un programma che, acquisiti due vettori A e B ciascuno di dimensione n, trovi la lunghezza del più lungo sottovettore comune.

Suggerimento: tenere presente che il sottovettore comune, se esiste, può iniziare da un qualunque elemento di A e da un qualunque elemento di B. Quindi serve:

- un meccanismo per gestire tutte le possibili posizioni di inizio nei due vettori (doppio ciclo for?);
- per ogni coppia di tali posizioni di inizio nei due vettori, un altro meccanismo (while?) che scorra i due vettori in modo coordinato, senza uscire dai loro range di elementi, per identificare l'eventuale sottovettore comune.

8 Matrici

Le matrici, ben note in matematica, in C sono implementate in due modi differenti a seconda che siano allocate staticamente o dinamicamente.

In entrambi i casi, una volta dichiarata ed allocata una matrice, i suoi elementi si riferiscono per mezzo di una *coppia di indici*, di valore intero, racchiusi ciascuno da una coppia di parentesi quadre, ad esempio:

```
M[i][i] = 100;
```

```
value = M[h][k];
```

Convenzionalmente il primo indice si considera l'indice di riga ed il secondo l'indice di colonna.

E' importante ricordare che, in C, anche nel caso delle matrici gli indici degli elementi partono da zero e non da uno. Ad esempio, il seguentre frammento di codice stampa i valori di una matrice `M` avente `n` righe ed `m` colonne:

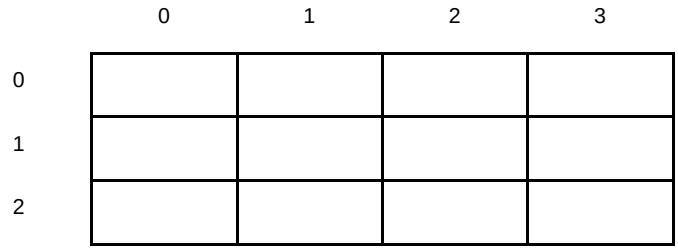
```
for (i = 0; i < n; i++) {
   for (j = 0; j < m; j++) {
      printf("%d ", M[i][j]);
   }
   printf("\n");
}
```

8.1 Dichiarazione e allocazione

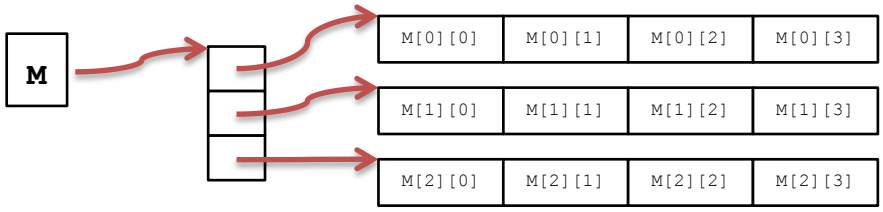
Una matrice può essere dichiarata e allocata staticamente o dinamicamente. Noi considereremo solo l'allocazione dinamica.

In C una matrice allocata dinamicamente risulta in un *vettore di puntatori*, avente tanti elementi quante sono le righe. Ciascun elemento del vettore è un puntatore ad un vettore di elementi della matrice, avente tanti elementi quanti sono quelli presenti nella corrispondente riga (le matrici allocate in questo modo non sono necessariamente rettangolari).

Ad esempio, la matrice **M** di tre righe e quattro colonne:



in memoria è allocata così:



Come si vede chiaramente dalla figura, la variabile `M` è un puntatore a puntatore poiché ci vuole una doppia deferenziazione per accedere, a partire da `M`, agli elementi della matrice.

Il codice per dichiarare la matrice `M` ed allocarla dinamicamente è il seguente:

```
int **M;

int n = 3;                                  //numero di righe
int m = 4;                                  //numero di colonne
int i;

M = (int**)malloc(n*sizeof(int*));          //allochiamo il vettore di
                                            //puntatori alle righe
for (i = 0; i < n; i++)                     //allochiamo tutti i vettori
   M[i] = (int*)malloc(m*sizeof(int));      //delle righe
```

Per deallocare una matrice allocata staticamente si utilizza la funzione `free()` ripetutamente; bisogna prima deallocare i vettori relativi alle singole righe della matrice, quindi il vettore di puntatori alle righe:

```
for (i = 0; i < n; i++) free(M[i]);

free(M);
```

8.2 Esempi ed esercizi

Esempio 10: allocazione dinamica matrice

Il programma seguente (esempio_10.c), dichiara una matrice da allocare dinamicamente, legge da finestra di terminale le sue dimensioni e quindi i valori degli elementi ed infine stampa la matrice riga per riga.

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

int main() {

   int** M;
   int n, m;
   int i, j;
   int val;

   printf("Numero di righe: ");
   scanf("%d", &n);
   printf("Numero di colonne: ");
   scanf("%d", &m);

   M = (int**)malloc(n*sizeof(int *));
   for (i = 0; i < n; i++)
      M[i] = (int*)malloc(m*sizeof(int));
   for (i = 0; i < n; i++)
      for (j = 0; j < m; j++) {
         scanf("%d", &val);       //leggiamo i valori degli elementi
         M[i][j] = val;
      }
   for (i = 0; i < n; i++) {
      for (j = 0; j < m; j++) {
         printf("%d ", M[i][j]);  //stampiamo i valori degli elementi
      }
      printf("\n");
   }
   return 0;
}
```

Esempio 11: un test sugli elementi di una matrice

Il programma seguente (esempio_11.c), dichiara una matrice da allocare dinamicamente, legge da finestra di terminale le sue dimensioni e quindi i valori degli elementi ed infine calcola e stampa il numero degli elementi tali per cui `M[i][j] == i + j`.

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

int main() {
   int** M;
   int n, m;
   int i, j;
   int val;
   int count;

   printf("Numero di righe: ");
   scanf("%d", &n);
   printf("Numero di colonne: ");
   scanf("%d", &m);

   M = (int**)malloc(n*sizeof(int *));
   for (i = 0; i < n; i++)
      M[i] = (int*)malloc(m*sizeof(int));
   for (i = 0; i < n; i++)
      for (j = 0; j < m; j++) {
         scanf("%d", &val);          //leggiamo i valori degli elementi
         M[i][j] = val;
      }
   count = 0;
   for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
         if (M[i][j] == i + j) count++;
   printf("Il numero degli elementi cercati e': %d\n", count);
   return 0;
}
```

Esempio 12: trasposta di una matrice

Il programma seguente (esempio_12.c), calcola e stampa la matrice trasposta `MT` di una matrice `M` letta in input.

```
int main() {
   int** M; //matrice in input
   int** MT; //matrice trasposta
   int n, m;
   int i, j;
   int val;

   printf("Numero di righe di M: ");
   scanf("%d", &n);
   printf("Numero di colonne di M: ");
   scanf("%d", &m);

   M = (int**)malloc(n*sizeof(int *)); //allochiamo M
   for (i = 0; i < n; i++)
      M[i] = (int*)malloc(m*sizeof(int));
   MT = (int**)malloc(m*sizeof(int *)); //allochiamo MT
   for (i = 0; i < m; i++)
      MT[i] = (int*)malloc(n*sizeof(int));
   for (i = 0; i < n; i++)
      for (j = 0; j < m; j++) {
         scanf("%d", &val); //leggiamo i valori di M
         M[i][j] = val;
      }
   for (i = 0; i < n; i++) //calcoliamo la matrice trasposta MT
      for (j = 0; j < m; j++)
         MT[j][i] = M[i][j];
   for (i = 0; i < m; i++) { //stampiamo MT
      for (j = 0; j < n; j++) {
         printf("%d ", MT[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

Esempio 13: redirezione dell'input

Il sorgente è identico a quello dell'esempio 12, ma va eseguito con redirezione dell'input. Il programma va lanciato col comando:

```
$ ./esempio_12.out < numeri.txt
```

dopo aver predisposto un file "numeri.txt" che contenga i valori necessari al programma (ad esempio questi):

```
5
7
27
…
…      35 numeri interi, uno per riga, senza null'altro su ogni riga se non il numero stesso
…
51
```

Si ponga attenzione a che i valori contenuti nel file "numeri.txt" siano congruenti con quanto il programma si aspetta sulla base delle stringhe di formato usate nelle `scanf()`.

Come si noterà eseguendolo, è opportuno eliminare dal sorgente le due stampe esplicative che indicano all'utente di inserire il numero di righe e il numero di colonne.

Esempio 14: la matrice è simmetrica?

Il programma seguente (`esempio_14.c`), verifica se una matrice quadrata è simmetrica.

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

int main() {
   int **M;
   int n;
   int i, j;
   int val;
   int simmetrica = 1;

   printf("Numero di righe e colonne: ");
   scanf("%d", &n);

   M = (int**)malloc(n*sizeof(int*));
   for (i = 0; i < n; i++)
      M[i] = (int*)malloc(n*sizeof(int));
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         scanf("%d", &val);              //leggiamo i valori degli elementi
         M[i][j] = val;
      }
   for (i = 1; i < n; i++)
      for (j = 0; j < i; j++)
         if (M[i][j] != M[j][i]) simmetrica = 0;
   if (simmetrica == 1) printf("La matrice e' simmetrica\n");
   else printf("La matrice non e' simmetrica\n");
   return 0;
}
```

Si noti che il codice proposto effettua sempre e comunque il controllo su tutti gli elementi della matrice (tranne quelli della diagonale principale), quindi non sfrutta il fatto che appena si rileva che la condizione `M[i][j] != M[j][i]` è verificata (e quindi la matrice non è simmetrica) si può immediatamente terminare l'iterazione.

Però un semplice `break` non è sufficiente, poiché interromperebbe solo il ciclo più interno e non quello esterno.

E' necessario impostare una struttura un pò più complicata.

Una prima soluzione è utilizzare un break nel ciclo più interno ed un altro nel ciclo più esterno:

```
for (i = 1; i < n; i++) {
   if (simmetrica == 0) break;
   for (j = 0; j < i; j++)
      if (M[i][j] != M[j][i]) {
         simmetrica = 0;
         break;
      }
}
```

Una seconda soluzione è basata sull'uso di due cicli `while` innestati, in ciascuno dei quali si verifica sia il valore dell'indice appropriato che il valore della variabile `simmetrica`:

```
i = 1;
while ((simmetrica == 1)&&(i < n)) {
   j = 0;
   while ((simmetrica == 1)&&(j < i)) {
      if (M[i][j] != M[j][i]) simmetrica = 0;
      j = j + 1;
   }
   i = i + 1;
}
```

Esercizio 06

Scrivere un programma che, acquisita una matrice quadrata M di dimensione n, determini se la matrice è un quadrato magico, ossia una matrice tale per cui le somme degli elementi di ciascuna riga, quelle degli elementi di ciascuna colonna, la somma degli elementi della diagonale principale e la somma degli elementi della diagonale secondaria siano tutte uguali fra loro:

- $\forall i: \sum_{j=0}^{n-1} M[i][j] = k$
- $\forall j: \sum_{i=0}^{n-1} M[i][j] = k$
- $\forall i: \sum_{i=0}^{n-1} M[i][i] = k$
- $\forall i: \sum_{i=0}^{n-1} M[n-i-1][i] = k$

per un *k* qualunque.

Esercizio 07

Una matrice M, quadrata, di dimensione n rappresenta i rapporti di conoscenza di un gruppo di n persone secondo quanto segue:

- contiene 2 sulla diagonale principale: M[i,i] = 2 per i = 0, …, m-1;
- contiene 1 in posizione i,j se i conosce j: M[i,j] = 1 se i conosce j, con j ≠ i.

Si deve verificare se esiste un "VIP" nel gruppo, cioè se esiste un indice i tale che i e' conosciuto da tutti e non conosce nessuno. Formalmente:

- M[i,j] = 0 per i ≠ j
- M[j,i] = 1 per i ≠ j

Osservazione: esiste una soluzione semplice che ha complessità $O(n^2)$, ma ne esiste una meno banale che ha complessità O(n).

Suggerimenti:

- non può esserci più di un VIP;
- se un elemento della matrice è zero, l'indice della relativa colonna non è un VIP;
- se un elemento della matrice è uno, l'indice della relativa riga non è un VIP.

9) Funzioni

Le *funzioni* sono delle porzioni del programma opportunamente organizzate in modo da:

- effettuare una computazione specifica sui dati che vengono loro forniti (detti *parametri*);
- eventualmente restituire un *risultato* della computazione.

Nella terminologia standard dei linguaggi di programmazione esse vengono spesso chiamate:

- *funzioni* quando restituiscono un risultato;
- *procedure* quando non restituiscono un risultato.

In C non esistono procedure, ma solo funzioni: possono restituire o no un risultato, a seconda di come vengono costruite.

Le funzioni sono uno strumento importantissimo per favorire:

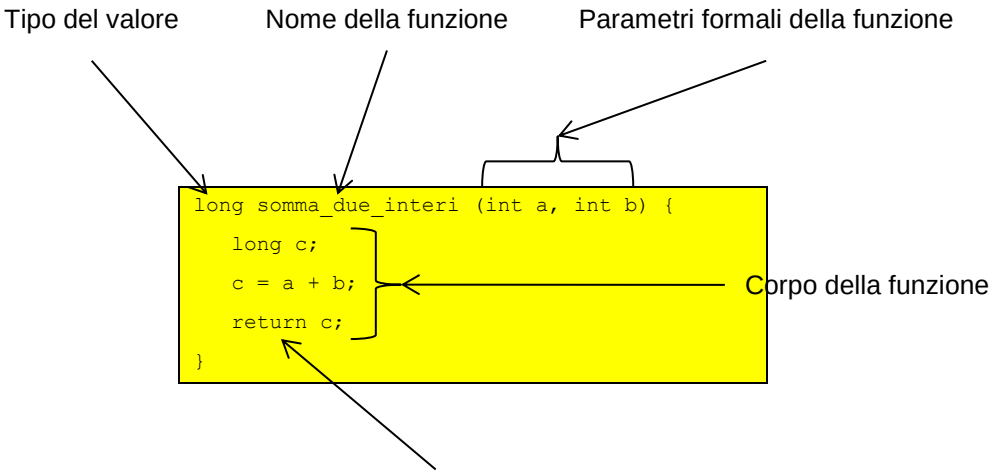
- la facilità di progettazione della soluzione complessiva e la chiarezza del programma, dato che consentono la suddivisione del problema da risolvere in sottoproblemi da affidare ciascuno a una o più funzioni;
- la riusabilità del codice, dato che una stessa funzione può essere utilizzata in tutti i problemi (e quindi i programmi) nei quali parte della soluzione complessiva si basa sulla soluzione del sottoproblema risolto da quella funzione.

9.1 Dichiarazione di una funzione

Una funzione va dichiarata prima di poter essere utilizzata. Per dichiarare una funzione è necessario specificare:

- il nome della funzione;
- i tipi di dati sui quali la funzione dovrà operare, ossia i parametri (detti in questo contesto *parametri formali*) della funzione;
- se la funzione restituisce un risultato oppure no:
    - in caso affermativo si deve specificare il tipo del risultato, che può essere un tipo essere un tipo primitivo (`int`, `float`, ecc.) oppure un tipo più complesso definito dall'utente (come vedremo più avanti);
    - viceversa si usa la parola chiave `void` per indicare il fatto che la funzione non restituisce alcun risultato;
- il *corpo* della funzione, ossia le istruzioni che devono essere eseguite quando si manda in esecuzione la funzione.

Un esempio di dichiarazione di funzione che restituisce un risultato (la somma di due valori interi passati come parametri) è il seguente.



Un esempio di funzione che non restituisce alcun valore (stampa i valori di due interi passati come parametri) è il seguente.

```
void stampa_due_interi (int a, int b) {
   printf("a=%d b=%d\n", a, b);
   return;
}
```

Aspetti importanti:

- le variabili dichiarate all'interno di una funzione sono *locali alla funzione stessa*: in altre parole hanno come ambito di visibilità la funzione stessa e al di fuori di essa non sono né definite né utilizzabili;
- i parametri formali sono anch'essi locali, cioè definiti e utilizzabili solo all'interno della funzione stessa;
- la funzione restituisce l'eventuale valore di ritorno attraverso l'istruzione `return`:
  - se la funzione restituisce un valore, la parola chiave `return` deve essere seguita dal valore che viene restituito (ad es. `return c;` nel primo dei due esempi sopra riportati);
  - se la funzione non restituisce alcun valore la parola chiave return non è seguita da alcun valore (come nel secondo esempio) e può essere omessa se è l'ultima istruzione;
- in entrambi i casi l'istruzione `return`, quando eseguita, causa l'immediata uscita dal corpo della funzione.

9.2 Chiamata (esecuzione) di una funzione

Per eseguire una funzione si esegue una operazione detta *chiamata della funzione*. Essa si concretizza in un'istruzione che contiene:

- il nome della funzione chiamata;
- i valori (ossia i parametri) che devono essere passati alla funzione: nel contesto di una chiamata di funzione tali valori si dicono *parametri attuali*, per distinguerli dai parametri formali di cui alla dichiarazione della funzione; NOTA: i valori passati come parametri attuali in fase di chiamata vengono assegnati ordinatamente ai parametri formali, e devono essere ciascuno compatibile con il valore del corrispondente parametro formale;
- se la funzione restituisce un valore, allora l'istruzione di chiamata della funzione di solito comprende anche l'assegnazione ad una variabile del valore restituito dalla funzione (ciò comunque non è obbligatorio).

Ad esempio, il seguente frammento di codice chiama la funzione `somma_due_interi` vista precedentemente:

```
int h = 10;
int k = 20;
long s;

s = somma_due_interi(h, k);
```

Dopo la chiamata di funzione, la variabile `s` vale 30.

il seguente frammento di codice invece chiama la funzione `stampa_due_interi` vista precedentemente, che non restituisce alcun risultato:

```
int h = 10;
int k = 20;

stampa_due_interi(h, k);
```

9.2.1 Esempio 15: calcolo del fattoriale tramite funzione

Riscriviamo il programma dell'esempio 03 (che legge un intero dalla finestra del terminale e stampa il fattoriale di tale valore) organizzando il calcolo del fattoriale per mezzo di una funzione (`esempio_15.c`).

```
#include <stdio.h>

long fattoriale (int n) {
   long n_fatt = 1;
   int i = 1;

   while (i <= n ) {
      n_fatt = n_fatt*i;
      i = i + 1; //oppure i++
   }
   return n_fatt;
}

int main() {
   int val;
   long val_fatt;

   printf("Inserisci valore: ");
   scanf("%d", &val);
   val_fatt = fattoriale(val);
   printf("Il fattoriale di %d vale %ld\n", val, val_fatt);
   return 0;
}
```

E' buona norma di programmazione non usare gli stessi nomi per i parametri formali (`val` nell'esempio) e le variabili che verranno passate come parametri attuali (`n` nell'esempio).

Altrettanto vale per i nomi delle variabili globali e quelli delle variabili locali delle funzioni.

In proposito si ricordi comunque che, qualora tali nomi coincidano, la variabile globale perde di validità all'interno della funzione, dove l'ambito di visibilità di quella locale alla funzione prende il sopravvento.

La chiamata di una funzione, solo nel caso in cui essa restituisca un risultato, può essere inserita ovunque sia possibile utilizzare un valore dello stesso tipo di quello restituito dalla funzione. Quindi, ad esempio, la chiamata può apparire all'interno di una espressione:

```
if (fattoriale(n) > fattoriale(m)) {
   …
} else {
   …
}
```

o anche come parametro attuale in un'altra chiamata di funzione:

```
printf("Il fattoriale di %d vale %ld\n", n, fattoriale(n));
```

9.3 Passaggio dei parametri

Il passaggio dei valori sui quali la funzione deve operare si chiama *passaggio dei parametri*, viene effettuato per mezzo dei parametri attuali e in linea di principio può avvenire in due modi diversi:

- passaggio dei parametri *per valore*: viene effettuata una copia di ogni variabile presente fra i parametri attuali utilizzati nella chiamata, ed alla funzione vengono trasmesse tali copie. La conseguenza è che qualunque modifica sui parametri attuali operata dentro il corpo della funzione agisce sulle copie e non sulle variabili utilizzate nel programma chiamante, le quali quindi dopo la terminazione della funzione non subiscono alcuna modifica;
- passaggio dei parametri *per riferimento*: viene passato alla funzione l'indirizzo di memoria di ogni variabile presente fra i parametri attuali utilizzati nella chiamata. La conseguenza è che qualunque modifica sui parametri attuali operata dentro il corpo della funzione agisce proprio sulle variabili utilizzate nel programma chiamante, le quali quindi dopo la terminazione della funzione riflettono tutte le modifiche operate all'interno della funzione.

In molti linguaggi di programmazione si usano due sintassi diverse a seconda dell'alternativa che si adottare, ma non in C.

In C l'unico meccanismo esistente per il passaggio dei parametri e' per valore: si crea sempre e comunque una copia di ogni parametro attuale ed alla funzione si passano le copie.

Ad esempio, nel seguente programma una variabile `n` viene passata come parametro attuale a una funzione che ne modifica il valore, ma al rientro nel programma chiamante la modifica si perde, dato la funzione ha operato su una copia della variabile.

```
#include <stdio.h>

void incrementa (int val) { //il passaggio e' per valore
   val = val + 1;           //la modifica opera su una copia
}

int main() {
   int n = 10;
   incrementa(n);     //n non viene influenzato dalla funzione
   printf("%d\n", n); //la stampa scrive il valore 10 e non 11
   return 0;
}
```

Di conseguenza, volendo ottenere un funzionamento identico al passaggio dei parametri per riferimento si deve passare nel parametro attuale, anziché la variabile, l'indirizzo di tale variabile tramite l'operatore `&` (oppure passando un puntatore alla variabile). L'indirizzo viene passato per valore (ne viene creata una copia), ma attraverso esso è possibile operare delle modifiche nella locazione di memoria puntata dall'indirizzo. Tali modifiche agiscono sulla memoria dove è allocata la variabile puntata e quindi permangono anche dopo la terminazione della funzione. Un esempio è il seguente.

```
#include <stdio.h>

void incrementa (int *val) { //l'indirizzo e' passato per valore
   *val = *val + 1;          //si scrive nella locazione di memoria
}                            //puntata dall'indirizzo

int main() {
   int n = 10;
   int *ptr = &n;     //indirizzo di memoria della variabile n
   incrementa(&n);    //oppure: incrementa(ptr);
   printf("%d\n", n); //la stampa scrive il valore 11 perche'
   return 0;          //la funzione ha modificato la locazione
}                     //di memoria dove e' allocata la variabile n
```

Ciò può essere necessario quando, ad esempio, la funzione deve restituire più di un risultato: uno di essi può essere restituito tramite il nome della funzione, gli altri devono essere restituiti sotto forma di modifiche alle opportune variabili del programma chiamante, delle quali perciò alla funzione si dovranno passare gli indirizzi di memoria tramite puntatori.

Si tenga presente che, nel caso si passi alla funzione un vettore `V` o una matrice `M` allocati dinamicamente, i relativi identificatori sono essi stessi dei puntatori (puntatore semplice nel caso del vettore, puntatore a puntatore nel caso della matrice) e quindi eventuali modifiche ai loro elementi operate all'interno della funzione permangono anche dopo la terminazione della funzione. In altre parole, di fatto il passaggio di vettori o matrici allocati dinamicamente avviene per riferimento (ciò che viene passato per valore è il puntatore al vettore o alla matrice). E' sempre necessario passare fra i parametri la dimensione del vettore (entrambe le dimensioni nel caso della matrice) poiché il puntatore di per se non convoglia tali informazioni.

Ad esempio, le seguenti due funzioni operano su un vettore il cui puntatore è passato come parametro formale: la prima funzione riempie il vettore di valori, la seconda funzione stampa i valori del vettore.

```
void leggi_vettore(int *vet, int dimensione) {
   int i, val;

   for (i = 0; i < dimensione; i++) {
         scanf("%d", &val);
         vet[i] = val;
   }
}

void stampa_vettore(int *vet, int dimensione) {
   int i;

   for (i = 0; i < dimensione; i++)
      printf("%d ", vet[i]);
   printf("\n");
}
```

Analoghe funzioni possono essere scritte per operare su matrici anziché su vettori: la principale differenza è che il parametro formale relativo alla matrice è un puntatore a puntatore anziché un puntatore:

```
void leggi_matrice(int **mat, int righe, int colonne) {
    int i, j, val;

    for (i = 0; i < righe; i++)
        for (j = 0; j < colonne; j++) {
            scanf("%d", &val);
            mat[i][j] = val;
        }
}

void stampa_matrice(int **mat, int righe, int colonne) {
    int i, j;

    for (i = 0; i < righe; i++) {
        for (j = 0; j < colonne; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}
```

9.3.1 Esempio 16: passaggio di una matrice a una funzione

Il programma seguente, come l'esempio 10, dichiara una matrice da allocare dinamicamente, legge da finestra di terminale le sue dimensioni e quindi la alloca. Poi però riempie di valori la matrice e la stampa riga per riga per mezzo delle due funzioni sopra viste (`esempio_16.c`).

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

void leggi_matrice(int **mat, int righe, int colonne) {
    int i, j, val;

    for (i = 0; i < righe; i++)
        for (j = 0; j < colonne; j++) {
            scanf("%d", &val);
            mat[i][j] = val;
        }
}

void stampa_matrice(int **mat, int righe, int colonne) {
    int i, j;

    for (i = 0; i < righe; i++) {
        for (j = 0; j < colonne; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

int main() {
    int** M;
    int i, n, m;

    printf("Numero di righe: ");
    scanf("%d", &n);
    printf("Numero di colonne: ");
    scanf("%d", &m);

    M = (int**)malloc(n*sizeof(int *));
    for (i = 0; i < n; i++)
        M[i] = (int*)malloc(m*sizeof(int));

    leggi_matrice(M, n, m);
    stampa_matrice(M, n, m);
    return 0;
}
```

9.4 Funzioni ricorsive

Se una funzione chiama se stessa (ossia, nel corpo della funzione appare una o più chiamate alla funzione stessa) la funzione si dice *funzione ricorsiva*.

Le funzioni ricorsive sono utilissime per risolvere numerosi problemi, in particolare su strutture dinamiche quali liste, alberi o grafi.

Come ampiamente discusso a lezione, un aspetto fondamentale del progetto di una funzione ricorsiva è la sua condizione di terminazione (detta anche caso base) perché si deve essere certi che esso (o uno di essi) venga sempre incontrato. In caso contrario, infatti, la funzione continuerebbe a chiamare se stessa, impegnando nuova memoria sullo stack di sistema ad ogni chiamata e causando in brevissimo tempo la terminazione dell'elaborazione per esaurimento della memoria disponibile.

9.4.1 Esempio 17: calcolo del fattoriale tramite funzione ricorsiva

Riscriviamo il programma dell'esempio 15 (che legge un intero dalla finestra del terminale e stampa il fattoriale di tale valore) organizzando il calcolo del fattoriale per mezzo di una funzione ricorsiva (`esempio_17.c`).

```
#include <stdio.h>

long fattoriale (long n) {
    if (n == 0) return 1;  //caso base
    return n*fattoriale(n - 1);
}

int main() {
    int val;
    long val_fatt;

    printf("Inserisci valore: ");
    scanf("%d", &val);

    val_fatt = fattoriale(val);
    printf("Il fattoriale di %d vale %ld\n", val, val_fatt);
    return 0;
}
```

Si noti che il caso base (l'unico) si incontra quando `n` è zero.

Nell'ipotesi che la funzione venga chiamata con un valore legittimo del parametro attuale (ossia un valore intero non negativo) vi è la certezza che il caso base sia incontrato.

Viceversa, se si chiamasse la funzione passandole un valore negativo il caso base non verrebbe mai incontrato con conseguente non terminazione della funzione ricorsiva.

Nell'ambito di questo corso si presuppone che i valori passati a ogni funzione (ricorsiva o no) siano legittimi, ossia congruenti con quanto la funzione si aspetta di ricevere.

9.5 Prototipi di funzione

Una funzione, come abbiamo detto, deve essere dichiarata prima di poter essere utilizzata (ossia prima che in una istruzione appaia una chiamata alla funzione).

Ciò però in alcuni casi non è possibile (come ad esempio quando una funzione A chiama una funzione B e la funzione B chiama la funzione A – ricorsione indiretta) oppure se si desidera che la funzione `main()` sia la prima ad essere dichiarata.

In questi casi si deve inserire:

- il *prototipo della funzione* prima del punto in cui la funzione viene chiamata;
- la dichiarazione della funzione dopo tale punto.

Il prototipo della funzione è costituito dalla intestazione della funzione (ossia la prima riga della sua dichiarazione, comprensiva dei parametri formali racchiusi fra parentesi tonde) terminata con un punto e virgola. L'esempio 17 diviene:

```
#include <stdio.h>

long fattoriale (long val);    //prototipo della funzione fattoriale

int main() {
   int n;
   long n_fatt;
   printf("Inserisci valore: ");
   scanf("%d", &n);

   n_fatt = fattoriale(n);
   printf("Il fattoriale di %d vale %ld\n", n, n_fatt);
   return 0;
}

long fattoriale (long val) {
   if (val == 0) return 1;
   return val*fattoriale(val - 1);
}
```

In effetti è una buona (e diffusa) pratica programmativa strutturare il codice di un programma C secondo questo schema di precedenza delle sue varie parti (ove presenti):

1. inclusioni (direttive `include`);
2. definizioni di costanti (direttive `define`) - che peraltro noi non usiamo;
3. variabili globali – che come detto è meglio non usare;
4. dichiarazioni di tipi – vedremo fra breve;
5. prototipi delle funzioni;
6. funzione `main()`;
7. tutte le altre funzioni.

Nota: l'ordine di elencazione dei prototipi di funzione non deve necessariamente essere lo stesso con il quale sono poi dichiarate le funzioni stesse.

9.6 Esempi ed esercizi

Esempio 18: trovare il minimo di un vettore tramite funzione ricorsiva

Scriviamo una funzione ricorsiva che trova il minimo elemento contenuto in un vettore di `n` elementi (`esempio_18.c`).

Idea ricorsiva:

- tale minimo è il minimo fra:
  - l'n-esimo elemento;
  - il minimo elemento contenuto nel sottovettore costituito dai primi `n-1` elementi (definizione ricorsiva);
- caso base: il minimo di un vettore di 1 elemento è l'elemento stesso.

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

int minimo_vettore(int *vet, int dim);
int minimo(int a, int b);
void leggi_vettore(int *vet, int dimensione);

int main() {
    int* V;
    int i, n, min;
    printf("Numero di elementi: ");
    scanf("%d", &n);

    V = (int*)malloc(n*sizeof(int));
    leggi_vettore(V, n);
    min = minimo_vettore (V, n);            //prima chiamata alla funzione ricorsiva
    printf("Il minimo vale: %d\n", min);
    return 0;
}
int minimo_vettore(int *vet, int dim) {
    int last = dim - 1;
    if (dim == 1) return vet[0];                                //caso base
    else return minimo(vet[last], minimo_vettore(vet, dim-1));  //chiamata ricorsiva
}
int minimo(int a, int b) {
    if (a < b) return a;
    else return b;
}
void leggi_vettore(int *vet, int dimensione) {
    int i, val;
    for (i = 0; i < dimensione; i++) {
           scanf("%d", &val);
           vet[i] = val;
    }
}
```

Esempio 19: calcolare la somma degli elementi di un vettore tramite funzione ricorsiva

Scriviamo una funzione ricorsiva che calcola la somma degli elementi contenuti in un vettore di `n` elementi (`esempio_19.c`).

Idea ricorsiva:

- tale somma si ottieme mediante addizione di:
  - l'n-esimo elemento;
  - la somma degli elementi contenuti nel sottovettore costituito dai primi `n-1` elementi (definizione ricorsiva);
- caso base: tale somma per un vettore di 1 elemento è l'elemento stesso.

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

int somma_vettore(int *vet, int dim);
void leggi_vettore(int *vet, int dimensione);

int main() {
   int* V;
   int i, n, somma;

   printf("Numero di elementi: ");
   scanf("%d", &n);

   V = (int*)malloc(n*sizeof(int));
   leggi_vettore(V, n);

   somma = somma_vettore (V, n);             //prima chiamata alla funzione ricorsiva
   printf("La somma vale: %d\n", somma);
   return 0;
}

int somma_vettore(int *vet, int dim) {
   int last = dim - 1;
   if (dim == 1) return vet[0];                              //caso base
   else return vet[last] + somma_vettore(vet, dim-1);        //chiamata ricorsiva
}

void leggi_vettore(int *vet, int dimensione) {
   int i, val;
   for (i = 0; i < dimensione; i++) {
          scanf("%d", &val);
          vet[i] = val;
   }
}
```

Esempio 20: verificare se un vettore è palindromo

Scriviamo una funzione ricorsiva che verifica se un vettore di `n` elementi è palindromo,ossia la sequenza dei suoi elementi è identica se letta da sinistra a destra o da destra a sinistra (`esempio_20.c`).

Idea ricorsiva:

- un vettore è palindromo se:
  - il primo e l'ultimo elemento sono uguali e se:
  - il sottovettore costituito dagli elementi che vanno dal secondo al penultimo compresi è palindromo (definizione ricorsiva);
- casi base (sono due distinti):
  - un vettore di un solo elemento è palindromo;
  - un vettore di due elementi è palindromo se i due elementi sono uguali.

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

int palindromo(int *vet, int inizio, int fine);
void leggi_vettore(int *vet, int dimensione);

int main() {
    int* V;
    int i, n, palin;
    printf("Numero di elementi: ");
    scanf("%d", &n);

    V = (int*)malloc(n*sizeof(int));
    leggi_vettore(V, n);
    palin = palindromo(V, 0, n-1);            //prima chiamata alla funzione ricorsiva
    printf("Palindromo: %d\n", palin);
    return 0;
}

int palindromo(int *vet, int inizio, int fine) {
    if (inizio == fine) return 1;                                          //primo caso base
    if (inizio == fine-1) return(vet[inizio]==vet[fine]);                  //secondo caso base
    else return ((vet[inizio]==vet[fine]) && palindromo(vet,inizio+1,fine-1)); //chiamata ricorsiva
}
void leggi_vettore(int *vet, int dimensione) {
    int i, val;
    for (i = 0; i < dimensione; i++) {
           scanf("%d", &val);
           vet[i] = val;
    }
}
```

Si noti che, dentro la funzione `palindromo()`, la chiamata ricorsiva non viene eseguita se gli estremi del vettore in esame sono diversi fra loro perché in tal caso l'espressione:

```
(vet[inizio]==vet[fine])
```

risulta falsa e quindi la valutazione dell'espressione logica complessiva, che è un AND, si arresta immediatamente senza valutare ciò che si trova a destra dell'AND, ossia la chiamata ricorsiva della funzione stessa.

10 Il problema della ricerca

Nel file `ricerca.c` scaricabile dal sito del corso sono contenute tutte le funzioni sotto descritte. Nota: per mantenere la conformità con lo pseudocodice riportato nelle dispense, i vettori contengono (n + 1) elementi, dei quali si utilizzano solo quelli con indici da 1 a n.

Tutte le funzioni restituiscono::

- l'indice dell'elemento se esso è presente;
- il valore `-1` altrimenti.

10.1 Ricerca sequenziale iterativa

La seguente funzione implementa la ricerca sequenziale di un valore in un vettore.

```
int ricerca_sequenziale(int *A, int v, int n) {
   int i;

   for (i = 1; i <= n; i++)
      if (A[i] == v) return i;
   return -1;
}
```

10.2 Ricerca binaria iterativa

La seguente funzione implementa la ricerca binaria di un valore in un vettore.

```
int ricerca_binaria(int *A, int v, int n) {
   int a = 1;
   int b = n;
   int m = (a + b)/2;

   while (A[m] != v) {
      if (A[m] > v)
         b = m - 1;
      else a = m + 1;
      if (a > b) return -1;
      m = (a + b)/2;
   }
   return m;
}
```

10.3 Ricerca sequenziale ricorsiva

La seguente funzione ricorsiva implementa la ricerca sequenziale di un valore in un vettore.

```
int ricerca_sequenziale_ricorsiva(int *A, int v, int n) {

   if (A[n] == v) return n;
   if (n == 1)
      return -1;
   else
      return ricerca_sequenziale_ricorsiva(A, v, n - 1);
}
```

La chiamata nel `main()` sarà:

```
indice = ricerca_sequenziale_ricorsiva(V, valore, n);
```

10.4 Ricerca binaria ricorsiva

La seguente funzione ricorsiva implementa la ricerca binaria di un valore in un vettore.

```
int ricerca_binaria_ricorsiva(int *A, int v, int i_min, int i_max) {
   int m;

   if (i_min > i_max) return -1;

   m = (i_min + i_max)/2;
   if (A[m] == v) return m;
   if (A[m] > v) return ricerca_binaria_ricorsiva(A, v, i_min, m-1);
   else return ricerca_binaria_ricorsiva(A, v, m+1, i_max);
}
```

La chiamata nel `main()` sarà:

```
indice = ricerca_binaria_ricorsiva(V, valore, 1, n);
```

11 Algoritmi di ordinamento

11.1 Insertion sort

La seguente funzione implementa l'insertion sort visto a lezione.

NOTA: come nel caso della ricerca, anche in questo e nei successivi algoritmi di ordinamento i vettori contengono (n + 1) elementi, dei quali si utilizzano solo quelli con indici da 1 a n. Questo per mantenere la totale conformità con i corrispondenti algoritmi espressi tramite pseudocodice nelle dispense

```
void insertionsort(int *A, int n){
   int i, j, x;

   for (j = 2; j <= n; j++) {                    // 1
      x = A[j];                                  // 2
      i = j - 1;                                 // 3
      while ((i > 0) && (A[i] > x)) {            // 4
         A[i + 1] = A[i];                        // 5
         i = i - 1;                              // 6
      }
      A[i + 1] = x;                              // 7
   }
}
```

Nel file `insertion_sort.c` scaricabile dal sito del corso è realizzato un programma che:

- legge un vettore da tastiera mediante la funzione `leggi_vettore()`;
- lo ordina mediante la funzione `insertion_sort()` sopra illustrata;
- lo stampa mediante la funzione `stampa_vettore()` dopo averlo ordinato.

11.2 Selection sort

La seguente funzione implementa il selection sort visto a lezione.

```
void selectionsort(int *A, int n){
   int i, j, m, temp;

   for (i = 0; i <= n - 2; i++) {
      m = i + 1;
      for (j = i + 2; j <= n; j++)
         if (A[j] < A[m])
            m = j;
      temp = A[m];   //scambia
      A[m] = A[i+1]; //vet[m] con
      A[i+1] = temp; //vet[i+1]
   }
}
```

Nel file `selection_sort.c` scaricabile dal sito del corso è realizzato un programma che:

- legge un vettore da tastiera mediante la funzione `leggi_vettore()`;
- lo ordina mediante la funzione `selection_sort()` sopra illustrata;
- lo stampa mediante la funzione `stampa_vettore()` dopo averlo ordinato.

11.3 Bubble sort

La seguente funzione implementa il bubble sort visto a lezione.

```
void bubblesort(int *A, int n){
   int i, j, temp;

   for (i = 1; i <= n; i++)
      for (j = n; j >= i + 1; j--)
         if (A[j] < A[j - 1]) {
            temp = A[j];             //scambia
            A[j] = A[j - 1];         //vet[j] con
            A[j - 1] = temp;         //vet[j-1]
         }
}
```

Nel file `bubble_sort.c` scaricabile dal sito del corso è realizzato un programma che:

- legge un vettore da tastiera mediante la funzione `leggi_vettore()`;
- lo ordina mediante la funzione `bubbler_sort()` sopra illustrata;
- lo stampa mediante la funzione `stampa_vettore()` dopo averlo ordinato.

11.4 Mergesort

Il codice C dell'algoritmo mergesort ricorsivo, operante su un vettore vettore `A`. è il seguente. I parametri `indice_primo` e `indice_ultimo` identificano la porzione di vettore su cui la chiamata ha effetto.

```
void mergesort(int *A, int indice_primo, int indice_ultimo){
   int indice_medio;

   if (indice_primo < indice_ultimo) {
      indice_medio = (indice_primo + indice_ultimo)/2;
      mergesort(A, indice_primo, indice_medio);
      mergesort(A, indice_medio + 1, indice_ultimo);
      fondi(A, indice_primo, indice_medio, indice_ultimo);
   }
}
```

La chiamata nel `main()` sarà:

```
mergesort(V, 1, n);
```

La funzione che fonde due sequenze ordinate è la seguente (si noti l'allocazione dinamica e la successiva deallocazione del vettore `B` d'appoggio).

```
void fondi(int *A, int indice_primo, int indice_medio, int indice_ultimo){
   int *B;
   int i, j, k;

   B = (int*)malloc((indice_ultimo - indice_primo + 2)*sizeof(int));

   i = indice_primo;                                                          //  1
   j = indice_medio + 1;                                                      //  2
   k = 1;                                                                     //  3

   while ((i <= indice_medio) && (j <= indice_ultimo)) {                      //  4
      if (A[i] < A[j]) {                                                      //  5
         B[k] = A[i];                                                         //  6
         i = i + 1;                                                           //  7
      }
      else {                                                                  //  8
         B[k] = A[j];                                                         //  9
         j = j + 1;                                                           // 10
      }
      k = k + 1;                                                              // 11
   }
   while (i <= indice_medio) { //il primo sottovettore non e' terminato          12
      B[k] = A[i];                                                            // 13
      i = i + 1;                                                              // 14
      k = k + 1;                                                              // 15
   }
   while (j <= indice_ultimo) { //il secondo sottovettore non e' terminato       16
      B[k] = A[j];                                                            // 17
      j = j + 1;                                                              // 18
      k = k + 1;                                                              // 19
   }

   //copia B[1..k-1] su A[indice_primo..indice_ultimo]                           20
   for (i = 1; i < k; i++){
      A[indice_primo + i - 1] = B[i];
   }
   free(B);
}
```

Nel file `merge_sort.c` scaricabile dal sito del corso è realizzato un programma che:

- legge un vettore da tastiera mediante la funzione `leggi_vettore()`;
- lo ordina mediante la funzione ricorsiva `mergesort()` sopra illustrata;
- lo stampa mediante la funzione `stampa_vettore()` dopo averlo ordinato.

11.5 Quicksort

Il codice C dell'algoritmo ricorsivo quicksort, operante su un vettore A, è il seguente.

```
void quicksort(int *A, int indice_primo, int indice_ultimo){
   int indice_medio;

   if (indice_primo < indice_ultimo) {
      indice_medio = partiziona(A, indice_primo, indice_ultimo);
      quicksort(A, indice_primo, indice_medio);
      quicksort(A, indice_medio + 1, indice_ultimo);
   }
}
```

La chiamata nel `main()` per un vettore `V` di `n` elementi sarà:

```
quicksort(V, 1, n);
```

La funzione di partizionamento del vettore è la seguente.

```
int partiziona(int *A, int indice_primo, int indice_ultimo){
   int indice_pivot = indice_primo; //scelta arbitraria                    //  1
   int i, j, valore_pivot, temp;
   valore_pivot = A[indice_pivot];                                         //  2
   i = indice_primo - 1;                                                   //  3
   j = indice_ultimo + 1;                                                  //  4

   while (1) {                                                             //  5
      do {                                                                 //  6
         j = j - 1;                                                        //  7
      } while (A[j] > valore_pivot);                                       //  8
      do {                                                                 //  9
         i = i + 1;                                                        // 10
      } while (A[i] < valore_pivot);                                       // 11
      if (i < j) {                                                         // 12
         temp = A[i];  A[i] = A [j]; A[j] = temp; //scambia A[i] con A[j]  // 13
      }
      else                                                                 // 14
         return j;                                                         // 15
   }
}
```

Nel file `quick_sort.c` scaricabile dal sito del corso è realizzato un programma che:

- legge un vettore da tastiera mediante la funzione `leggi_vettore()`;
- lo ordina mediante la funzione `quicksort()` sopra illustrata;
- lo stampa mediante la funzione `stampa_vettore()` dopo averlo ordinato.

11.6 Heapsort

Le seguenti funzioni implementano quelle corrispondenti illustrate sulle dispense del corso in relazione all'heapsort.

Le piccole diffrenze nel codice sono dovute alla necessità di evitare l'uso di variabili globali (nel caso in questione le variabili globali `n` e `heap_size` utilizzate nello pseudocodice delle dispense). Nelle funzioni sotto riportate il ruolo di tali variabili globali è svolto dai parametri formali `n` e `heap_size`.

Funzioni `left()` e `right()`

Si ricorda la relazione posizionale fra elementi dell'albero:

- la radice occupa la posizione di indice 1;
- i figli sinistro e destro del nodo in posizione i si trovano rispettivamente nelle posizioni 2i e 2i + 1.

```
int left(int index) {
   return 2*index;
}

int right(int index) {
   return 2*index + 1;
}
```

Funzione `heapify()`

```
void heapify(int *A, int i, int heap_size) {
   int indice_massimo, L, R, temp;
   L = left(i);
   R = right(i);
   if ((L <= heap_size)&&(A[L] > A[i]))
      indice_massimo = L;
   else
      indice_massimo = i;
   if ((R <= heap_size)&&(A[R] > A[indice_massimo]))
      indice_massimo = R;
   if (indice_massimo != i) {
      temp = A[i];                    //scambia
      A[i] = A[indice_massimo];       //A[i] con
      A[indice_massimo] = temp;       //A[indice_massimo]
      heapify(A, indice_massimo, heap_size);
   }
}
```

Funzione `buildheap()`

```
void buildheap(int *A, int n) {
   int i;

   for (i = n/2; i >= 1; i--)
      heapify(A, i, n);
}
```

Funzione `heapsort()`

```
void heapsort(int *A, int n){
   int i, temp;
   int heap_size = n;

   buildheap(A, n);
   for (i = n; i >= 2; i--) {
      temp = A[i];   //scambia
      A[i] = A[1];   //A[i] con
      A[1] = temp;   //A[1]
      heap_size = heap_size - 1;
      heapify(A, 1, heap_size);
   }
}
```

La chiamata nel `main()` per un vettore `V` di `n` elementi sarà:

```
heapsort(V, n);
```

Nel file `heap_sort.c` scaricabile dal sito del corso è realizzato un programma che:

- legge un vettore da tastiera mediante la funzione `leggi_vettore()`;
- lo ordina mediante la funzione `heapsort()` sopra illustrata;
- lo stampa mediante la funzione `stampa_vettore()` dopo averlo ordinato.

12 Strutture dati

Vediamo come implementare in C alcune delle strutture dati viste a lezione:

- liste semplici, liste doppie, code, pile;
- alberi.

Verranno illustrate implementazioni basate su:

- allocazione dinamica delle opportune componenti elementari (*elementi* di liste, code, pile e *nodi* degli alberi);
- gestione dei collegamenti fra componenti elementati tramite puntatori.

12.1 Definizione di tipi in C

In C è possibile definire *tipi di dati* complessi in funzione delle proprie esigenze. Tali tipi di dati possono consistere di una moltitudine di altri tipi dati (elementari o no), opportunamente organizzati.

Per i nostri scopi serve l'uso combinato di due parole chiave del linguaggio:

- `struct` che serve a definire la struttura dati voluta (spesso detta *record*), contenente al suo interno i dati opportuni, organizzati ciascuno in un *campo del record*;
- `typedef` che serve a definire un *tipo di dato* sulla base di strutture dati (`struct`) e/o tipi di dati precedentemente definiti.

Ad esempio, vogliamo definire un elemento di lista semplice che contenga:

- un campo `key` contenente un valore intero (la *chiave* dell'elemento di lista);
- un campo `next` contenente un puntatore ad un altro elemento della lista.

Il seguente frammento di codice C definisce un struttura dati di nome `int_list`

```
struct int_list{
   int key;
   struct int_list *next;
};
```

adatta al nostro scopo, e:

```
typedef struct int_list elemento_lista;
```

dichiara questa struttura dati come un tipo di dato di nome `elemento_lista`. In realtà il funzionamento di `typedef` è il seguente: ovunque, nel successivo codice, si incontri l'identificatore del tipo (`elemento_lista`) ad esso viene sostituito ciò che appare nella dichiarazione del tipo (`struct int_list`).

E' pratica commune riunire queste due fasi in una sola, ed anche dare alla definizione del tipo lo stesso nome usato nella dichiarazione della struttura dati:

```
typedef struct int_list{
   int key;
   struct int_list *next;
} int_list;
```

Per fare riferimento ai singoli campi di una struttura dati si usano due sintassi diverse, a seconda che la struttura sia stata allocata staticamente oppure dinamicamente.

Nel primo caso si usa l'operatore . per accedere al singolo campo:

```
typedef struct int_list{
   int key;
   struct int_list *next;
} int_list;

int_list elemento; //allocazione statica

elemento.key = 10;
elemento.next = NULL;
```

Nel secondo caso (che a noi interessa di più) si usano il puntatore all'elemento e l'operatore -> per accedere al singolo campo dell'elemento stesso:

```
typedef struct int_list{
   int key;
   struct int_list *next;
} int_list;

int_list *ptr_elemento;
ptr_elemento = (int_list*)malloc(sizeof(int_list)); //allocazione dinamica

ptr_elemento->key = 10;
ptr_elemento->next = NULL;
```

ATTENZIONE: se si accede alla struttura attraverso un puntatore a puntatore bisogna ricordarsi di circondare la deferenziazione del puntatore a puntatore con le parentesi tonde:

```
int_list *ptr_elemento;
ptr_elemento = (int_list*)malloc(sizeof(int_list)); //allocazione dinamica
int_list **ptr_ptr_elemento = &ptr_elemento; //definiamo il puntatore a puntatore

(*ptr_ptr_elemento)->key = 10;
(*ptr_ptr_elemento)->next = NULL;
```

12.2 Liste semplici

12.2.1 Inserimento in una lista

La seguente funzione `insert()` riceve nei parametri un numero intero e un puntatore a una lista, quindi:

- alloca un nuovo elemento di lista;
- vi scrive il numero intero nel campo `key`;
- inserisce l'elemento in testa alla lista della quale ha ricevuto il puntatore;
- restituisce il puntatore alla lista (che si modifica sempre dopo l'inserzione).

```
int_list* insert(int_list *head, int x){
   int_list *p;

   //creo ed inizializzo il nuovo elemento della lista
   p = (int_list*)malloc(sizeof(int_list));
   p->key = x;
   p->next = NULL;

   if (head != NULL) p->next = head; //la lista non e' vuota, attacco in testa
   return p; //il puntatore d'accesso alla lista e' cambiato
}
```

Si noti che, poiché si attacca in testa, il primo elemento della lista è l'ultimo ad essere stato inserito.

12.2.2 Scansione di una lista

La seguente funzione `stampa_lista()` è un esempio di come si scandisce una lista, effettuando una specifica operazione (in questo caso la stampa della chiave) per ciascun elemento della lista.

```
void stampa_lista(int_list *head){
   int_list *p;

   p = head;
   while (p != NULL) {        //ciclo di scansione della lista
      printf("%d ", p->key); //stampa dell'elemento
      p = p->next;            //passo al prossimo elemento di lista
   }
   return;
}
```

La seguente è una formulazione ricorsiva della medesima funzionalità:

```
void stampa_lista_ric(int_list *head){

   if (head == NULL) return; //caso base
   else {
      stampa_lista_ric(head->next); //chiamata ricorsiva sul resto della lista
      printf("%d ", head->key);    //stampa dell'elemento corrente
   }
   return;
}
```

Si noti che questa funzione stampa gli elementi a partire da quello che si trova in fondo alla lista, ossia nello stesso ordine in cui sono stati inseriti. Scambiando l'ordine delle due istruzioni dentro l'`else` si rovescia l'ordine di stampa.

12.2.3 Esempi ed esercizi su liste

Esempio 21: creazione e stampa di una lista

Scriviamo un programma che, utilizzando le funzioni sopra definite, acquisisce numeri interi non negativi da tastiera, li inserisce in una lista che poi scandisce (sia con la funzione iterativa che con quella ricorsiva) per stamparli. Il ciclo di lettura si interrompe inserendo un numero negativo. Nel file `esempio_21.c` è contenuto il programma completo.

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

typedef struct int_list {
   int key;
   struct int_list *next;
} int_list;

int_list* insert(int_list *head, int x);
void stampa_lista(int_list *head);
void stampa_lista_ric(int_list *head);

int main() {
   int valore;
   int_list *list_ptr = NULL; //puntatore alla lista
   while (1) {
      scanf("%d", &valore);
      if (valore < 0) break;
      list_ptr = insert(list_ptr, valore);
   }
   stampa_lista(list_ptr);
   stampa_lista_ric(list_ptr);
   return 0;
}
//***************** Inserire qui il codice delle tre funzioni ****************************
```

Esempio 22: massimo di una lista

La seguente funzione restituisce il puntatore all'elemento massimo contenuto in una lista.

```
int_list* massimo(int_list *head){
    int_list *p, *max_p;
    int max;

    if (head == NULL) return NULL; //lista vuota
    p = head;
    max_p = p;
    max = p->key;
    while (p != NULL) {      //scansione della lista
        if (p->key > max) {   //ho trovato un nuovo massimo
            max = p->key;     //aggiorno il massimo
            max_p = p;        //aggiorno il puntatore al massimo
        }
        p = p->next;          //passo al prossimo elemento
    }
    return max_p;             //restituisco il puntatore al massimo
}
```

Nel file `esempio_22.c` è contenuto il programma completo.

Esempio 23: eliminazione di un elemento da una lista

La seguente funzione ricorsiva elimina da una lista il primo elemento incontrato (se esso esiste) avente chiave uguale al valore passato e restituisce il puntatore alla lista stessa.

```
int_list* elimina_ric(int_list *head, int val){
    int_list *p;

    if (head == NULL) return NULL; //caso base, lista vuota
    if ((head->key == val)) { //altro caso base:
        p = head->next;        //l'elemento da eliminare
        free(head);            //e' il primo della lista
        return p;
    }
    else {                     //devo proseguire la ricerca
        head->next = elimina_ric(head->next, val); //elimino ricorsivamente dal resto della lista
        return head;                               //restituisco il puntatore alla lista
    }
}
```

La versione iterativa è più complicata e meno intuitiva:

```
int_list* elimina(int_list *head, int val){
    int_list *p, *next_p;

    if (head == NULL) return NULL; //lista vuota
    if ((head->key == val)) {     //l'elemento da eliminare
        p = head->next;            //e' il primo della lista
        free(head);
        return p;
    }

    p = head;
    while (p->next != NULL) { //scansione della lista
        next_p = p->next;
        if (next_p->key == val) {  //ho trovato l'elemento da eliminare
            p->next = next_p->next; //lo cortocircuito
            free(next_p);           //lo elimino dalla memoria
            return head;            //ho terminato, restituisco il puntatore alla lista
        }
        p = p->next;       //passo al prossimo elemento
    }
    return head;          // ho terminato, restituisco il puntatore alla lista
}
```

Nel file `esempio_23.c` è contenuto il programma completo che utilizza entrambe le funzioni.

Esempio 24: eliminazione di tutte le occorrenze di un elemento da una lista

La seguente funzione ricorsiva elimina da una lista tutte le occorrenze di elementi aventi chiave uguale al valore passato e restituisce il puntatore alla lista stessa.

```
int_list* elimina_occorrenze(int_list *head, int val){
    int_list *p;

    if (head == NULL) return NULL; //caso base, lista vuota
    head->next = elimina_occorrenze(head->next, val); //elimina ricorsivamente dal resto della lista
    if ((head->key == val)) { //l'elemento da eliminare
        p = head->next;        //e' il primo della lista
        free(head);
        return p;
    }
    else return head;          //restituisco il puntatore alla lista
}
```

Nel file `esempio_24.c` è contenuto il programma completo.

Esercizio 08

Dato il puntatore `L` ad una lista ed un valore `x`, inserire un nuovo elemento con chiave `x` in fondo alla lista (anziché in testa).

Esercizio 09

Scomporre una lista `L` in due liste, `L_p` ed `L_d`, contenenti rispettivamenti gli elementi di `L` pari e dipsari (senza creare copie degli elementi).

Esercizio 10

Date due liste `L1` ed `L2` fonderle in un'unica lista senza creare copie degli elementi.

Esercizio 11

Date due liste `L1` ed `L2` ordinate in senso crescente fonderle in un'unica lista ordinata in senso decrescente senza creare copie degli elementi.

Esercizio 12

Data una lista `L` ed chiave `x` (che si assume contenuta in un elemento di `L`), trovare il predecessore di quell'elemento in `L`.

Esercizio 13

Dato il puntatore `L` ad una lista ed un valore `x`, restituire il puntatore al primo elemento contenente `x` se esso esiste, `NULL` altrimenti. Progettare sia la versione iterativa sia quella ricorsiva.

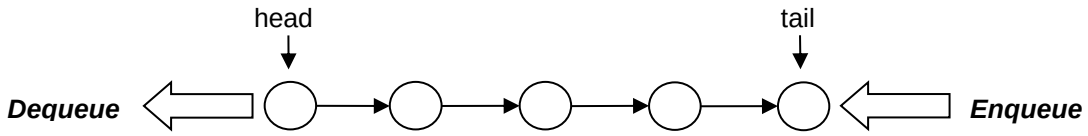
Esercizio 14

Modificare la funzione iterativa `int_list* elimina(int_list *head, int val)` in modo che elimini tutte le occorrenze dell'elemento e non solo la prima.

12.3 Code

Come abbiamo visto a lezione le code sono strutture FIFO. Sono caratterizzate da due puntatori:

- `head`, puntatore all'estremità dalla quale si estraggono gli elementi;
- `tail`, puntatore all'estremità nella quale si inseriscono gli elementi.



e da due operazioni primitive:

- `enqueue()` per inserire un elemento;
- `dequeue()` per estrarre un elemento.

Per via del fatto che entrambe le operazioni possono modificare tutt'e due i puntatori `head` e `tail`, non è possibile restituirli come valore di ritorno delle funzioni. Perciò è necessario passare alle funzioni i puntatori `head` e `tail` per riferimento. In altre parole, alle funzioni vanno passati gli indirizzi dei due puntatori: i parametri formali sono di tipo puntatore a puntatore.

12.3.1 Enqueue

In questa implementazione la funzione `enqueue()` riceve nei parametri gli indirizzi dei puntatori `head` e `tail` e un numero intero, crea un elemento della coda, lo riempie col valore intero passato e lo attacca in fondo alla coda.

```
void enqueue(int_list **head, int_list **tail, int x) {
    int_list *p;

    //creo ed inizializzo il nuovo elemento della coda
    p = (int_list*)malloc(sizeof(int_list));
    p->key = x;
    p->next = NULL;

    if (*tail == NULL) {    //la coda e' vuota
        *head = p;
        *tail = p;
    }
    else {
        (*tail)->next = p;  //la coda non e' vuota, attacco in fondo alla coda
        *tail = p;          //aggiorno il puntatore al fondo della coda
    }
    return;
}
```

12.3.2 Dequeue

In questa implementazione la funzione `dequeue()` riceve nei parametri gli indirizzi dei puntatori `head` e `tail`, estrae dalla coda l'elemento che si trova in cima e restituisce il puntatore a tale elemento.

```
int_list*  dequeue(int_list **head, int_list **tail) {
   int_list *p;

   if (*head == NULL) //la coda e' vuota
      return NULL;

   p = *head;        //la coda non e' vuota, estraggo dalla testa della coda
   *head = p->next; //aggiorno il puntatore alla testa della coda
   if (*head == NULL) *tail = NULL; //se la coda e' vuota aggiorna tail
   return p;         //restituisco l'elemento estratto dalla coda
}
```

Infine, una funzione di utilità può essere scritta per verificare se una coda è vuota prima di cercare di estrarre un elemento:

```
int codavuota(int_list * head) {
   return (head == NULL);
}
```

Si noti che dentro questa funzione non si modifica il puntatore `head`, che quindi si può passare per valore (si passa la variabile `head` e non il suo indirizzo).

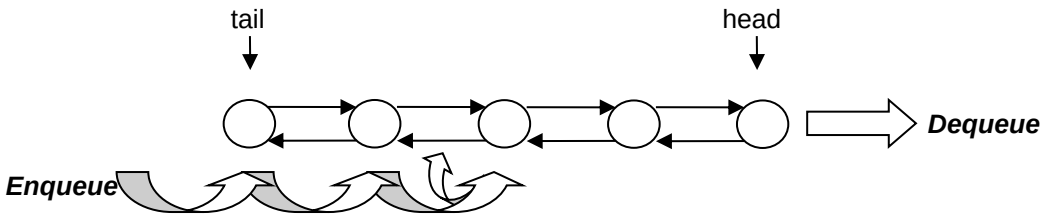
Esempio 25: inserimenti ed estrazioni da una coda

Il programma seguente (`esempio_25.c`) crea una coda mediante ripetute chiamate alla `enqueue()`, quindi estrae man mano ciascun elemento stampandoli. Essendo una coda (FIFO) l'ordine di stampa è identico all'ordine di inserzione.

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

typedef struct int_list {
    int key;
    struct int_list *next;
} int_list;

void enqueue(int_list **head, int_list **tail, int x);
int_list* dequeue(int_list **head, int_list **tail);
int codavuota(int_list *head);

int main() {
    int valore;
    int_list *queue_head = NULL; //puntatore di testa della coda
    int_list *queue_tail = NULL; //puntatore di fine della coda
    int_list *element;

    while (1) {
        scanf("%d", &valore);
        if (valore < 0) break;
        enqueue(&queue_head, &queue_tail, valore);
    }

    while (!codavuota(queue_head)) {
        element = dequeue(&queue_head, &queue_tail);
        printf("%d ", element->key);
        free(element);
    }
    printf("\n");
    return 0;
}

//***************** Inserire qui il codice delle tre funzioni ****************************
```

12.4 Code con priorità

Come abbiamo visto a lezione la coda con priorità è una variante della coda, nella quale la posizione di ciascun elemento dipende dal valore della priorità. Di conseguenza, gli elementi di una coda con priorità sono collocati in ordine crescente (o decrescente, a seconda dei casi) rispetto alla grandezza considerata come priorità.



Nella nostra implementazione:

- la priorità è il valore del campo `key`;
- gli elementi della coda sono doppiamente collegati mediante i campi `prev` e `next`;
- le due operazioni primitive le chiameremo:
  - `priority_enqueue()` per inserire un elemento;
  - `dequeue()` per estrarre un elemento.

Per via del fatto che gli elementi hanno due puntatori ciascuno è necessaria la opportuna definizione di tipo:

```
typedef struct int_pr_queue {
   int key;
   struct int_pr_queue *prev;
   struct int_pr_queue *next;
} int_pr_queue;
```

12.4.1 Priority enqueue

In questa implementazione la funzione `priority_enqueue()` è formulata ricorsivamente:

- se l'elemento da inserire ha priorità minore o uguale di quella dell'elemento puntato da `tail` allora lo inserisce come predecessore di `tail`;
- altrimenti se `tail->next` non è `NULL` chiama ricorsivamente se stessa sulla restante parte della coda con priorità; se invece `tail->next` è `NULL` inserisce l'elemento come successore di `tail`.

La funzione riceve nei parametri gli indirizzi dei puntatori `head` e `tail` e un numero intero, crea un elemento della coda, lo riempie col valore intero passato e lo inserisce, nella coda con priorità, nella posizione che gli spetta in funzione del valore della chiave.

```
void priority_enqueue(int_pr_queue **head, int_pr_queue **tail, int x) {
   int_pr_queue *p;

   //creo ed inizializzo il nuovo elemento della coda
   p = (int_pr_queue*)malloc(sizeof(int_pr_queue));
   p->key = x;
   p->prev = NULL;
   p->next = NULL;

   if ((*head == NULL)&&(*tail == NULL)) { //la coda e' vuota
      *head = p;
      *tail = p;
      return;
   }
   if (p->key <= (*tail)->key) {//trovato punto di inserzione: fra tail e
                                //predecessore di tail (che puo' essere NULL)
      p->next = *tail;          //collegamento da p verso tail (avanti)
      p->prev = (*tail)->prev; //collegamento da p verso predecessore di tail (indietro)
      (*tail)->prev = p;        //collegamento da tail verso p (indietro)
      *tail = p;                //collegamento da predecessore di tail verso p (avanti)
                               //NOTA: questo viene scritto direttamente
                               //nel campo next dell'elemento
                               //che e' stato passato per riferimento nella chiamata ricorsiva
      return;
   }
   //se arrivo qui, l'elemento da inserire e' maggiore di quello puntato da tail
   if ((*tail)->next != NULL)                    //la coda non e' finita
      priority_enqueue(head, &((*tail)->next), x);//chiamata ricorsiva sul resto della coda
   else {                  //la coda e' finita, devo attaccare qui
      (*tail)->next = p;   //collegamento da tail verso p (avanti)
      p->prev = *tail;     //collegamento da p verso tail (indietro)
      *head = p;           //sposto avanti head
   }
   return;
}
```

12.4.2 Dequeue

In questa implementazione la funzione `dequeue()` riceve nei parametri gli indirizzi dei puntatori `head` e `tail`, estrae dalla coda l'elemento che si trova in cima e restituisce il puntatore a tale elemento. L'unica differenza rispetto a quella definita per la coda è che è necessario gestire anche il campo `prev`.

```
int_pr_queue* dequeue(int_pr_queue **head, int_pr_queue **tail) {
   int_pr_queue *p;

   if (*head == NULL) //la coda e' vuota
      return NULL;
   p = *head;                 //la coda non e' vuota, estraggo dalla testa della coda
   *head = p->prev;           //aggiorno il puntatore alla testa della coda
   if (*head == NULL) *tail = NULL; //se la coda e' vuota aggiorno tail
   else (*head)->next = NULL; //altrimenti metto a NULL il campo next del nuovo head
   return p;                  //restituisco l'elemento estratto dalla coda
}
```

Infine, la solita funzione di utilità per verificare se la coda con priorità è vuota prima di cercare di estrarre un elemento:

```
int codavuota(int_pr_queue *head) {
   return (head == NULL);
}
```

Esempio 26: inserimenti ed estrazioni da una coda con priorità

Il programma seguente (`esempio_26.c`) crea una coda con priorità mediante ripetute chiamate alla `priority_enqueue()`, quindi estrae man mano ciascun elemento stampandolo. Essendo una coda con priorità l'ordine di stampa dipende dall'ordinamento delle chiavi e non dalla sequenza di inserzione.

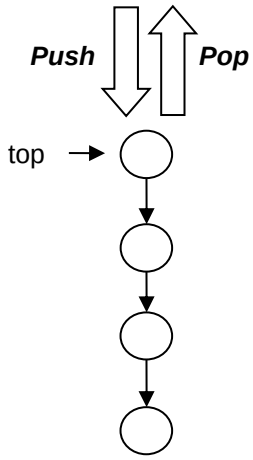
```
#include <stdio.h>
#include <stdlib.h>

typedef struct int_pr_queue {
    int key;
    struct int_pr_queue *prev;
    struct int_pr_queue *next;
} int_pr_queue;

void priority_enqueue(int_pr_queue **head, int_pr_queue **tail, int x);
int_pr_queue* dequeue(int_pr_queue **head, int_pr_queue **tail);
int codavuota(int_pr_queue *head);

int main() {
    int valore;
    int_pr_queue *queue_head = NULL; //puntatore di testa della coda
    int_pr_queue *queue_tail = NULL; //puntatore di fine della coda
    int_pr_queue *element;

    while (1) {
        scanf("%d", &valore);
        if (valore < 0) break;
        priority_enqueue(&queue_head, &queue_tail, valore);
    }

    while (!codavuota(queue_head)) {
        element = dequeue(&queue_head, &queue_tail);
        printf("%d ", element->key);
        free(element);
    }
    printf("\n");
    return 0;
}

//***************** Inserire qui il codice delle tre funzioni ****************************
```

Esercizio 15

Dato un vettore di n interi ed una coda con priorità inizialmente vuota, ordinare il vettore senza fare alcun confronto fra elementi all'interno del vettore stesso.

12.5 Pile

Come abbiamo visto a lezione le pile sono strutture LIFO. Sono caratterizzate da un solo puntatore, di norma chiamato `top`:



e da due operazioni primitive:

- `push()` per inserire un elemento;
- `pop()` per estrarre un elelento.

Nella `push()` il puntatore `top` si può passare per valore (la sua modifica si può restituire nel valore di ritorno della funzione) mentre nella `pop()` si deve passare l'indirizzo di `top` poiché la funzione deve restituire nel valore di ritorno il puntatore all'elemento estratto.

12.5.1 Push

In questa implementazione la funzione `push()` riceve nei parametri il puntatore `top` e un numero intero, crea un elemento della pila, lo riempie col valore intero passato e lo attacca in cima alla pila.

```
int_list* push(int_list *top, int x) {
   int_list *p;

   //creo ed inizializzo il nuovo elemento della pila
   p = (int_list*)malloc(sizeof(int_list));

   p->key = x;
   p->next = top;  // attacco in cima alla pila
   return p;       //aggiorno top
}
```

12.5.2 Pop

In questa implementazione la funzione `pop()` riceve nei parametri l'indirizzo del puntatore `top`, estrae dalla pila l'elemento che si trova in cima e restituisce il puntatore a tale elemento.

```
int_list* pop(int_list **top) {
   int_list *p;

   if (*top == NULL) //la pila e' vuota
      return NULL;

   p = *top;          //la pila non e' vuota, estraggo dalla cima
   *top = p->next;    //aggiorno il puntatore alla cima della
   return p;          //restituisco l'elemento estratto dalla coda
}
```

Infine, la funzione di utilità che verifica se una pila è vuota prima di cercare di estrarre un elemento:

```
int pilavuota(int_list *top) {
   return (top == NULL);
}
```

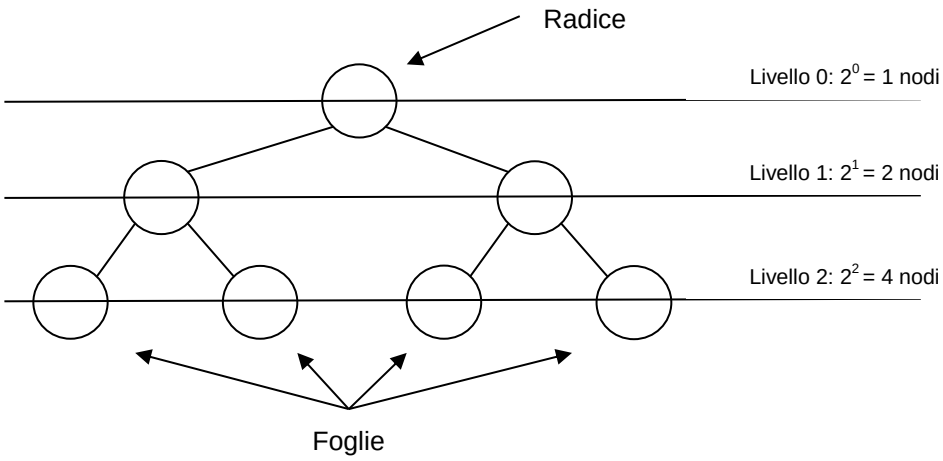
Esempio 27: inserimenti ed estrazioni da una pila

Il programma seguente (`esempio_27.c`) crea una pila mediante ripetute chiamate alla `push()`, quindi estrae man mano ciascun elemento stampandolo. Essendo una pila (LIFO) l'ordine di stampa è inverso rispetto all'ordine di inserzione.

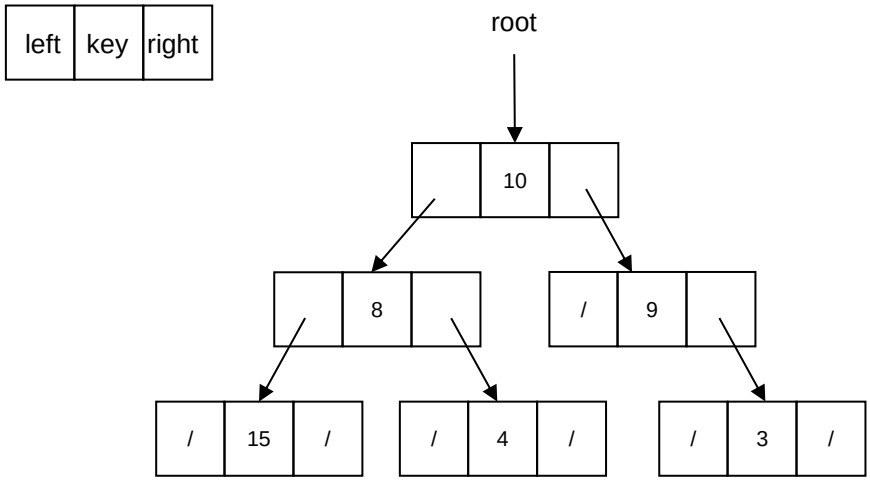
```
#include <stdio.h>
#include <stdlib.h>

typedef struct int_list {
    int key;
    struct int_list *next;
} int_list;

int_list* push(int_list *top, int x);
int_list* pop(int_list **top);
int pilavuota(int_list *top);

int main() {
    int valore;
    int_list *stack_top = NULL; //puntatore alla cima della pila
    int_list *element;

    while (1) {
        scanf("%d", &valore);
        if (valore < 0) break;
        stack_top = push(stack_top, valore);
    }
    while (!pilavuota(stack_top)) {
        element = pop(&stack_top);
        printf("%d ", element->key);
        free(element);
    }
    printf("\n");
    return 0;
}

//***************** Inserire qui il codice delle tre funzioni ****************************
```

12.6 Alberi binari

Come abbiamo visto a lezione gli alberi binari sono una sottoclasse degli alberi radicati e ordinati. Essi hanno la particolarità che ogni nodo ha al più due figli. Poiché sono alberi ordinati, i due figli di ciascun nodo si distinguono in figlio sinistro e figlio destro.



L'implementazione che illustriamo prevede che la chiave sia un valore intero ed è basata sull'uso di puntatori (nelle figure il simbolo */* rappresenta il valore `NULL`):



e si realizza con la seguente definizione di tipo:

```
typedef struct int_bin_tree {
   int key;
   struct int_bin_tree *left;
   struct int_bin_tree *right;
} int_bin_tree;
```

12.6.1 Creazione di un albero binario

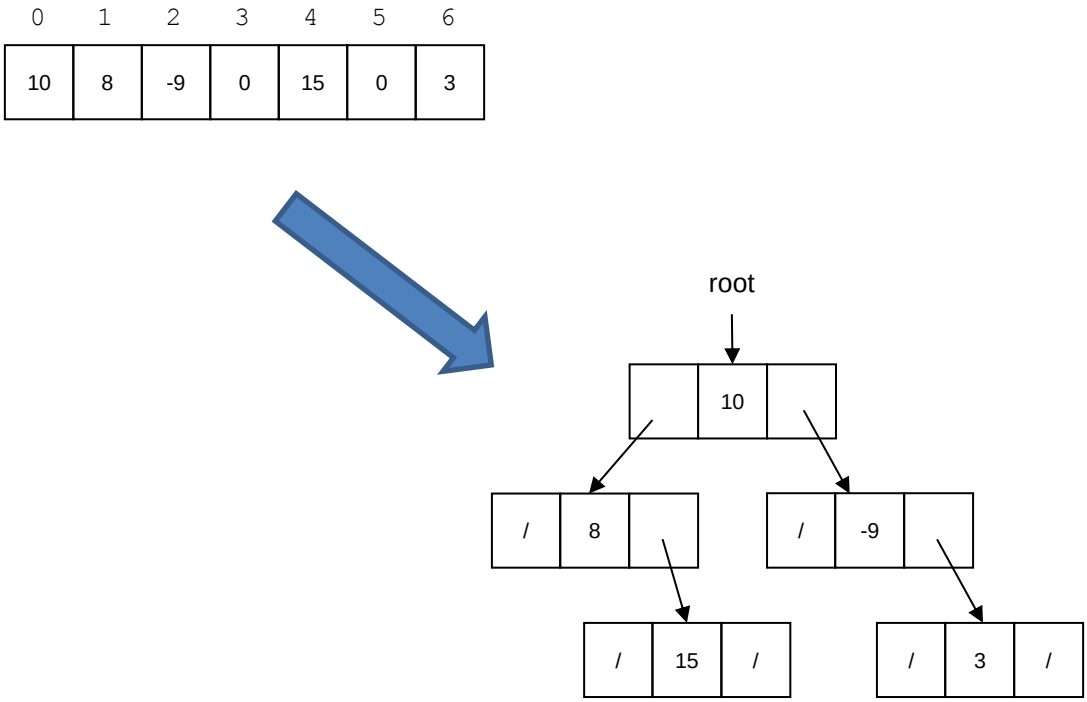
Un modo semplice per creare un albero binario, sul quale poi applicare l'algoritmo voluto, è:

- riempire un vettore che rappresenti l'albero mediante notazione posizionale;
- scandire il vettore, leggere i valori delle chiavi e creare man mano l'albero inserendovi i corrispondenti nodi.

Cpome già visto nell'ambito dell'heapsort, tenendo presente che in C gli indici degli elementi di un vettore partono da zero e non da uno la relazione posizionale fra elementi dell'albero è la seguente:

- la radice occupa la posizione di indice 0;
- i figli sinistro e destro del nodo in posizione i si trovano rispettivamente nelle posizioni 2i + 1 e 2i + 2.

Assumiamo per convenzione che il valore zero di un elemento del vettore indichi l'assenza del corrispondente nodo dell'albero.



La funzione ricorsiva che crea un albero binario a partire da un vettore è la seguente. Riceve nei parametri di ingresso il puntatore all'albero, il vettore, l'indice a partire dal quale scandire il vettore, la dimensione del vettore. Restituisce il puntatore all'albero creato.

```
int_bin_tree* from_array_to_tree(int *V, int start, int size) {
   int_bin_tree *t;

   if (start >= size) return NULL; //superata la fine del vettore

   if (V[start] == 0)return NULL;  //non c'e' nodo dell'albero da aggiungere
   else {
      //creo ed inizializzo il nuovo nodo dell'albero
      t = (int_bin_tree*)malloc(sizeof(int_bin_tree));
      t->key = V[start];
      t->left = NULL;
      t->right = NULL;
   }
   t->left = from_array_to_tree(V, 2*start+1, size);  //chiamata ricorsiva per creare
                                                      // il sottoalbero sinistro
   t->right = from_array_to_tree(V, 2*start+2, size); //chiamata ricorsiva per creare
                                                      //il sottoalbero destro
   return t;     //restituisco la radice dell'albero
}
```

Nel `main()` questa funzione andrà chiamata passando zero come valore del parametro `start`.

12.6.2 Visualizzazione di un albero binario

Senza un ambiente grafico a disposizione non è semplice visualizzare, oltre ai valori delle chiavi, anche la struttura di un albero binario.

Viene in aiuto la stampa di un albero binario mediante la notazione parentetica:

- stampa parentetica (sp) di un albero vuoto `t`:

$$sp_t: -$$

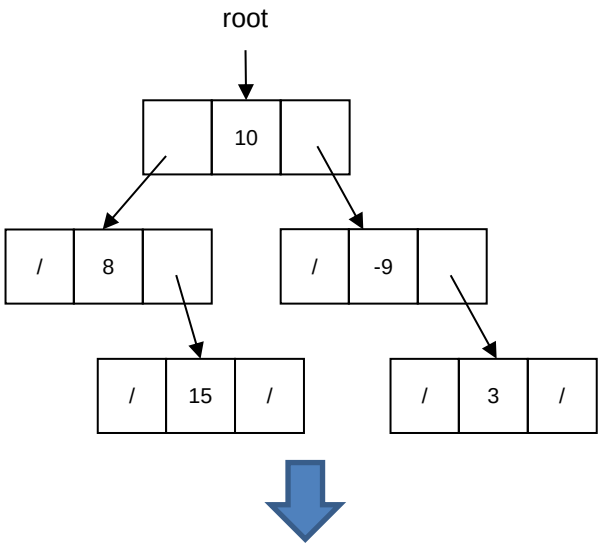
- stampa parentetica di un albero non vuoto `t`:

$$sp_t: \texttt{chiave\_della\_radice} \ (\ sp_{\texttt{sottoalbero\_sin}} \ , \ sp_{\texttt{sottoalbero\_des}} \ )$$

Si noti che questa definizione è essa stessa ricorsiva e ricorda da vicino il fatto che l'albero binario può essere facilmente definito ricorsivamente; infatti esso:

- è vuoto, oppure
- consiste di una radice e di due sottoalberi che sono anch'essi alberi binari.

Ad esempio:



La funzione ricorsiva che realizza la stampa parentetica è la seguente.

```
void stampa_parentetica(int_bin_tree *root) {

   if(root == NULL) {
      printf("-");
      return;
   }

   printf("%d(", root->key);         //stampa il valore della chiave della radice
                                     //e una parentesi aperta
   stampa_parentetica(root->left);   //stampa parentetica del sottoalbero sinistro
   printf(",");                      //stampa la virgola che separa i due sottoalberi
   stampa_parentetica(root->right);//stampa parentetica del sottoalbero destro
   printf(")");                      //stampa una parentesi chiusa
}
```

Si noti che, a differenza delle visite, la stampa parentetica identifica univocamente la struttura (cioè le relazioni padre-figlio) dell'albero binario.

Esempio 28: creazione di un albero binario e sua stampa parentetica

Il programma seguente (`esempio_28.c`) crea e riempie un vettore coi dati immessi da tastiera. Lo interpreta poi come un albero realizzato mediante la notazione posizionale e crea il corrispondente albero implementato mediante puntatori. Quindi stampa quest'ultimo mediante la notazione parentetica.

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

typedef struct int_bin_tree {
   int key;
   struct int_bin_tree *left;
   struct int_bin_tree *right;
} int_bin_tree;

int_bin_tree* from_array_to_tree(int *v, int start, int size);
void stampa_parentetica(int_bin_tree *root);

int main() {
   int i, n, valore;
   int *A;
   int_bin_tree *tree_root = NULL; //puntatore alla radice dell'albero

   printf("Inserisci la dimensione del vettore: ");
   scanf("%d", &n);

   A = (int*)malloc(sizeof(int)*n);
   for (i = 0; i < n; i++) {
      scanf("%d", &valore);
      A[i] = valore;
   }

   tree_root = from_array_to_tree(A, 0, n);
   stampa_parentetica(tree_root);
   printf("\n");
   return 0;
}

//****************   Inserire qui il codice delle due funzioni  **********************
```

12.6.3 Visite preorder, inorder e postorder

Abbiamo visto a lezione che sugli alberi binari molti problemi sono risolti per mezzo di una opportuna visita (ricorsiva), che a seconda dei casi può essere:

- visita in preordine (preorder);
- visita inordine (inorder);
- visita in postordine (postorder).

Illustriamo nel seguito un esempio per ciascuna delle tre visite, in ciascuno dei quali l'operazione compiuta su ogni nodo è semplicemente la stampa del valore della relativa chiave.

Naturalmente l'ordine di stampa prodotto sullo stesso albero sarà in generale differente per ciascuna delle tre visite. Si noti che la funzione `stampa_parentetica()` è in sostanza essa stessa una visita (preorder).

Visita (stampa) preorder

```
void stampa_preorder(int_bin_tree *root) {

   if (root == NULL) return;
   printf("%d ", root->key);       //questa e' l'operazione sul nodo
   stampa_preorder(root->left);
   stampa_preorder(root->right);
   return;
}
```

Visita (stampa) inorder

```
void stampa_inorder(int_bin_tree *root) {

   if (root == NULL) return;
   stampa_inorder(root->left);
   printf("%d ", root->key);       //questa e' l'operazione sul nodo
   stampa_inorder(root->right);
   return;
}
```

Visita (stampa) postorder

```
void stampa_postorder(int_bin_tree *root) {

   if (root == NULL) return;
   stampa_postorder(root->left);
   stampa_postorder(root->right);
   printf("%d ", root->key);       //questa e' l'operazione sul nodo
   return;
}
```

Esempio 29: creazione e stampa preorder, inorder e postorder di un albero binario

Il programma seguente (`esempio_29.c`) crea e riempie un vettore coi dati immessi da tastiera. Lo interpreta poi come un albero realizzato mediante la notazione posizionale e crea il corrispondente albero implementato mediante puntatori. Quindi stampa l'albero appena creato con tutte e tre le funzioni di stampa (preorder, inorder e postorder) sopra definite.

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

typedef struct int_bin_tree {
    int key;
    struct int_bin_tree *left;
    struct int_bin_tree *right;
} int_bin_tree;

int_bin_tree* from_array_to_tree(int *v, int start, int size);
void stampa_preorder(int_bin_tree *root);
void stampa_inorder(int_bin_tree *root);
void stampa_postorder(int_bin_tree *root);

int main() {
    int i, n, valore;
    int *A;
    int_bin_tree *tree_root = NULL; //puntatore alla radice dell'albero

    printf("Inserisci la dimensione del vettore: ");
    scanf("%d", &n);

    A = (int*)malloc(sizeof(int)*n);
    for (i = 0; i < n; i++) {
        scanf("%d", &valore);
        A[i] = valore;
    }

    tree_root = from_array_to_tree(A, 0, n);
    stampa_preorder(tree_root);
    printf("\n");
    stampa_inorder(tree_root);
    printf("\n");
    stampa_postorder(tree_root);
    printf("\n");
    return 0;
}

//***************** Inserire qui il codice delle quattro funzioni ****************************
```

Esempio 30: calcolo della somma dei valori delle chiavi di un albero binario

Il seguente programma (`esempio_30.c`) contiene una funzione ricorsiva `somma_chiavi()` che calcola la somma dei valori delle chiavi contenute in un albero binario.

Essa è basata su una delle visite. Poiché per poter calcolare la somma totale delle chiavi dell'albero si deve:

- aver calcolato la somma delle chiavi del sottalbero sinistro della radice,
- aver calcolato la somma delle chiavi del sottalbero destro della radice,
- sommare a tali valori il valore della chiave della radice,

è chiaro che è necessaria una visita postorder, dove il calcolo della somma finale è il lavoro fatto sul nodo. Ciò vale ricorsivamente per entrambi i sottoalberi della radice.

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

typedef struct int_bin_tree {
    int key;
    struct int_bin_tree *left;
    struct int_bin_tree *right;
} int_bin_tree;

int_bin_tree* from_array_to_tree(int *v, int start, int size);
int somma_chiavi(int_bin_tree *root);

int main() {
    int i, n, valore;
    int somma = 0;
    int *A;
    int_bin_tree *tree_root = NULL; //puntatore alla radice dell'albero

    printf("Inserisci la dimensione del vettore: ");
    scanf("%d", &n);

    A = (int*)malloc(sizeof(int)*n);
    for (i = 0; i < n; i++) {
        scanf("%d", &valore);
        A[i] = valore;
    }

    tree_root = from_array_to_tree(A, 0, n);
    somma = somma_chiavi(tree_root);
    printf("Somma chiavi: %d\n", somma);
    return 0;
}
```

```
int somma_chiavi(int_bin_tree *root) {

    if (root == NULL) return 0;
    return root->key + somma_chiavi(root->left) + somma_chiavi(root->right);
}

//***************** Inserire qui il codice di from_array_to_tree() ****************************
```

Esercizio 16

Scrivere un programma che stampi il massimo fra i valori contenuti nelle chiavi di un albero binario.

Esercizio 17

Scrivere un programma che stampi la somma delle chiavi che si trovano sul cammino dalla radice a una foglia avente somma delle chiavi massima.

Esercizio 18

Scrivere un programma che calcoli il numero dei nodi a distanza `k` dalla radice. Opzionalmente si stampino i relativi valori.

12.7 Alberi binari di ricerca (ABR)

Come abbiamo visto a lezione gli alberi binari di ricerca sono alberi binari nei quali:

- il valore della chiave contenuta in un nodo è maggiore o uguale al valore della chiave contenuta in ciascun nodo del suo sottalbero sinistro (se esso esiste);
- il valore della chiave contenuta in un nodo è minore al valore della chiave contenuta in ciascun nodo del suo sottalbero destro (se esso esiste).

Il tipo di dato usato è lo stesso precedentemente illustrato per gli alberi binari.

12.7.1 Ricerca in un ABR

La seguente funzione ricorsiva ricerca una chiave in un albero binario di ricerca, restituendo il puntatore al primo nodo incontrato che la contiene oppure NULL se essa non è presente.

```
int_bin_tree* ABR_search(int_bin_tree *root, int k) {

   if (root == NULL) //l'albero e' vuoto
      return NULL;
   if (root->key == k) return root;  //chiave trovata
   if (root->key > k)   //la chiave va cercata nel sottoalbero sinistro
      return ABR_search(root->left, k);
   else                 //la chiave va cercata nel sottoalbero destro
      return ABR_search(root->right, k);
}
```

12.7.2 Inserimento in un ABR

La seguente funzione ricorsiva inserisce una chiave in un albero binario di ricerca, creando un nuovo nodo contenente tale chiave e collocando il nodo nell'ABR in modo da mantenerne la proprietà.

```
int_bin_tree* ABR_insert(int_bin_tree *root, int k) {
   int_bin_tree *t;

   if (root == NULL) { //l'albero e' vuoto
      //creo ed inizializzo il nuovo nodo dell'albero
      t = (int_bin_tree*)malloc(sizeof(int_bin_tree));
      t->key = k;
      t->left = NULL;
      t->right = NULL;
      return t;
   }
```

```
   if (root->key >= k)  //il nuovo nodo va inserito nel sottoalbero sinistro
      root->left = ABR_insert(root->left, k);
   else                 //il nuovo nodo va inserito nel sottoalbero destro
      root->right = ABR_insert(root->right, k);
   return root;
}
```

Questa funzione può essere utilizzata per creare un ABR, inizialmente vuoto, mediante inserzioni successive di chiavi. La struttura finale dell'ABR dipenderà dall'ordine di inserimento delle chiavi (ad es., inserendo chiavi via via sempre maggiori si avrà un albero degenere, nel quale ogni nodo ha il solo figlio destro).

12.7.3 Ricerca di minimo e massimo in un ABR

Le seguenti funzioni ricorsive ricercano rispettivamente il minimo e il massimo delle chiavi contenute nell'ABR, restituendo il puntatore al relativo nodo.

```
int_bin_tree* ABR_min(int_bin_tree *root) {

   if (root == NULL) return NULL;       //l'albero e' vuoto

   if (root->left == NULL) return root; //minimo trovato
   else return ABR_min(root->left);     //la ricerca prosegue nel sottoalbero sinistro
}
```

```
int_bin_tree* ABR_max(int_bin_tree *root) {

   if (root == NULL) return NULL;        //l'albero e' vuoto

   if (root->right == NULL) return root; //massimo trovato
   else return ABR_min(root->right);     //la ricerca prosegue nel sottoalbero destro
}
```

12.7.4 Eliminazione di una chiave in un ABR

L'eliminazione di una chiave da un ABR è piuttosto complicata dato che, come abbiamo visto a lezione, non si può semplicemente eliminare un nodo dall'albero se esso ha entrambi i figli.

Formulando la funzione in modo ricorsivo è possibile evitare il ricorso ad un apposito campo `parent`, indispensabile invece nella formulazione iterativa riportata sulle dispense del corso.

```
int_bin_tree* ABR_delete(int_bin_tree *root, int k){
   int_bin_tree *p_app;

   if (root == NULL) return root; //l'albero e' vuoto

   if (root->key == k){                              //chiave trovata
      if (root->left == NULL && root->right == NULL){ //il nodo da eliminare è una foglia
         free(root);
         return NULL;
      }

      if (root->left == NULL){//il nodo da eliminare ha solo il sottoalbero destro
         p_app = root->right;
         free(root);
         return p_app;
      }

      if (root->right == NULL){//il nodo da eliminare ha solo il sottoalbero sinistro
         p_app = root->left;
         free(root);
         return p_app;
      }

      //il nodo ha entrambi i figli
      //metto in p_app il più piccolo elemento del sottoalbero destro di t
      p_app = ABR_min(root->right);

      //sovrascrivo la chiave del nodo
      root->key = p_app->key;

      //richiamo ricorsivamente la rimozione di p_app->key sul sottoalbero destro
      //il nodo da togliere sarà una foglia o avrà un solo figlio
      root->right = ABR_delete(root->right, p_app->key);

      return root;
   }

   //se nel nodo non ho trovato la chiave scendo ricorsivamente nell'albero
   //alla ricerca della chiave da eliminare.
   if (root->key > k) root->left = ABR_delete(root->left, k);
   if (root->key < k) root->right = ABR_delete(root->right, k);
   return root;
}
```

Esempio 31: creazione e successive modifiche di un ABR

Il programma seguente (`esempio_31.c`) crea un ABR mediante inserzioni successive delle chiavi immesse da tastiera (la sequenza di immissione termina inserendo il valore 0), quindi permette all'utente di effettuare una arbitraria sequenza di ricerche, inserzioni e/o eliminazioni. Dopo ogni operazione di modifica il programma stampa l'albero mediante la notazione parentetica. Può essere facilmente modificato aggiungendo la possibilità di cercare e stampare i valori minimo e massimo.

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

typedef struct int_bin_tree {
   int key;
   struct int_bin_tree *left;
   struct int_bin_tree *right;
} int_bin_tree;

int_bin_tree* ABR_search(int_bin_tree *root, int k);
int_bin_tree* ABR_insert(int_bin_tree *root, int k);
int_bin_tree* ABR_delete(int_bin_tree *root, int k);
int_bin_tree* ABR_min(int_bin_tree *root);
int_bin_tree* ABR_max(int_bin_tree *root);
void stampa_parentetica(int_bin_tree *root);

int main() {
   int valore, oper;
   int_bin_tree *ABR_root = NULL; //puntatore alla radice dell'albero
   int_bin_tree *node;

   while (1) {
      scanf("%d", &valore);
      if (valore == 0) break;
      ABR_root = ABR_insert(ABR_root, valore);
   }
   stampa_parentetica(ABR_root);
   printf("\n");
```

```
  while (1) {
    printf("Operazione (1: ricerca; 2: inserzione; 3: eliminazione;
      altri valori: uscita): ");
    scanf("%d", &oper);
    if ((oper != 1)&&(oper != 2)&&(oper != 3)) break;
    if (oper == 1) {
      printf("Chiave da cercare: ");
      scanf("%d", &valore);
      node = ABR_search(ABR_root, valore);
      if (node == NULL) printf("La chiave non e' presente\n");
      else printf("La chiave e' presente\n");
    }
    if (oper == 2) {
      printf("Chiave da inserire: ");
      scanf("%d", &valore);
      ABR_root = ABR_insert(ABR_root, valore);
      stampa_parentetica(ABR_root);
      printf("\n");
    }
    if (oper == 3) {
      printf("Chiave da eliminare: ");
      scanf("%d", &valore);
      ABR_root = ABR_delete(ABR_root, valore);
      stampa_parentetica(ABR_root);
      printf("\n");
    }
  }
  return 0;
}
//******* Inserire qui il codice delle funzioni *****************
```

Esercizio 19

Stampare le chiavi di un ABR in ordine crescente.

Esercizio 20

Dato un ABR, trasformarlo in un ***ABR opposto***, ossia un albero binario di ricerca nel quale sono invertiti i ruoli dei due sottoalberi. In altre parole:

- il valore della chiave contenuta in un nodo è minore del valore della chiave contenuta in ciascun nodo del suo sottalbero sinistro (se esso esiste);
- il valore della chiave contenuta in un nodo è maggiore o uguale al valore della chiave contenuta in ciascun nodo del suo sottalbero destro (se esso esiste).

12.8 Visite in ampiezza e profondità di un albero binario

12.8.1 Visita in ampiezza

Visitare un albero binario in ampiezza (ossia visitando tutti i nodi che si trovano a distanza *k* dalla radice prima di visitare i nodi che si trovano a distanza *k + 1* dalla radice) non è una operazione che si possa fare in modo semplice operando esclusivamente sull'albero, né con un approccio iterativo né con un approccio ricorsivo.

Viceversa, la tecnica che abbiamo visto a lezione per realizzare la visita in ampiezza di un grafo (utilizzo di una coda nella quale inserire i nodi quando si "scoprono" e dalla quale estrarli quando si visitano) può essere impiegata anche nel caso degli alberi binari. Dato che sull'albero non vi sono cicli, non vi è né la necessità di marcare i nodi quando vengono "scoperti" né quella di ricordarsi chi sia il predecessore di ciascun nodo (sull'albero esso è unico).

Dato che un elemento della coda d'appoggio deve contenere non più un intero ma un puntatore (a un nodo dell'albero) la definizione di tipo deve essere adeguata alla nuova situazione:

```
typedef struct ptr_list {
   int_bin_tree *tree_node;
   struct ptr_list *next;
} ptr_list;
```

Chiamiamo questo tipo `ptr_list` per ricordarci che serve a costruire una "lista di puntatori" anziché una "lista di interi" come il tipo `int_list`.

Ovviamente tale modifica richiede di ridefinire tutte le funzioni che utilizzano tale nuovo tipo, in particolare le funzioni `enqueue()`, `dequeue()` e `coda_vuota()` necessarie per gestire una coda i cui elementi siano di questo tipo. Non si riporta qui il codice di tali funzioni, che è comunque contenuto nel file `esempio_32.c` relativo all'esempio 32 del quale si illustrano nel seguito le parti principali.

La funzione che realizza la stampa di tutti i nodi di un albero binario visitandoli in ampiezza per mezzo di una coda è la seguente:

```
void stampa_in_ampiezza(int_bin_tree *root) {
   ptr_list *q_element;
   ptr_list *q_head = NULL;
   ptr_list *q_tail = NULL;

   if (root == NULL) return; //albero vuoto, non faccio nulla
   enqueue(&q_head, &q_tail, root); //inserisco in coda la radice
   while(!codavuota(q_head)) { //finche' ci sono elementi in coda
      q_element = dequeue(&q_head, &q_tail); //estraggo il prossimo nodo dalla coda
      printf("%d ", (q_element->tree_node)->key); //stampo il valore della chiave
      if ((q_element->tree_node)->left != NULL) {              //se ha figlio sinistro
         enqueue(&q_head, &q_tail, (q_element->tree_node)->left);//inserisco in coda
      }
      if ((q_element->tree_node)->right != NULL) {               //se ha figlio destro
         enqueue(&q_head, &q_tail, (q_element->tree_node)->right);//inserisco in coda
      }
      free(q_element);
   }
}
```

Si parte inserendo nella coda la radice dell'albero. Quindi si entra in un ciclo nel quale:

- si estrae un nodo dalla coda;
- si stampa il valore della chiave;
- si inseriscono nella coda i suoi figli (prima il sinistro, poi il destro) se esistono.

Il ciclo termina quando la coda non contiene più elementi.

12.8.2 Visita in profondità

Come nel caso dei grafi, se si utilizza una pila come struttura d'appoggio anziché una coda si ottiene una visita in profondità, che nella implementazione sotto illustrata coincide funzionalmente con la visita in preordine.

Anche in questo caso è necessario ridefinire le funzioni `push()` e `pop()` e `pila_vuota()` per adeguarle al tipo `ptr_list`. Anch'esse sono contenute nel file `esempio_32.c`.

La funzione che realizza la stampa di tutti i nodi di un albero binario visitandoli in profondità per mezzo di una pila è riportata nel seguito. Si noti che, volendo visitare in profondità prima il sottoalbero sinistro e poi quello destro, i puntatori alle relative radici vanno inseriti nella pila nell'ordine opposto: prima il puntatore al sottoalbero destro, poi quello al sottoalbero sinistro.

```
void stampa_in_profondita(int_bin_tree *root) {
   ptr_list *s_element;
   ptr_list *s_top = NULL;

   if (root == NULL) return; //albero vuoto, non faccio nulla
   s_top = push(s_top, root); //inserisco nella pila la radice
   while(!pilavuota(s_top)) { //finche' ci sono elementi nella pila
      s_element = pop(&s_top); //estraggo il prossimo nodo dalla pila
      printf("%d ", (s_element->tree_node)->key);  //stampo il valore della chiave
      if ((s_element->tree_node)->right != NULL) {           //se ha figlio destro
         s_top = push(s_top, (s_element->tree_node)->right); //lo inserisco nella pila
      }
      if ((s_element->tree_node)->left != NULL) {           //se ha figlio sinistro
         s_top = push(s_top, (s_element->tree_node)->left); //lo inserisco nella pila
      }
      free(s_element);
   }
}
```

12.8.3 Esempio 32: stampa delle chiavi di un albero binario in ampiezza e profondità

Il programma seguente crea e riempie un vettore coi dati immessi da tastiera. Lo interpreta poi come un albero realizzato mediante la notazione posizionale e crea il corrispondente albero implementato mediante puntatori. Quindi stampa l'albero appena creato con tre diverse funzioni di stampa:

1. stampa mediante la visita in ampiezza (par. 12.8.1);
2. stampa mediante la visita in profondità (par. 12.8.2);
3. stampa mediante la visita in preordine (par. 12.6.3).

E' facile vedere che la seconda e la terza stampa coincidono.

Di seguito è riportato solo il codice della funzione `main()`. l'intero programma (comprensivo di tipi dati, prototipi delle funzioni e definizione delle funzioni stesse) è contenuto nel file `esempio_32.c`.

```
int main() {
   int i, n, valore;
   int *A;
   int_bin_tree *tree_root = NULL; //puntatore alla radice dell'albero

   printf("Inserisci la dimensione del vettore: ");
   scanf("%d", &n);

   A = (int*)malloc(sizeof(int)*n);
   for (i = 0; i < n; i++) {
      scanf("%d", &valore);
      A[i] = valore;
   }

   tree_root = from_array_to_tree(A, 0, n);
   stampa_in_ampiezza(tree_root);
   printf("\n");
   stampa_in_profondita(tree_root);
   printf("\n");
   stampa_preorder(tree_root);
   printf("\n");
   return 0;
}
```