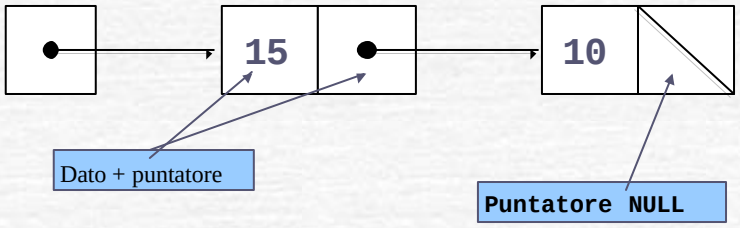
Gestione di strutture dati

- Le strutture autoreferenziate contengono uno o più campi puntatori a strutture dello stesso tipo.
- Possono essere collegate insieme per formare strutture dati come liste, code, pile, alberi.
- **Esempio**:



Gestione di strutture dati - liste

```
struct nodo {
      int dato;
      struct nodo *prossimo;
   }
```

```
typedef struct nodo Nodo;
typedef nodo *punt;
```

- Il puntatore **prossimo** punta ad un oggetto di tipo nodo, collegando un nodo al nodo successivo.
- L'accesso alla lista avviene tramite il puntatore al primo nodo della lista.
- Il nodo seguente viene acceduto tramite il campo puntatore del nodo corrente.
- Il puntatore dell'ultimo nodo è settato a NULL al fine di marcare la fine della lista.
- È' possibile usare una lista invece di un vettore se ho un numero imprecisato di elementi.

Gestione di strutture dati - liste

- Inserimento di un elemento in una lista ordinata:

```
void inserisci( punt *puntlista, int valore )
{  punt new_elem, precedente, corrente;

  new_elem = malloc( sizeof( Nodo ) );
  if ( new_elem != NULL ) {     /* nodo allocato */
    new_elem->dato = valore;
    new_elem->prossimo = NULL;
    precedente = NULL;
    corrente = *puntlista;
    while ( corrente != NULL && valore > corrente->dato ) {
      precedente = corrente;              /* si posiziona sul   */
      corrente = corrente->prossimo;  /* prossimo nodo  */
    }
```

Gestione di strutture dati - liste

```
if ( precedente == NULL ) {
      new_elem->prossimo = *puntlista;
      *puntlista = new_elem;
    }
    else {
      precedente->prossimo = new_elem;
      new_elem->prossimo = corrente;
    }
  }
  else
    printf( "%d non inserito. Memoria insufficiente.\n", valore );
}
```

Gestione di strutture dati - liste

- Cancellazione di un elemento da una lista ordinata:

```
int cancella( punt *puntlista, int valore )
{
  punt precedente, corrente, ausil;
  if ( valore == ( *puntlista )->dato ) {
    ausil = *puntlista;
    *puntlista = ( *puntlista )->prossimo;        /* elimina il nodo
   dalla lista */
    free( ausil );                                /* libera la memoria
   del nodo */
     return valore;
  }
  else {
    precedente = *puntlista;
    corrente = ( *puntlista )->prossimo;
```

Gestione di strutture dati - liste

```
while ( corrente != NULL && corrente->dato != valore ) {
       precedente = corrente;                     /* si posiziona sul */
       corrente = corrente->prossimo;   /* prossimo nodo        */
    }

    if ( corrente != NULL ) {
       ausil = corrente;
       precedente->prossimo = corrente->prossimo;
       free( ausil );
       return valore;
    }
  }

  return -1;
}
```

Gestione di strutture dati - pile

- Nelle pile (stack), i nodi vengono aggiunti ed eliminati solo dalla cima della pila.
- Si adotta quindi una tecnica last-in, first-out (LIFO)
- Il campo prossimo dell'ultimo nodo della pila è posto a NULL.
- Le operazioni di inserimento, rimozione di elementi dalla pila vengono indicati con **push** e **pop**
- **push** aggiunge un nuovo nodo sulla cima della pila
- **pop** rimuove un elemento dalla cima della pila, restituendo il valore contenuto nel nodo eliminato.

Gestione di strutture dati - pile

- Inserimento di un nodo sulla pila (push)

```
void push( punt *stack, int info )
{
  punt new_elem;

  new_elem = malloc( sizeof( Nodo ) );
  if ( new_elem != NULL ) {
    new_elem->dato = info;
    new_elem->prossimo = *stack;
    *stack = new_elem;
  }
  else
    printf("%d non inserito. Memoria insufficiente.\n ", info );
}
```

Gestione di strutture dati - pile

- Cancellazione di un nodo dalla pila (pop)

```
int pop( punt *stack )
{
  punt ausil;
  int valore;

  ausil = *stack;
  valore = ( *stack )->dato;
  *stack = ( *stack )->prossimo;
  free( ausil );
  return valore;
}
```

Gestione di strutture dati - code

- I nodi vengono eliminati solo in testa.
- I nodi vengono aggiunti solo in coda.
- Tecnica detta first-in, first-out (FIFO)
- Le operazioni ammissibili sono quindi **accoda** ed **elimina**.

Gestione di strutture dati - code

- Accodamento di un elemento in coda:

```
void accoda( punt *testa, punt *coda, int valore )
{
  punt new_elem;

  new_elem = malloc( sizeof( Nodo ) );

  if ( new_elem != NULL ) {
    new_elem->dato = valore;
    new_elem->prossimo = NULL;
```

Gestione di strutture dati - code

```
      if (  *testa ) == NULL)
       *testa = new_elem;
    else
      ( *coda )->prossimo = new_elem;

    *coda = new_elem;
  }
  else
    printf("%d non inserito. Memoria insufficiente.\n", valore );
}
```

Gestione di strutture dati - code

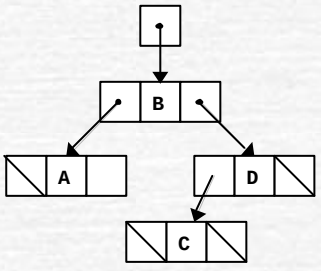
- Eliminazione di un elemento dalla coda:

```
int elimina( punt *testa, punt *coda )
{
  int valore;
  punt ausil;

  valore = ( *testa )->dato;
  ausil = *testa;
  *testa = ( *testa )->prossimo;
  if ( *testa == NULL )
        *coda = NULL;
  free( ausil );
  return valore;
}
```

Gestione di strutture dati - alberi

- Il nodo di un albero contiene due o più puntatori, a differenza dei nodi delle strutture dati viste sinora.
- Considereremo alberi binari, con due campi puntatori.
- **Esempio**:



Gestione di strutture dati - alberi

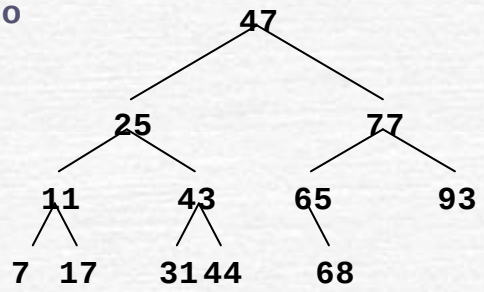
- La definizione del singolo nodo dell'albero potrà essere:

```
struct nodoalbero {
  struct nodoalbero *sinistro;
  int dato;
  struct nodoalbero *destro;
};

typedef struct nodoalbero Nodoalbero;
typedef Nodoalbero *puntalbero;
```

Gestione di strutture dati - alberi

- Un possibile esempio di applicazione è l'albero di ricerca binaria, in cui non esistono nodi duplicati e, per qualunque nodo dell'albero che non sia una foglia, il campo informazione di tutti i nodi del sottoalbero sinistro sono minori di quello del nodo considerato, mentre quelli del sottoalbero destro sono maggiori.
- **Esempio**



Gestione di strutture dati - alberi

- L'inserimento di un nuovo nodo nell'albero avverrà come nodo foglia, rispettando l'ordinamento dell'albero di ricerca. Verrà allocato il nuovo nodo e la funzione di inserimento richiamerà sé stessa (sul sottoalbero sinistro o destro a seconda del valore da inserire e del campo info del nodo dell'albero che sto considerando), fino a raggiungere un campo puntatore pari a NULL.
- In aggiunta è possibile definire delle funzioni di visita dell'albero in modalità **infissa**, **prefissa** o **postfissa**:
  - **infissa**: visita sottoalbero sx, elaboro il nodo corrente, visita sottoalbero dx.
  - **prefissa**: elaboro il nodo corrente, visita sottoalbero sx, visita sottoalbero dx.
  - **postfissa**: visita sottoalbero sx, visita sottoalbero dx, elaboro il nodo corrente,

Gestione di strutture dati - alberi

- Inserimento di un nodo nell'albero di ricerca binario:

```
void insertNode( puntalbero *albero, int valore )
{
  if ( *albero == NULL ) {    /* *albero vuoto  */
     *albero = malloc( sizeof( Nodoalbero ) );

     if ( *albero != NULL ) {
        ( *albero )->dato = valore;
        ( *albero )->sinistro = NULL;
        ( *albero )->destro = NULL;
     }
     else
        printf("%d non inserito. Memoria insufficiente.\n", valore);
  }
```

Gestione di strutture dati - alberi

```
else
    if ( valore < ( *albero )->dato )
      insertNode( &( ( *albero )->sinistro ), valore );
    else if ( valore > ( *albero )->dato )
      insertNode( &( ( *albero )->destro ), valore );
    else
      printf( ”informazione già presente nell’albero" );
}
```

Gestione di strutture dati - alberi

- Visita infissa dell’albero di ricerca binario:

```
void infisso( puntalbero albero )
{
  if ( albero != NULL ) {
    infisso( albero->sinistro );
    printf( "%3d", albero->dato );
    infisso( albero->destro );
  }
}
```

Gestione di strutture dati - alberi

- Visita prefissa dell'albero di ricerca binario:

```
void prefisso( puntalbero albero )
{
  if ( albero != NULL ) {
    printf( "%3d", albero->dato );
    prefisso( albero->sinistro );
    prefisso( albero->destro );
  }
}
```

Gestione di strutture dati - alberi

- Visita postfissa dell'albero di ricerca binario:

```
void postfisso( puntalbero albero )
{
  if ( albero != NULL ) {
    postfisso( albero->sinistro );
    postfisso( albero->destro );
    printf( "%3d", albero->dato );
  }
}
```

I/O su file ad accesso casuale

- Si differenziano dai file ad accesso sequenziale per la possibilità di poter accedere a punti precisi senza dover scandire l'intero file fino a quel punto.
- Abbiamo già creato ed usato file ad accesso sequenziale mediante le funzioni di libreria di I/O printf, scanf, getc, etc. Hanno lo svantaggio di non poter essere modificati senza il rischio di poter coprire/danneggiare altri dati. Questo perché, in generale, contengono dati memorizzati a lunghezza variabile. Infatti, ad esempio, 4.2873, 7.9, 0.9828188 sono tutti numeri a virgola mobile, ma vengono memorizzati sul file con dimensioni differenti.

I/O su file ad accesso casuale

- Nei file ad accesso casuale, l'accesso ad un singolo record avviene istantaneamente e senza scandire altri record.
- Possono essere inseriti, modificati, cancellati dati senza distruggere altri dati.
- Questo è possibile utilizzando record a lunghezza fissa.
- I dati in questo tipo di file non sono formattati (raw bytes).
- Tutti i dati dello stesso tipo occupano la stessa quantità di memoria.
- Di solito non sono direttamente leggibili dall'utente.

I/O su file ad accesso casuale

- Le più importanti funzioni di scrittura e lettura sono fwrite e fread.
- **fwrite** trasferisce un certo numero di dati da una locazione in memoria su un file.
  Esempio:
  fwrite(&valore, sizeof(int), 1, puntfile); con
  &numero – indirizzo da cui iniziare il trasferimento
  sizeof(int) – Numero di byte da trasferire (per il singolo record)
  1 – Numero di record da trasferire: ad esempio per i vettori si specifica la dimensione del vettore. In questo caso viene trasferito un solo elemento
  puntfile - file su cui trasferire i dati

I/O su file ad accesso casuale

- Da notare come sia possibile scrivere strutture o vettori di strutture.
  Esempio:

  fwrite(lista, sizeof (struct nodo),1,puntfile);

  scrive il singolo nodo della lista a partire dalla posizione corrente del puntatore di accesso del file.

  fwrite(vettore, sizeof (int),100, puntfile);

  scrive un vettore di 100 interi a partire dalla posizione corrente del puntatore di accesso al file.

I/O su file ad accesso casuale

- **fread** trasferisce un certo numero di dati da un file ad una locazione di memoria.
  Esempio:

  ```
  fread( albero->sinistro, sizeof(Nodoalbero), 1, puntfile);
  ```

  anche in questo caso quindi è possibile operare con strutture dati e con vettori, esattamente come per **fwrite**.

I/O su file ad accesso casuale

- **fseek** setta il puntatore di accesso al file in una data posizione.
  Utilizzo:
  fseek(puntatore, offset, costante_simbolica);
  - **puntatore**: puntatore a file
  - **offset**: posizione in cui scrivere rispetto alla posizione iniziale
  - **costante simbolica**: specifica la posizione iniziale:
    SEEK_SET - all'inizio del file
    SEEK_CUR - alla corrente posizione
    SEEK_END - alla fine del file

I/O su file ad accesso casuale

- **Esempio**: scrittura/ modifica di record a lunghezza fissa in una data posizione all'interno del file

```
#include <stdio.h>

struct scheda {
  int posizione;
  char cognome[ 15 ];
  char nome[ 10 ];
  char telefono[ 10 ];
};

int main()
{
  FILE *puntfile;
  struct scheda amico = { 0, "", "", “” };
```

I/O su file ad accesso casuale

```
  if ( ( puntfile = fopen( "rubrica.dat", "r+" ) ) == NULL )
    printf( "Il file non può essere aperto\n" );
  else {
    printf( "Inserisci la posizione della scheda"
          " ( 1 - 100, 0 per terminare )\n? " );
    scanf( "%d", &amico.posizione );

    while ( amico.posizione != 0 ) {
      printf( "Enter cognome, nome, telefono\n? " );
      fscanf( stdin, "%s %s %s", amico.cognome,
            amico.nome, amico.telefono );
```

I/O su file ad accesso casuale

```
        fseek( puntfile, ( amico.posizione - 1 ) *
              sizeof( struct scheda ), SEEK_SET );
        fwrite( &amico, sizeof( struct scheda ), 1,
               puntfile );
        printf( "Inserisci la posizione della scheda\n " );
        scanf( "%d", &amico.posizione );
      }

      fclose( puntfile );
    }

    return 0;
}
```

I/O di basso livello, descrittori di file

- Come esistono primitive tipo fopen, fread fwrite, fscanf, fgets, le cui operazioni sono però limitate ai file, il C mette a disposizione delle primitive che possono agire indifferentemente su file o su dispositivi di I/O di tipo diverso, quali interfacce di rete, interfacce seriali ecc.
- Si utilizzano quindi una serie di primitive per accedere a entità di tipo diverso. Queste fanno parte di librerie cosiddette per **l'I/O di basso livello**, che permettono di operare su diversi device. Vengono ad esempio utilizzate per scrivere applicazioni per comunicazioni via rete.

I/O di basso livello, descrittori di file

- Ogni dispositivo per I/O viene aperto con una funzione detta **open** che restituisce un intero detto "descrittore di file". Questo numero intero rappresenta un indice per una struttura dati che descrive le modalità con cui operare sul dispositivo. Tale descrittore verrà utilizzato in tutte le chiamate per l'effettuazione di I/O.
- Dal momento che si parla di operazioni a basso livello, non esiste più distinzione tra file di testo e file binari. Tutti i file sono visti come concatenazione di byte.

I/O di basso livello, descrittori di file

- Un esempio d'uso di queste primitive è il seguente:

```
/*leggo i byte di un file binario. Il nome del file e' su riga di comando */
#include <stdio.h>
#include <fcntl.h>
main(int argc, char **argv)
{ int fd; int bytes_read; char byte;
if((fd=open(argv[1],O_RDONLY))=-1) exit(1); /*errore,file non aperto*/
while ( (bytes_read=read(fd,&byte,1))>0 ) /* leggo un byte */
{
.... uso il byte letto .....
}
if (bytes_read==-1) exit(1); /* errore in lettura file */
close(fd);
}
```

I/O di basso livello, descrittori di file

- Per aprire un file si usa la funzione:

  int open(char *filename, int flag, int perms);

  che ritorna un file descriptor con valore >=0, oppure -1 se l'operazione fallisce.
- Il parametro **flag** controlla l'accesso al file ed è una combinazione (un OR bit a bit) ha i seguenti predefiniti valori definiti nel file fcntl.h: O_APPEND, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_WRONLY ecc., con i seguenti significati:
- O_APPEND apre per aggiungere dati in fondo al file, e posiziona il puntatore alla posizione corrente alla fine del file
- O_CREAT crea un file nuovo se non esiste
- O_EXCL viene usato con O_CREAT, se il file già esiste causa errore

I/O di basso livello, descrittori di file

- O_RDONLY apre e consente solo la lettura, si posiziona all'inizio del file
- O_RDWR apre e consente sia lettura che scrittura si posiziona all'inizio
- O_WRONLY apre e consente solo la scrittura
- O_TRUNC se il file esiste viene troncato, riparte da lunghezza zero
- Il parametro "**perms**" specifica i permessi da assegnare al file quando questo viene creato, in caso contrario viene ignorato.
- Per creare un file si puo' anche usare la funzione:

  int creat(char *filename, int perms);

I/O di basso livello, descrittori di file

- Per chiudere un file si usa:

```
int close(int fd);
```

- Per leggere/scrivere uno specificato numero di bytes da/su un file immagazzinati in una locazione di memoria specificata da "buffer" si utilizzano:

```
int read(int fd, char *buffer, unsigned length);
int write(int fd, char *buffer, unsigned length);
```

che ritornano il numero di byte letti/scritti o -1 se falliscono.

Parametri su riga di comando

- Si definisce il main nel deguente modo:
  int main(int argc, char * argv[])
  dove
  - **int argc** è il numero di argomenti passati
  - **char *argv[]** è un vettore di stringhe (vettore di puntatori a carattere) che conterrà la lista dei parametri inseriti sulla riga di comando.
    argv[0] è il primo parametro, argv[1] il secondo e così via.
- **Esempio:** **$ dividi 400 8**
  argc: 3
  argv[0]: ”dividi”
  argv[1]: ”400”
  argv[2]: ”8"

Parametri su riga di comando

```
#include <stdlib.h>
int main ( int argc, char *argv[] )
{   int a, b;
    float c;

    if ( argc != 3 )
        printf( “Uso: dividi num1 num2\n” );
    else
    {   a = atoi( argv[1] );
        b = atoi( argv[2] );
        c = (float) a / b;
        printf(“%d / %d = %f\n”, a, b, c);
    }
    return 0;
}
```

Operatori di ridirezione e pipe dell’I/O

- In C è possibile ridirigere lo standard input/output (normalmente tastiera e video)
- I simboli di **ridirezione** sono ‘<‘ e ‘>’
- E’ una caratteristica del sistema operativo, non del linguaggio C.
- **Esempio:**

  $ dividi < input.txt

  In questo modo il programma legge i valori di input da un file, piuttosto che da tastiera.

  $ dividi < input.txt > output.txt

  Legge i valori dal file input.txt e genera un file output.txt con i risultati

Operatori di ridirezione e pipe dell'I/O

- Output in append: si utilizza l'operatore '>>'.
  Esempio:
  $ dividi >> output.txt
  appende l'output di dividi alla fine del file output.txt. Preserva quindi il contenuto precedente. Invece
  $ dividi > output.txt
  avrebbe cancellato l'eventuale contenuto precedente di output.txt.
- **Operatore pipe:** è rappresentato dal simbolo '|'
- Permette di utilizzare l'output di un programma come input di un altro
- **Esempio**:
  $ codifica | decodifica
  In questo modo, ad esempio, l'output di codifica diventa l'input di decodifica.

Definizione di tipi enumerati

- Un tipico impiego della definizione di tipi è il caso dei tipi enumerati, ossia che ammettono un numero finito di valori ordinati linearmente.
- **Esempio**:
  typedef enum {lun,mar,mer,gio,ven,sab,dom} giorni;
- Il compilatore attribuisce ai valori di un tipo enumerato un numero intero progressivo a partire da 0. Sfruttando questa particolarità possiamo definire un tipo booleano:
  typedef enum {false, true} boolean;
- Questa dichiarazione attribuisce a false il valore 0 e a true il valore 1: quindi un test del tipo
  if (true) ...
  avrà il comportamento atteso.

Approfondimenti sull’allocazione dinamica

- Finora non si è mai approfondito il concetto di dimensione di rappresentazione in memoria dei tipi di dati utilizzati.
- Questo perché a seconda della versione di C o dell'architettura, lo spazio richiesto per memorizzare i dati può essere diverso.
- In C esiste, come già detto, l’operatore sizeof che serve per determinare qual’ è l'ingombro in memoria di un certo dato. Esso richiede come argomento uno specificatore di tipo o una espressione, ritornando un intero che corrisponde al numero di byte di memoria necessari per rappresentare gli oggetti di tale tipo o un oggetto del tipo corrispondente all'espressione.

Approfondimenti sull’allocazione dinamica

- Esempio: proviamo ad eseguire il programma

```
#include <stdio.h>

void main()
{
int a;
double b;
char c[30];
double d[30];

printf("Dimensioni dei tipi interi.\n");
printf("\tNumero byte per \"char\": %i\n", sizeof(char));
printf("\tNumero byte per \"short\": %i\n", sizeof(short));
```

Approfondimenti sull'allocazione dinamica

```
printf("\tNumero byte per \"int\": %i\n", sizeof(int));
printf("\tNumero byte per \"long\": %i\n", sizeof(long));
printf("Dimensioni dei tipi float.\n");
printf("\tNumero byte per \"float\": %i\n", sizeof(float));
printf("\tNumero byte per \"double\": %i\n", sizeof(double));
printf("\tNumero byte per \"long double\": %i\n", sizeof(long double));
printf("Dimensioni dei tipi puntatori.\n");
printf("\tNumero byte per \"char *\": %i\n", sizeof(char *));
printf("\tNumero byte per \"short *\": %i\n", sizeof(short *));
printf("\tNumero byte per \"int *\": %i\n", sizeof(int *));
printf("\tNumero byte per \"long *\": %i\n", sizeof(long *));
printf("\tNumero byte per \"float *\": %i\n", sizeof(float *));
printf("\tNumero byte per \"double *\": %i\n", sizeof(double *));
printf("\tNumero byte per \"long double *\": %i\n", sizeof(long double *));
```

Approfondimenti sull'allocazione dinamica

```
printf("Dimensioni di espressioni.\n");
printf("\tNumero byte per \"7 + 14\": %i\n", sizeof(7 + 14));
printf("\tNumero byte per \"7.0 / 14.0\": %i\n", sizeof(7.0 / 14.0));
printf("\tNumero byte per \"a\": %i\n", sizeof(a));
printf("\tNumero byte per \"b\": %i\n", sizeof(b));
printf("\tNumero byte per \"c\": %i\n", sizeof(c));
printf("\tNumero byte per \"d\": %i\n", sizeof(d));
}
```

Approfondimenti sull'allocazione dinamica

- Il risultato in output, sul compilatore Borland turbo C++ 3.0 utilizzato in laboratorio, sarà:

Dimensioni dei tipi interi.
Numero byte per "char": 1
Numero byte per "short": 2
Numero byte per "int": 2
Numero byte per "long": 4
Dimensioni dei tipi float.
Numero byte per "float": 4
Numero byte per "double": 8
Numero byte per "long double": 10
Dimensioni dei tipi puntatori.
Numero byte per "char *": 2
Numero byte per "short *": 2

Approfondimenti sull'allocazione dinamica

Numero byte per "int *": 2
Numero byte per "long *": 2
Numero byte per "float *": 2
Numero byte per "double *": 2
Numero byte per "long double *": 2
Dimensioni di espressioni.
Numero byte per "7 + 14": 2
Numero byte per "7.0 / 14.0": 8
Numero byte per "a": 2
Numero byte per "b": 8
Numero byte per "c": 30
Numero byte per "d": 240

Approfondimenti sull'allocazione dinamica

- **Osservazioni**:
  - Un puntatore, indipendentemente dal dato a cui effettivamente punta, ha sempre la stessa dimensione: infatti il puntatore rappresenta un indirizzo in memoria e la quantità di byte necessari per memorizzare questo indirizzo è fissa dipende dall'architettura del nostro sistema.
  - Nelle stampe relative alle espressioni, nelle prime due è come se chiedessimo la dimensione di un oggetto di tipo int nel primo caso e di tipo double nel secondo. Le ultime righe ci dicono qual'è l'occupazione in memoria di ciascuna delle variabili argomento dell'operatore sizeof. Quelle relative ai vettori danno come risultato il numero di byte richiesti per memorizzare i vettori nel loro complesso e quindi il risultato sarà 30*sizeof(char) e 30*sizeof(double).

Approfondimenti sull'allocazione dinamica

- funzione **calloc()**: simile a **malloc()**, viene utilizzata per lo stesso scopo, ossia allocare memoria dinamicamente. La differenza sostanziale è che invece di ricevere un solo parametro indicante il numero di byte richiesti, questa funzione riceve due parametri, ossia il numero di oggetti che vogliamo siano allocati e quanti byte sono richiesti per memorizzare ciascuno di questi oggetti. Inoltre, questa funzione garantisce l'inizializzazione a zero dell'area di memoria restituita.
- **Esempio**:

```
vettore = (double *)calloc(n, sizeof(double));
```

- visto che anche calloc restituisce un puntatore generico a void, è richiesto il cast al tipo corretto (puntatore a double, trattandosi di vettore di double).

Approfondimenti sull'allocazione dinamica

- la funzione **realloc()** richiede come parametri il puntatore alla prima locazione di memoria dell'oggetto precedentemente allocato e la nuova dimensione dell'oggetto. Restituisce il nuovo puntatore alla memoria reallocata o NULL in caso di insuccesso. Se si specifica una nuova dimensione pari a zero, allora la memoria viene liberata.

Funzioni a numero di argomenti variabile

- Si utilizzano le definizioni contenute nel file **stdarg.h,** in particolare:
- **va_list:** definizione di struttura.
- **va_start()**: funzione di inizializzazione, riceve in input una struttura di tipo **va_list** e il numero di parametri variabile.
- **va_arg()**: funzione che ritorna un parametro ogni volta che viene richiamata. Automaticamente punterà al prossimo parametro. Riceve in input una struttura di tipo **va_list** e il tipo che deve essere ritornato.
- **va_end()**: funzione di chiusura.

Funzioni a numero di parametri variabile

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

double somma( int, ... );

int main()
{  double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

   printf( "%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
         "w = ", w, "x = ", x, "y = ", y, "z = ", z );
   printf( "%s%.3f\n%s%.3f\n%s%.3f\n",
         "La somma di w e x è ",  somma( 2, w, x ),
         "La somma di w, x, e y è ", somma( 3, w, x, y ),
         "La somma di w, x, y, e z è ", somma( 4, w, x, y, z ) );
   return 0;
 }
```

Funzioni a numero di parametri variabile

```
double somma( int i, ... )
{
   double totale = 0;
   int j;
   va_list ap;

   va_start( ap, i );

   for ( j = 1; j <= i; j++ )
     totale += va_arg( ap, double );

   va_end( ap );
   return totale;
}
```

Funzioni matematiche - math.h

- int abs (int n) - valore assoluto di un intero.
- double acos(double x) - arcocoseno di x.
- double asin(double x) - arcoseno di x.
- double atan(double x) - arcotangente di x.
- double ceil(double x) - il piu piccolo intero maggiore o uguale a x.
- double cos(double x) - coseno di angolo x in radianti
- double sin(double x) - seno di angolo x in radianti
- double exp(double x - esponenziale
- double fabs (double x) - valore assoluto di un double
- double floor(double x) - il piu grande intero minore o uguale a x.
- labs(long n) - valore assoluto di un long

Funzioni matematiche - math.h

- double log(double x) - logaritmo naturale
- double log10 (double x) - logaritmo in base 10
- double pow (double x, double y) - potenza, x elevato alla y.
- void srand(unsigned seed) - inizializza il generatore di numeri casuali con un certo valore.
- int rand (void) - genera un numero casuale tra 0 e 32.
- int random(int max_num) - genera un numero casuale tra 0 e max_num.
- void randomize(void) - inizializza il generatore di numeri casuali.
- double sqrt(double x) - radice quadrata
- double tan(double x) - tangente di un angolo in radianti

Funzioni per la gestione del tempo - time.h

- **clock_t clock(void);**
  restituisce il tempo di CPU utilizzato dal processo chiamante. Il valore restituito è convenzionalmente il numero di clock eseguiti. Per ottenere il tempo in secondi deve essere diviso per la costante **CLOCKS_PER_SEC.**
- **Esempio**:

```
clock_t start, end;
start = clock();

istruzioni del programma ...

end = clock();
printf("tempo impiegato in secondi: %lf\n",
(double)(end-start) / (double)CLOCKS_PER_SEC );
```