Array e arimetica dei puntatori

Gli array la cui dimensione viene definita all'interno del programma come una costante vengono detti array statici. È possibile dichiarare degli array la cui dimensione viene determinata solo all'atto della esecuzione del programma. Questo genere di array vengono detti array dinamici. Una prerogativa degli array dinamici è la possibilitè di modificare la loro dimensione durante la escuzione del programma.

Array statici

Array statici

Gli array statici sono quelli la cui dimensione (numero di elementi) viene specificata nel codice. Si chiamano statici perchè, una volta compilati, la loro dimensione non cambia. I due modi per dichiarare un array statico sono:

1. specificando la dimensione
2. dando gli elementi iniziali

Il seguente programma statici.c crea due vettori `a` e `b`, il primo di interi e il secondo di numeri reali. Si usando i due meccanismi di dichiarazione: il primo vettore viene dichiarato dando la sua dimensione, per il secondo diamo gli elementi che deve inizialmente contenere.

```
/*
  I due modi di dichiarare un array statico.
*/

int main() {
  int a[10];
  float b[]={0.12, 3.23, 1.23, 1e2, -12.43};

  return 0;
}
```

Il punto fondamentale è che in entrambi i casi, il numero di elementi del vettore è specificato nel codice del programma. Nel primo caso, il numero fra parentesi quadre è 10, quindi il vettore contiene 10 elementi. Nel secondo caso, non ci sono numeri fra parentesi, ma la lista dei valori iniziali contiene 5 elementi. Il numero di elementi di questi due vettori rimane sempre lo stesso durante tutta l'esecuzione del programma. Questo è il motivo per cui questi array vengono detti *statici*.

Un vettore si può pensare come un insieme di variabili dello stesso tipo. Ogni variabile di un vettore viene identificata dal nome del vettore più un numero intero positivo. Nell'esempio di sopra, la dichiarazione dell'array `int a[10];` crea un insieme di variabili `a[0]`, `a[1]`, ecc fino ad `a[9]`. Ognuna di queste variabili è caratterizzata da un indirizzo, un numero di byte occupati e un tipo. In particolare, il tipo è lo stesso per tutte le variabili di uno stesso vettore, e quindi anche lo spazio occupato da ognuna di esse. L'indirizzo associato è invece differente. Il programma indstatico.c, riportato qui sotto, stampa indirizzo e byte occupati da ognuna delle variabili che compongono il vettore `a`.

```
/*
  Indirizzi e byte occupati dagli elementi di un array statico.
*/

int main() {
```

```
  int a[10];
  float b[]={0.12, 3.23, 1.23, 1e2, -12.43};

  int i;

  for(i=0; i<=9; i++)
    printf("a[%d] ha indirizzo=%x e occupa %d bytes\n",
           i, &(a[i]), sizeof(a[i]) );

  return 0;
}
```

Se si prova a compilare ed eseguire questo programma, si ottiene una stampa di questo genere:

```
a[0] ha indirizzo=bffff428 e occupa 4 bytes
a[1] ha indirizzo=bffff42c e occupa 4 bytes
a[2] ha indirizzo=bffff430 e occupa 4 bytes
a[3] ha indirizzo=bffff434 e occupa 4 bytes
a[4] ha indirizzo=bffff438 e occupa 4 bytes
a[5] ha indirizzo=bffff43c e occupa 4 bytes
a[6] ha indirizzo=bffff440 e occupa 4 bytes
a[7] ha indirizzo=bffff444 e occupa 4 bytes
a[8] ha indirizzo=bffff448 e occupa 4 bytes
a[9] ha indirizzo=bffff44c e occupa 4 bytes
```

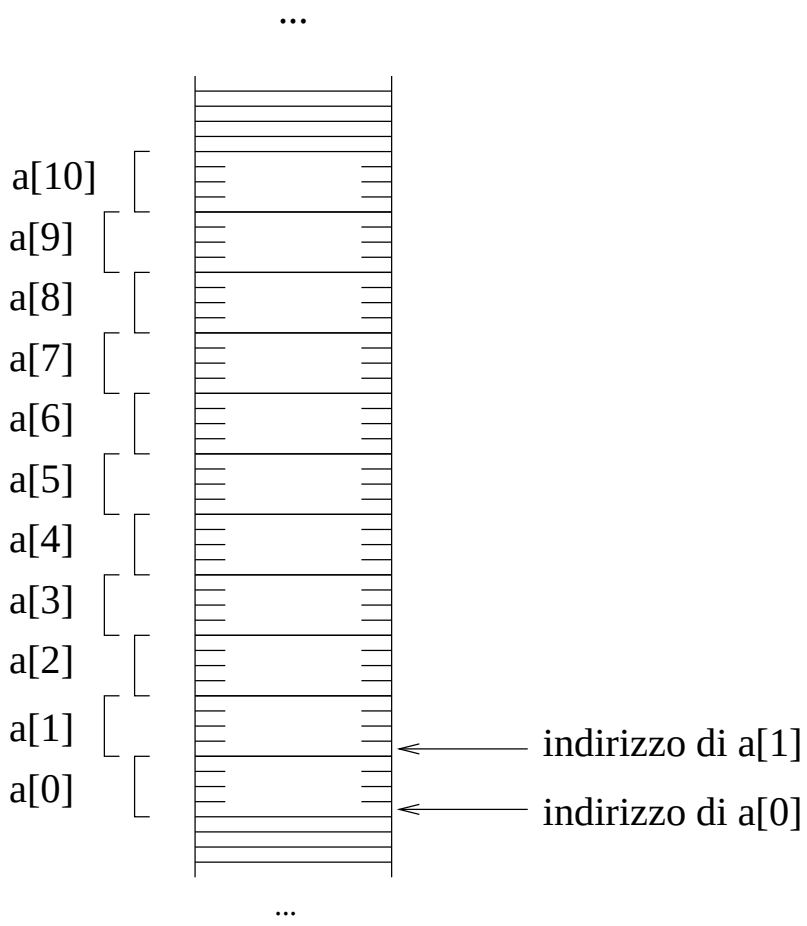
È chiaro che tutte le variabili hanno la stessa dimensione, il che è assolutamente ragionevole considerando che sono tutte variabili dello stesso tipo (gli array sono per definizione composti da variabili di un tipo). Essendo inoltre variabili diverse, i loro indirizzi sono anche essi tutti diversi. Se al contrario due elementi fossero associati allo stesso indirizzo, modificare l’uno sarebbe equivalente a modificare l’altro, cosa che non ci si aspetta che avvenga. Per lo stesso motivo, è chiaro che le zone di memoria che corrispondono a due elementi diversi non devono sovrapporsi neanche parzialmente.

Un altro fatto che risulta evidente dalla stampa di sopra è che le aree di memoria delle variabili sono consecutive: prima ci sono i quattro byte di `a[0]`, i successivi quattro sono quelli di `a[1]`, ecc. Questo fatto non è incidentale, ma avviene sempre per ogni possibile tipo di elemento del vettore.

La figura qui accanto dà una rappresentazione grafica della memoria occupata da questo vettore (rispetto alle figure precedenti, usiamo rettangoli più schiacciati per rappresentare i byte).

Come si può vedere, questo vettore occupa una zona di memoria che comincia con l'indirizzo di `a[0]`, ed è larga quanto il numero di elementi del vettore moltiplicato per `sizeof(int)`, ossia occupa `sizeof(int)` byte per ogni elemento del vettore.

In effetti, indirizzo e numero di byte occupati da un vettore statico si possono trovare anche usando le solite primitive del C, ossia `&` e `sizeof`.



In particolare, `&` fornisce l'indirizzo in cui inizia la zona di memoria occupata dal vettore (e quindi coincide con l'inizio della zona di memoria occupata dal suo primo elemento), mentre `sizeof` dice il numero di byte occupati dal vettore (questo è uguale al numero di elementi del vettore moltiplicato per la dimensione di ognuno di essi). Il seguente programma indvett.c stampa indirizzo e numero di byte occupati da un vettore statico.

```
/*
  Indirizzo e numero di byte di un vettore statico.
*/

int main() {
  int a[10];
  float b[]={0.12, 3.23, 1.23, 1e2, -12.43};

  printf("a ha indirizzo=%x e occupa %d bytes\n", &a, sizeof(a));

  printf("indirizzo del primo elemento=%x\n", &(a[0]));
  printf("byte occupati da ogni elemento=%d\n", sizeof(a[1]));

  return 0;
}
```

L’esecuzione di questo programma genera un risultato di questo tipo (i valori numerici dell’indirizzo possono ovviamente cambiare):

```
a ha indirizzo=bffff428 e occupa 40 bytes
indirizzo del primo elemento=bffff428
byte occupati da ogni elemento=4
```

È chiaro che `&a` e `&(a[0])` coincidono. Si vede anche chiaramente che il numero di byte occupati dal vettore è uguale al numero di byte occupati da ogni elemento moltiplicato per il numero di elementi.

Nota importante.

In questa e in alcune pagine successive, si parla di dimensione di un vettore, intendendo il numero di elementi che lo compongono. Per esempio, il vettore dichiarato con `int a[10];` è composto da 10 elementi, quindi ha dimensione 10. Per indicare il risultato della `sizeof`, usiamo invece il termine ‘‘numero di byte occupati’’. Il alcuni testi si può trovare ‘‘dimensione’’ come sinonimo dello spazio occupato da una variabile. Il contesto permette di solito di capire di cosa si sta parlando.

Allocazione di array

Allocazione di array

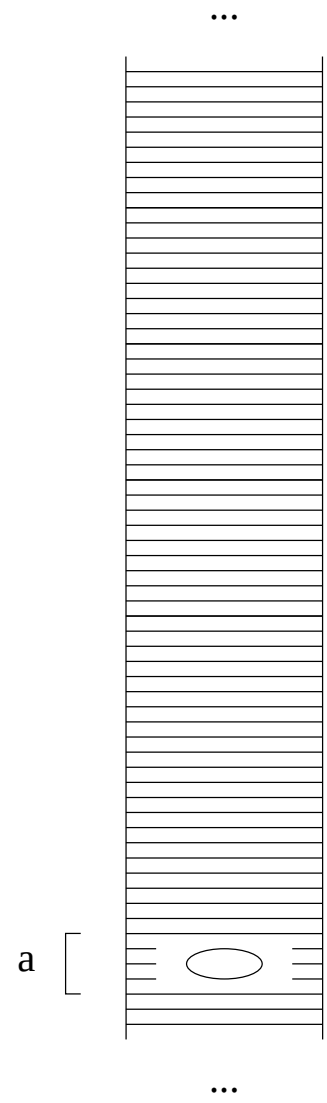
Nella pagina sulla allocazione di memoria si è visto come sia possibile creare una zona di memoria durante la esecuzione di un programma. Nel caso della allocazione di una singola variabile di tipo intero, la cosa non risultava molto utile. Ci sono invece dei vantaggi nel caso in cui si voglia allocare un intero vettore.

Supponiamo infatti di voler ricevere in input un vettore. Il primo numero che viene dato è la dimensione di questo vettore, mentre i numeri successivi sono gli elementi. Se volessimo realizzare un programma del genere usando i vettori statici, avremmo bisogno di decidere a priori un numero massimo di elementi.

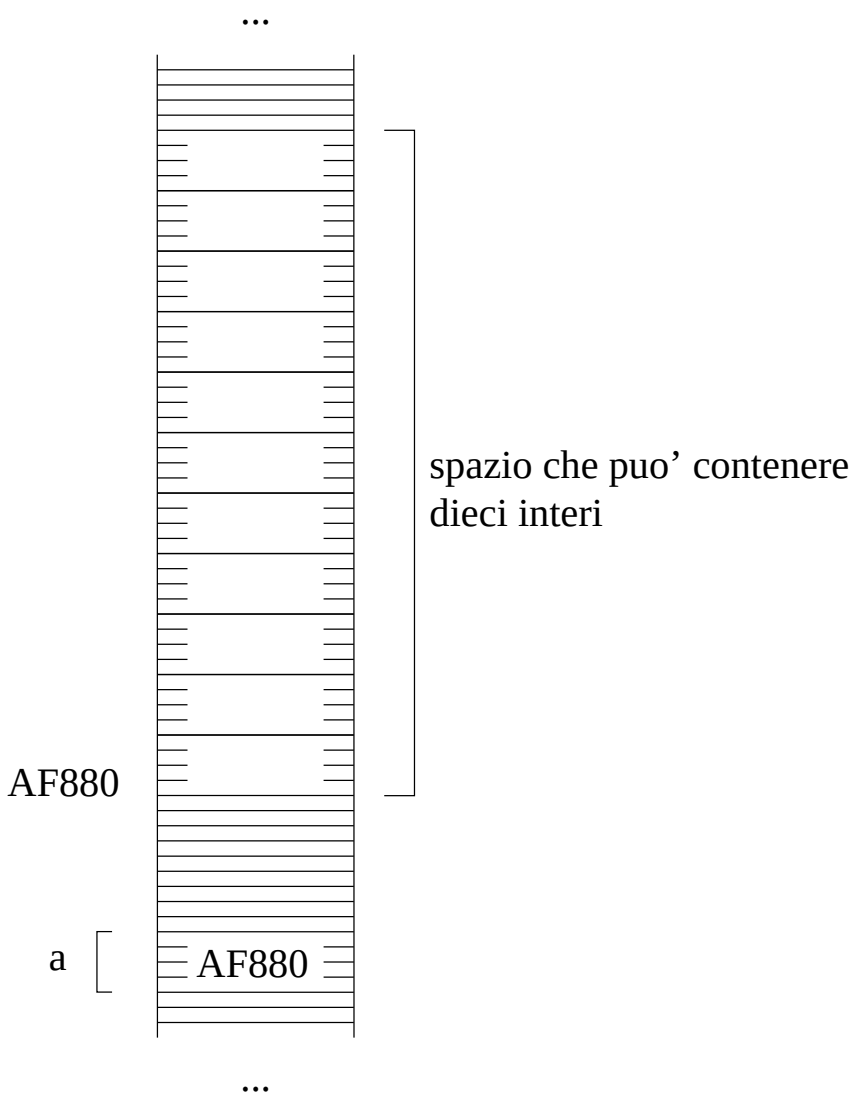
Allocando invece un vettore con la `malloc`, è possibile decidere in esecuzione quanti elementi questo vettore deve contenere.

I passi da eseguire sono i seguenti.

1. se il vettore è di tipo `int`, si definisce una variabile puntatore a `int`, per esempio `int *a`;
2. si determina la dimensione del vettore (nel nostro caso, si prende la dimensione come input);
3. si allocano un numero appropriato di byte: se il vettore deve contenere `x` elementi di tipo `t`, allora il numero di byte è dato da `x * sizeof(t)`;
4. si passa il numero di byte da allocare alla funzione `malloc`, e si mette il risultato nel puntatore;
5. da questo momento in poi, si può trattare la variabile puntatore come se fosse una variabile di tipo vettore.

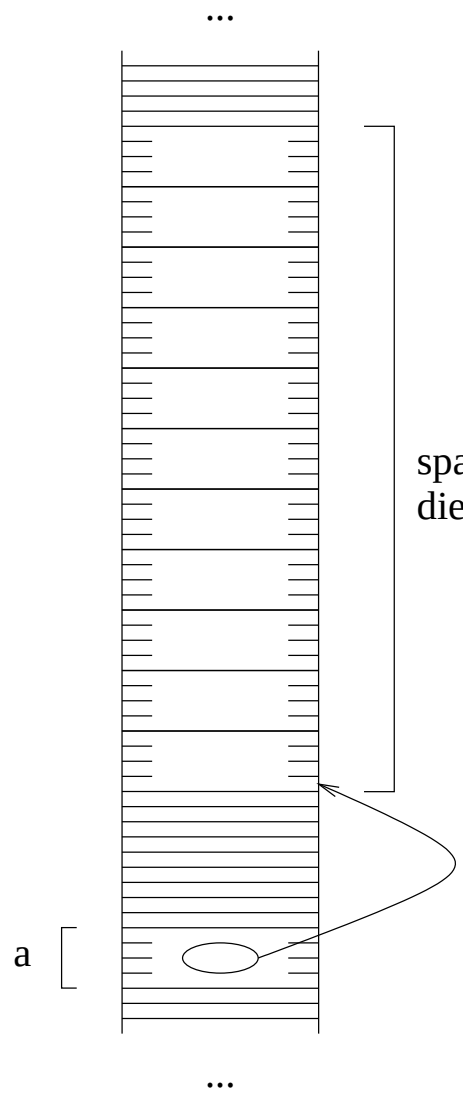


La figura qui accanto mostra lo stato iniziale della memoria. Si noti che esiste una sola variabile puntatore `x`, e che quindi il programma ha a disposizione solo 4 byte di memoria.



La funzione `malloc` mette a disposizione del programma una zona di memoria larga quanto il numero che riceve come argomento. In questo caso, la zona di memoria è larga `10*sizeof(int)`, ossia quanto basta per contenere dieci interi.

Il valore di ritorno della funzione `malloc` è l'indirizzo iniziale della zona di memoria che è stata riservata per il programma. Nell'esempio, questa zona di memoria inizia dalla locazione `AF880` (questo è un numero scritto in esadecimale). Questo numero viene messo nella variabile `a`.



La freccia è semplicemente una notazione comoda per far vedere quale è l'indirizzo che sta scritto in una variabile puntatore. Occorre però sempre ricordare che si tratta di una semplice notazione, e che quello che effettivamente sta scritto nella variabile puntatore è un numero.

Il seguente programma dinamico.c legge un numero in input, e alloca un vettore che contiene un numero di elementi pari al numero letto da input. Il vantaggio di questo modo di creare i vettori è che si può decidere la dimensione del vettore durante l'esecuzione del programma. In questo caso, per esempio, è l'utente a scegliere il numero di elementi del vettore, cosa che è impossibile da stabilire prima che il programma venga eseguito.

```
/*
  Allocazione di un array dinamico.
*/

#include<stdlib.h>

int main() {
  int *a;
  int d;

  printf("Quanti elementi? ");
  scanf("%d", &d);

  a=malloc(d*sizeof(int));

  a[0]=12;
```

```
  a[1]=a[0]+40;

  return 0;
}
```

Si noti che, una volta creata una zona di memoria di dimensione opportuna con la funzione `malloc`, si può usare il puntatore `a` come se fosse un vettore di interi. Per esempio, si può memorizzare un valore in `a[0]`, oppure usare il valore di `a[4]` in una espressione, ecc.

Il programma leggivettore.c è un esempio più completo, in cui si legge da input un numero, si alloca un vettore grande come questo numero, e poi si leggono tutti i suoi elementi.

```
/*
  Legge un vettore il cui numero di elementi
viene dato da input.
*/

#include<stdlib.h>

int main() {
  int *a;
  int d;
  int i;

                        /* chiede il numero di elementi */
  printf("Quanti elementi? ");
  scanf("%d", &d);

                        /* alloca il vettore */
  a=malloc(d*sizeof(int));

                        /* legge gli elementi */
  for(i=0; i<=d-1; i++) {
    printf("Dammi l'elemento a[%d]: ", i);
    scanf("%d", &(a[i]));
  }

                        /* stampa gli elementi del vettore */
  for(i=0; i<=d-1; i++)
    printf("a[%d] vale %d\n", i, a[i]);

  return 0;
}
```

La memoria per array statici e dinamici

La memoria per array statici e dinamici

Se si fa `int a[10]` oppure `int *b` e poi `b=malloc(10*sizeof(int))`, si ottiene in entrambi i casi un vettore di 10 elementi. Da questo punto in poi, sono definite le variabili `a[0]`, `a[1]` ecc, cosí come le variabili `b[0]`, `b[1]`, ecc.

Esistono però delle differenze nel modo in cui questi due vettori sono memorizzati. Questo risulta evidente se si prova a stampare indirizzo e numero di byte occupati da essi, come viene fatto nel programma statdin.c riportato qui sotto.

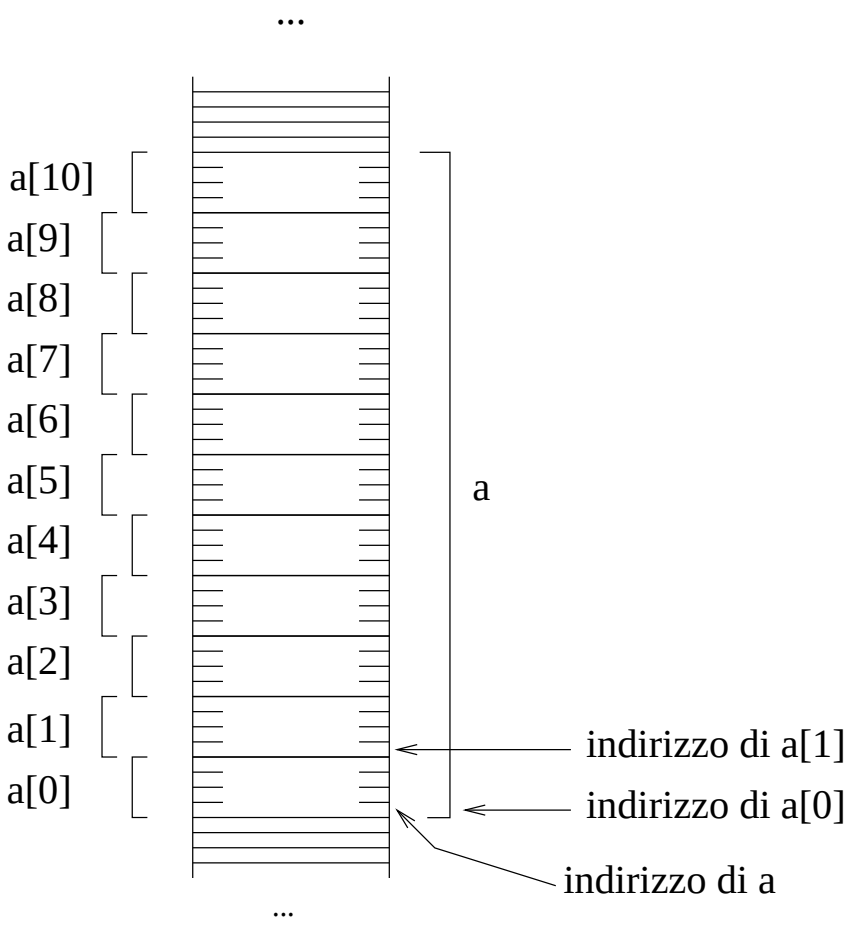
```
/*
  Differenza di allocazione di memoria fra
array statici e dinamici.
*/

#include<stdlib.h>

int main() {
  int a[10];
  int *b;

  b=malloc(10*sizeof(int));

                /* da questo punto in poi, a e b sembrano uguali, ma... */

  printf("a: indirizzo=%x indirizzo primo elemento=%x byte occupati=%d\n",
         &a, &(a[0]), sizeof(a));

  printf("b: indirizzo=%x indirizzo primo elemento=%x byte occupati=%d\n",
         &b, &(b[0]), sizeof(b));

  return 0;
}
```

Quello che si ottiene è un output di questo genere:

```
a: indirizzo=bffff428 indirizzo primo elemento=bffff428 byte occupati=40
b: indirizzo=bffff424 indirizzo primo elemento=8049688 byte occupati=4
```

Risultano quindi evidenti le differenze. Per prima cosa, l'indirizzo del vettore `b` non coincide con l'indirizzo del suo primo elemento `b[0]`. Secondo, il numero di byte occupati non è più pari al numero di byte occupati da ogni elemento moltiplicato per il numero di elementi. Si può anzi facilmente verificare che il numero di byte occupati da `b` non cambia se si allocano più o meno elementi.

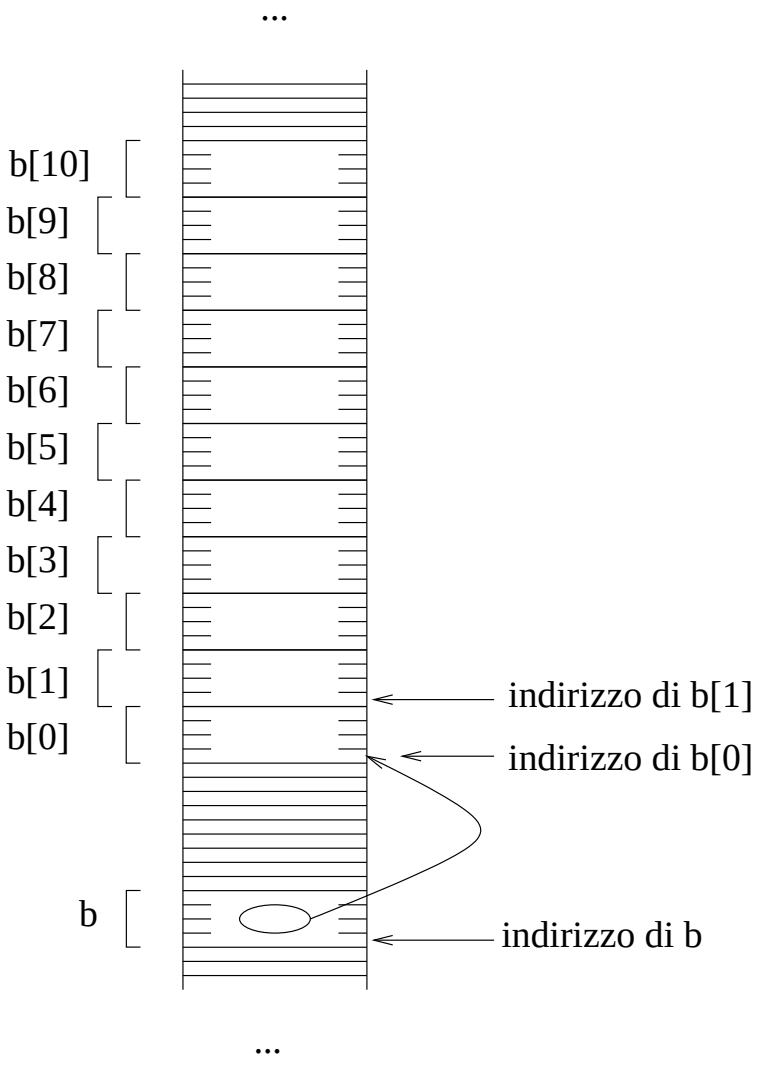
La prima figura mostra il modo in cui è disposto in memoria un array statico. Sono evidenziati gli indirizzi del vettore stesso, e dei suoi primi elementi.



Il caso del vettore dinamico è differente. Un vettore dinamico è in effetti un puntatore a intero, dal momento che è stato definito come `int *b`. Questo significa che occupa lo spazio che occupano tutti gli altri puntatori (per esempio, quattro elementi). Nel nostro caso, `b` occupa quattro elementi, come nella figura.

Quando si esegue la istruzione `malloc(10*sizeof(int))`, viene riservata una zona di memoria grande abbastanza da contenere 10 interi, se messi l'uno accanto all'altro. Si noti che la variabile puntatore `b` non è un argomento di `malloc`. Questo significa che la funzione `malloc` non influenza la variabile `b`, tranne che per il fatto che in essa viene messo l'indirizzo di inizio della zona di memoria creata da `malloc`

Questo spiega perfettamente la figura a fianco: la variabile `b` è un puntatore, e quindi ha la dimensione che hanno tutti i puntatori. Quando si crea la zona di memoria, l'indirizzo iniziale viene messo in questa variabile puntatore.



Quello che non viene spiegato, a questo punto, è il fatto che la zona di memoria creata da `malloc` si possa accedere usando `b[0]`, `b[1]`, ecc. In effetti, quello che si è detto nella parte sui puntatori è che, se `b` è un puntatore a intero, allora `*b` si comporta come una variabile di tipo intero.

Aggiungiamo ora una nuova regola sui puntatori: se `b` è un puntatore a intero, allora `b[0]` è equivalente a `*b`. Questo significa `b[0]` è come se fosse una variabile la cui zona di memoria associata è costituita dai byte a partire dall'indirizzo contenuto in `b`.

Per `b[1]` vale una regola analoga: dal punto di vista della programmazione ad alto livello, è equivalente a una variabile; la sua zona di memoria associata è quella subito sopra a `b[0]`. Lo stesso vale per `b[2]` ecc. Volendo riassumere, se `b` è una variabile di tipo puntatore a intero, allora si comporta come un vettore di interi, la cui dimensione dipende dal numero di byte creati con `malloc`. In questo gli array statici e dinamici si comportano nello stesso modo. La differenza si vede quando si trovano l'indirizzo del primo elemento del vettore, e la dimensione totale del vettore.

Per concudere, si fa osservare come la zona di memoria associata a un vettore statico rimane sempre la stessa durante la esecuzione del programma. Al contrario, per i vettori dinamici può cambiare. Per esempio, è possibile fare:

```
int *b;

b=malloc(10*sizeof(int));

/* alcune operazioni */

b=malloc(20*sizeof(int));
```

Alla fine della seconda `malloc` la dimensione del vettore è cambiata, e anche l'inizio della zona di memoria occupata può non essere lo stesso.

Arimetica dei puntatori

Arimetica dei puntatori

Finora abbiamo visto due modi di interpretare un puntatore: una variabile `int *p` è l'indirizzo iniziale della zona di memoria in cui si trova un intero, oppure è un vettore dinamico. Abbiamo poi visto come i vettori dinamici sono semplicemente dei puntatori a un indirizzo iniziale da cui inizia una sequenza di dati (nel nostro esempio, di interi).

Possiamo quindi dire che, in entrambi i casi, un puntatore a intero è l'indirizzo iniziale di una zona di memoria in cui sono memorizzati uno o più interi consecutivamente. La notazione `p[i]`, quando `p` è un puntatore, serve a indicare l'i-esimo intero di questa sequenza.

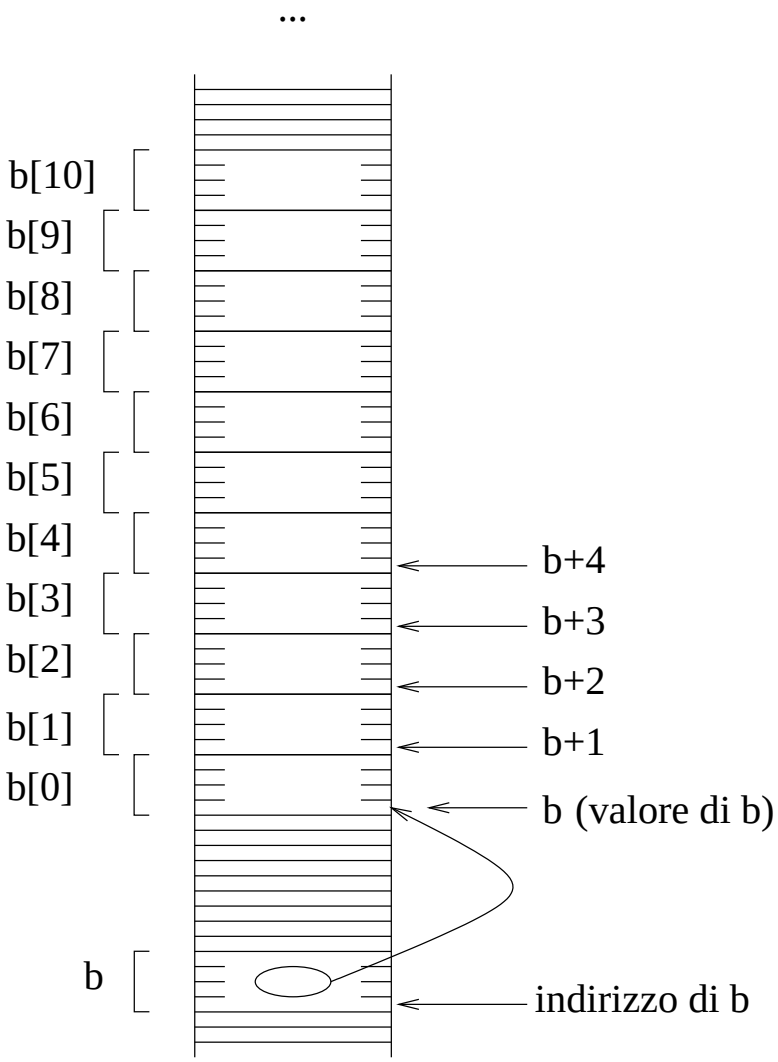
In C, è possibile effettuare delle semplici operazioni arimetiche sui puntatori, e fra puntatori e interi. Il caso più semplice è quello in cui si somma un intero a un puntatore. Il risultato non è però quello che ci si potrebbe aspettare a prima vista. Infatti, `p+3` *non* coincide con il valore di `p` a cui si somma tre. Il motivo è che l'indirizzo che si ottiene in questo modo sarebbe poco significativo. Infatti, sarebbe l'indirizzo del terzo byte del primo intero della zona puntata da `p`. È chiaro che spostarsi in mezzo a un intero in questo modo non è molto utile. Per questa ragione, il risultato della somma `p+3` viene definito come l'indirizzo del terzo intero della sequenza puntata da `p`. Il suo valore numerico è quindi ottenuto sommando a `p` il numero di byte occupati dall'intero, moltiplicato per tre.

Tutto questo discorso vale per gli interi. Per puntatori a un tipo di dato generico la regola è:

> *un puntatore a un tipo è l'indirizzo iniziale di una zona di memoria in cui sono memorizzati uno o più valori di quel tipo.*
>
> *`p+i` è l'indirizzo iniziale dell'i-esimo valore della sequenza puntata da `p`*

Se per esempio il tipo `double` si rappresenta con otto byte, e `p` è definito come `double *p`, allora `p` è un puntatore a `double`, ossia è l'indirizzo iniziale di una zona di memoria in cui si trova una sequenza di uno o più reali. Per la somma, `p+3` è l'indirizzo del terzo elemento della sequenza: dato che ogni reale occupa otto byte, il valore numerico di `p+3` è 3*8 volte maggiore del valore numerico di `p`.

La figura qui sotto indica gli indirizzi i cui valori sono rappresentati da somme fra puntatori e interi, nel caso di un puntatore a intero.



Dal momento che `b` è l'indirizzo iniziale di un intero, possiamo accedere a questo intero usando la notazione `*b`. Ma anche `b+1` è l'indirizzo iniziale di un intero, a cui possiamo quindi accedere come `*(b+1)`. Lo stesso vale per `*(b+2)`, `*(b+3)`, ecc. Facendo un confronto con la figura qui sopra, si vede chiaramente come `*b` e `b[0]` sono la stessa cosa. Inoltre, `*(b+1)` e `b[1]` sono ancora la stessa identica zona di memoria, ecc. Questo avviene per tipi qualsiasi, non solo interi, ed è dovuta al fatto che sia la notazione `b[i]` che la espressione `*(b+i)` indicano l'i-esimo elemento della zona di memoria puntata da `b`. In generale, vale la regola che `b[i]` e `*(b+i)` sono equivalenti.

È anche possibile sottrare un intero a una variabile di tipo puntatore, ma per il momento questo non ci interessa (in altre parole, si può fare `p-3`, che ha un significato abbastanza chiaro). È anche possibile fare la sottrazione fra due puntatori.

Il concetto generale che raccoglie tutte queste diverse espressioni è quello di *lvalue*, che è una qualsiasi espressione che ritorna un indirizzo.

Liberare la memoria

Liberare la memoria

La funzione `malloc` serve a dire che ci occorre una zona di memoria da usare. Quello che succede quando viene chiamata la funzione è che il calcolatore individua una zona di memoria libera (cioè una zona che non è già occupata da variabili o usata in altro modo), e ne restitusce l'indirizzo iniziale. Oltre a questo, viene memorizzato in qualche modo che questa zona di memoria è in uso, e che quindi non deve venire più usata da altre variabili, e non deve venire restituita da successive chiamata della funzione `malloc`. Se cosí non fosse, si potrebbero verificare situazioni in cui due puntatori, che ci aspettiamo non essere in relazione, puntano a una stessa locazione, con conseguenti effetti inattesi.

Quando una zona di memoria non serve più, occorre rilasciarla chiamando la funzione `free`. Questa funzione dice che la zona di memoria non ci serve più, e che può quindi venire usata in altro modo.

Il vantaggio di chiamare la funzione `free` è che, in questo modo, la memoria non più utilizzata dal programma può venire ‘‘riciclata’’, ossia riutilizzata. Si considerino i due programmi seguenti nolibera.c e libera.c. Il primo continua ad allocare sempre nuovi blocchi di memoria. Alla fine, tutta la memoria sarà stata assegnata, per cui la successive chiamate alla funzione `malloc` falliscono.

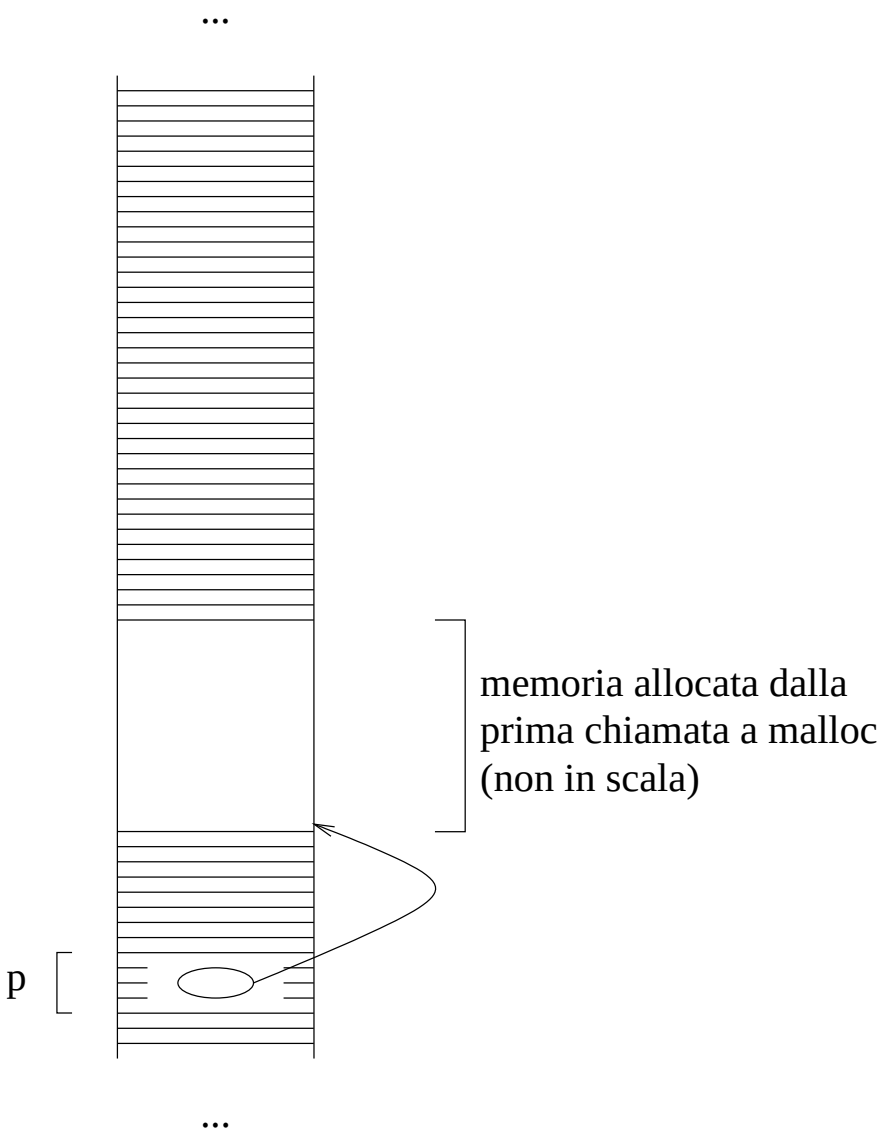
```
/*
  Alloca memoria senza mai liberarla.
*/

#include<stdlib.h>

int main() {
  int i;
  int *p;


  for(i=0; i<=100000; i++) {

                /* alloca cinquantamila interi */
     p=malloc(50000*sizeof(int));
     if(p==NULL) {
       printf("Impossibile allocare altra memoria\n");
       exit(1);
     }
  }

  return 0;
}
```

Al contrario, il programma libera.c libera la memoria. In questo modo, la memoria che non serve più viene rilasciata, e può quindi venire usata di nuovo. Per esempio, la funzione `malloc` può in seguito riservare nuovamente questa zona.

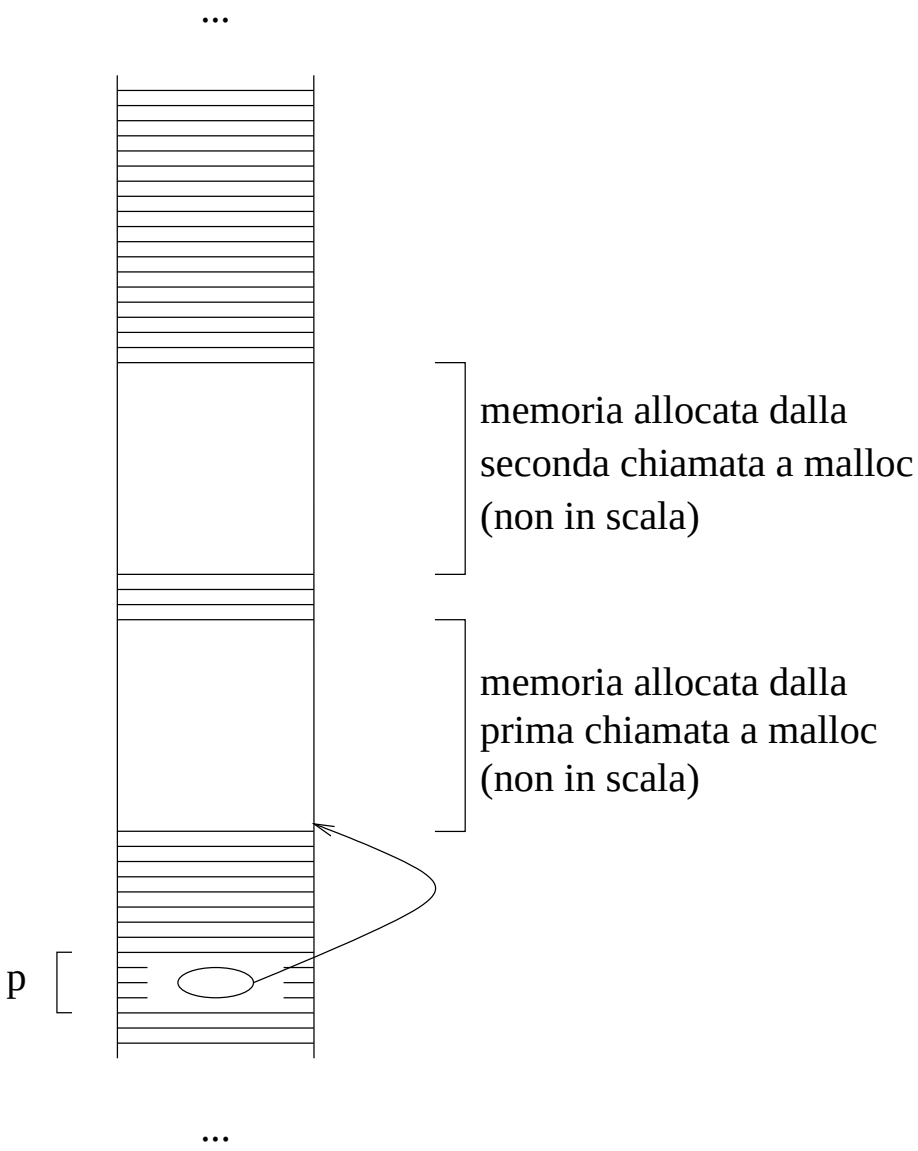
```
/*
  Alloca memoria e la libera.
*/

#include<stdlib.h>

int main() {
  int i;
  int *p;
```

```
  for(i=0; i<=100000; i++) {

                /* alloca cinquantamila interi */
     p=malloc(50000*sizeof(int));
     if(p==NULL) {
       printf("Impossibile allocare altra memoria\n");
       exit(1);
     }


                /* libera la memoria */
     free(p);
  }

  return 0;
}
```

La differenza fra il primo programma e il secondo è che il primo programma continua a riservare zone di memoria, anche se non vengono più usate. Per esempio, dopo la prima chiamata alla funzione `malloc` la situazione in memoria potrebbe essere la seguente:

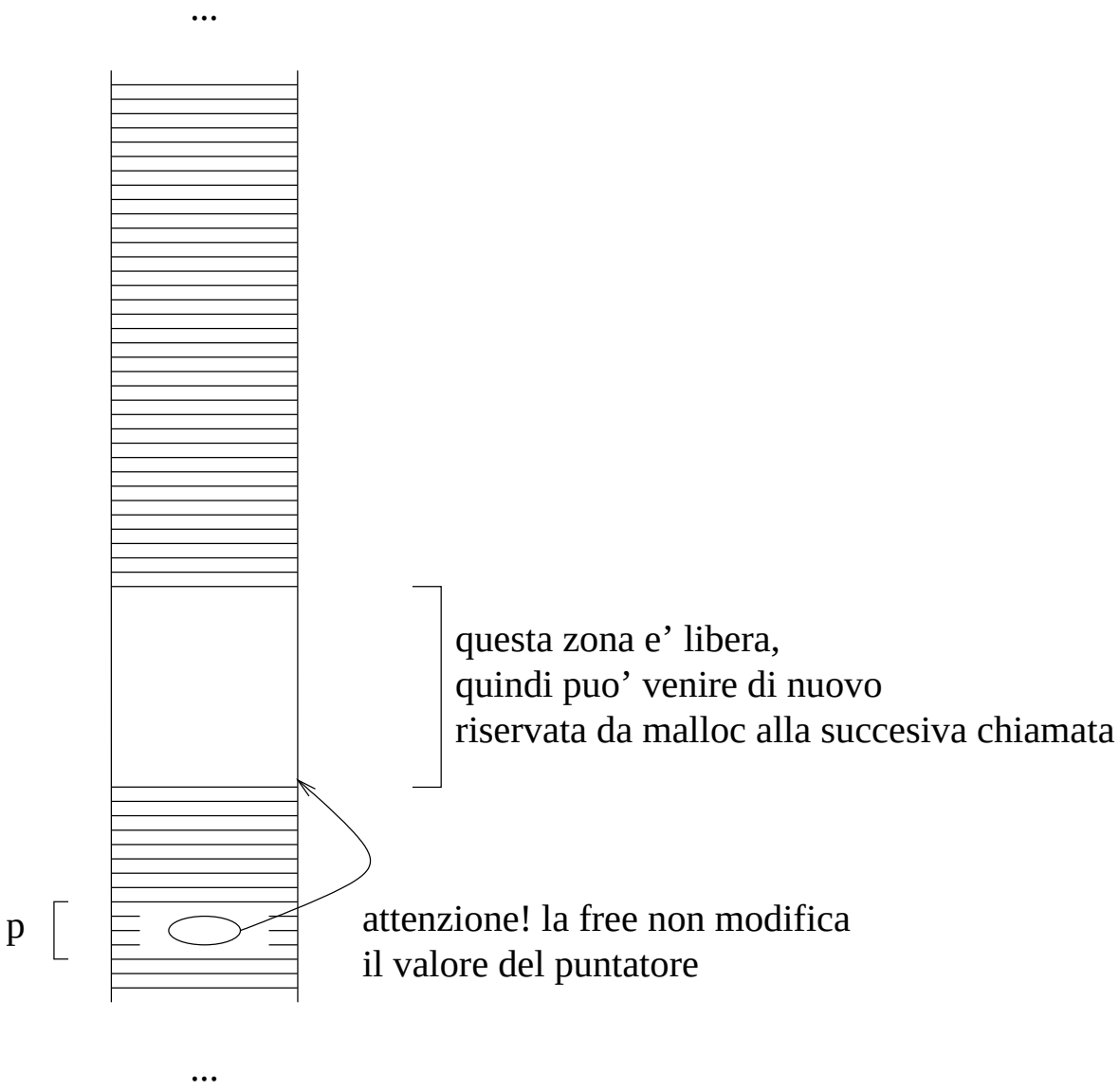


Quando si fa la seconda chiamata, la memoria che è stata allocata in precedenza risulta riservata, quindi la funzione `malloc` è costretta a usare una differente zona di memoria, che viene a sua volta riservata.



È chiaro che, se si continuano ad allocare sempre nuove zone di memoria, alla fine non ci sarà più memoria disponibile.

Al contrario, il secondo programma libera la memoria. Quindi, dopo la prima chiamata alla funzione `malloc`, e dopo la chiamata alla funzione `free`, la zona allocata viene marcata come ‘‘non più usata’’, e quindi può venire allocata dalla successiva chiamata alla funzione `malloc`.



Si può quindi dire che la memoria libera (non allocata, ossia che non è stata già riservata) è tornata ad essere quella che era prima della malloc.

Nota: la funzione `free`, dal momento che il puntatore viene passato per valore, non può modificare il valore del puntatore, quindi *il puntatore non vale `NULL` dopo la chiamata*. Nel puntatore c'è ancora l'indirizzo della zona precedentemente allocata. Dal momento che questa zona è ora libera, può venire riallocata e modificata, quindi ogni accesso a questa zona (usando `p[i]` oppure `*p`) è da considerarsi un errore di programmazione.

Riallocazione di memoria

Riallocazione di memoria

L'uso congiunto delle funzioni `malloc` e `free` consente di aumentare la zona di memoria associata ad un vettore (dinamico). Supponiamo infatti di aver allocato un vettore di cento elementi `p`, e ci accorgiamo che questi cento elementi non bastano. Quello che possiamo fare è:

1. creiamo un vettore di duecento elementi
2. copiamo i cento elementi del vettore nella nuova zona di memoria
3. liberiamo la vecchia zona di memoria puntata da `p`
4. assegnamo a `p` l'indirizzo della nuova zona di memoria

È facile rendersi conto che questa sequenza di istruzioni ci porta ad avere che `p` punta a una zona di memoria di duecento elementi, i cui primi cento hanno lo stesso valore dei cento elementi dell'array originario. Abbiamo quindi realizzato l'allungamento dell'array.

In C, è possibile realizzare la stessa cosa usando la funzione `realloc`. In prima approssimazione, possiamo dire che la funzione fa queste operazioni di nuova allocazione, copia, e rilascio della vecchia zona di memoria. In pratica, la funzione viene sempre usata in questo modo:

```
p=realloc(p, numero_byte);
```

Ossia si passa come prima parametro un puntatore, e si mette il risultato nello stesso puntatore. L'effetto è quello di aver aumentato la dimensione della zona puntata da `p`.

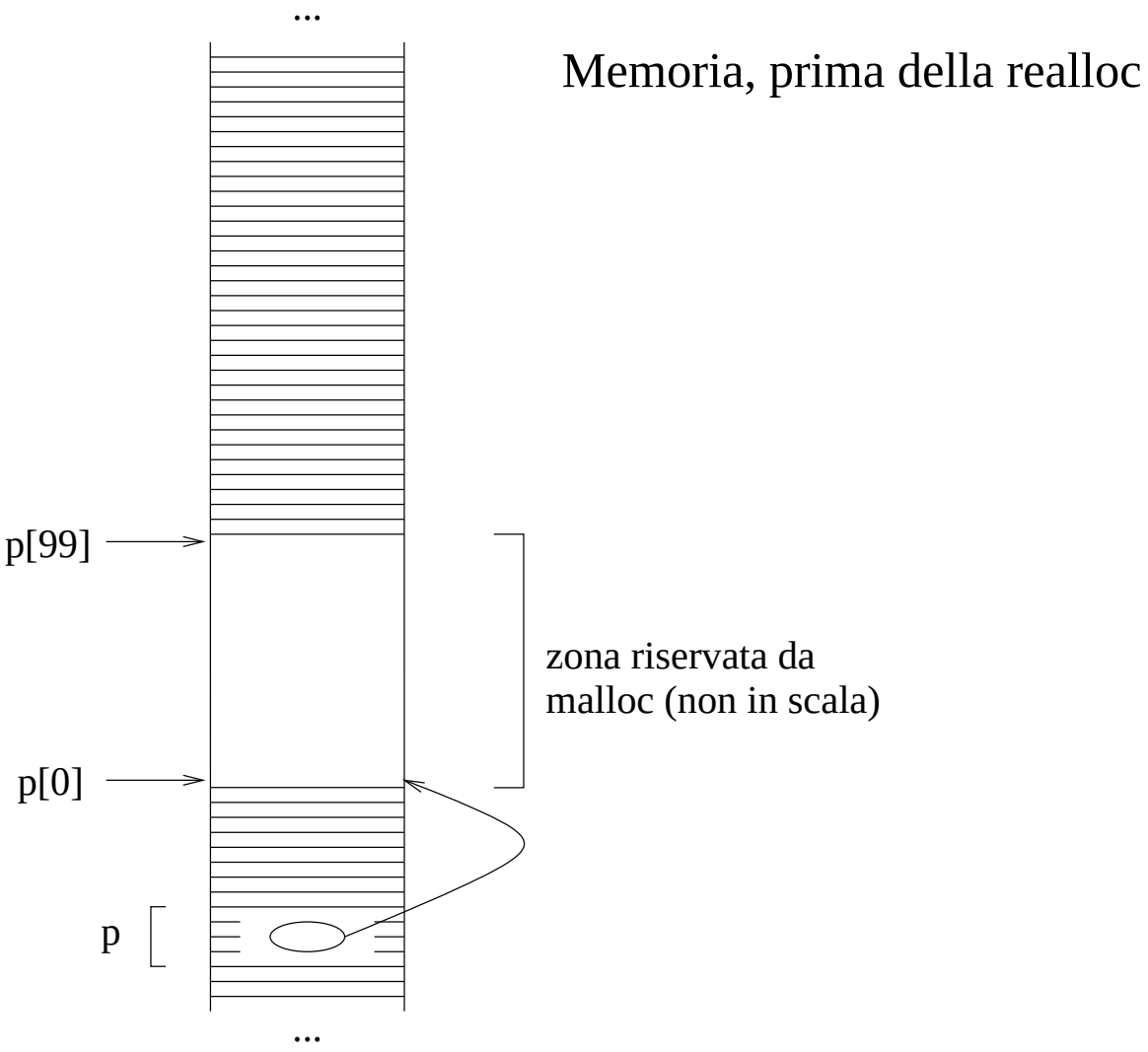
Il seguente programma realloces.c mostra un esempio di uso di questa funzione. Inizialmente si alloca una zona di cento interi, e quindi sono corrette solo le operazioni su `p[0]...p[99]`, mentre l'uso o assegnazione di `p[100]`, `p[101]` ecc sono scorrette perchè fuori dalla zona assegnata. Dopo aver fatto la realloc, la zona assegnata è diventata grande abbastanza da contenere duecento interi, per cui ora le operazioni su `p[0]...p[199]` sono tutte valide.

Per concludere, notiamo come la funzione `realloc` si può anche chiamare più volte sullo stesso puntatore. Nell'esempio di sotto, nel caso in cui duecento elementi non fossero stati sufficienti, si poteva fare una nuova realloc per allocarne 300, poi ancora una volta per allocarne 400, ecc.

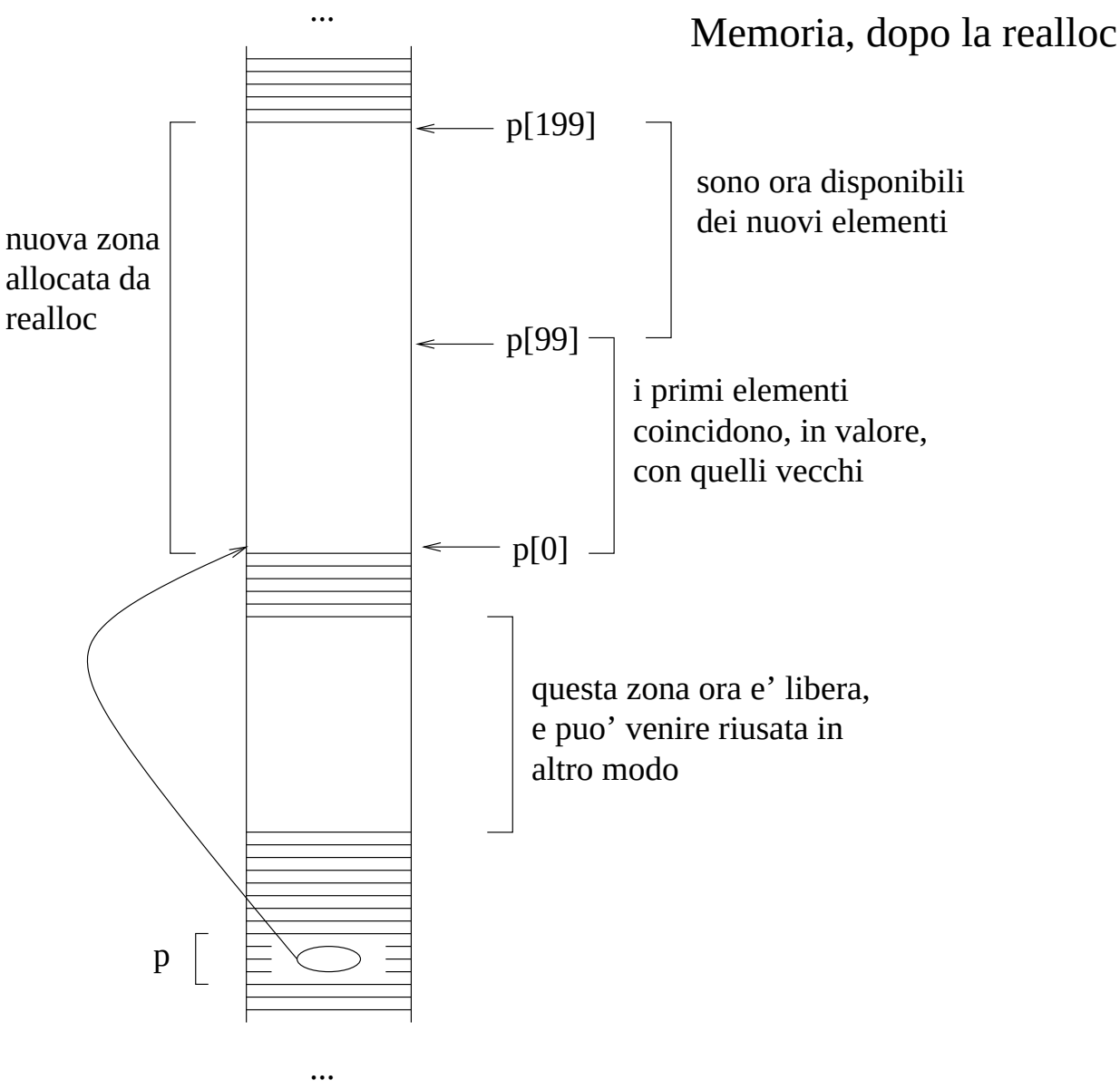
```
/*
  Alloca un array, e poi lo rialloca.
*/

#include<stdlib.h>

int main() {
  int *p;
  int i;

                /* allocazione iniziale */
  p=malloc(100*sizeof(int));


                /* operazione corretta */
  p[20]=1;
  p[99]=-12;


                /* operazione scorrette: p[112] non e' nella
                zona di memoria riservata dalla malloc */
  p[112]=-32;


                /* riallocazione */
  p=realloc(p, 200*sizeof(int));


                /* ora si puo' usare fino a p[199] */
```

```
  p[112]=-32;   /* operazione corretta */


                /* la parte precedentemente allocata del vettore
                rimane inalterata */
  printf("p[20]=%d p[99]=%d p[112]=%d\n", p[20], p[99], p[112]);

  return 0;
}
```

Dopo la prima allocazione, la situazione in memoria è la seguente:



Dopo la riallocazione, è stata creata una nuova zona di memoria che può contenere duecento elementi, e i valori della zona di memoria precedente sono stati copiati. La vecchia zona di memoria è anche stata liberata.



Lettura array con allocazione di memoria

Lettura array con allocazione di memoria

Il seguente programma legge un array da file. Il file ha questo formato: il primo intero è la dimensione dell'array, ossia il numero di elementi dell'array. Seguono gli elementi dell'array.

Si noti che non è in generale possibile memorizzare l'array in un vettore statico, dal momento che il numero di elementi che deve contenere non è noto in fase di compilazione. Infatti, questo numero risulta noto solo quando il file viene aperto e il primo intero viene letto. Si può usare un array statico soltanto quando esiste già, in fase di compilazione, una stima del massimo numero di elementi.

Il programma seguente leggiarray.c legge quindi il primo intero dal file, e lo memorizza nella variabile `n`. Dal momento che questo numero è la dimensione che l'array deve avere, possiamo ora allocare l'array con la funzione `malloc`. Dal momento che si tratta di `n` interi, allochiamo `n*sizeof(int)` byte. A questo punto, possiamo leggere, uno per volta, gli elementi dell'array da file.

```
/*
  Legge un array da file.
*/

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

int main() {
  int n;
  int *vett;
  FILE *fd;
  int i;

                /* apre il file */
  fd=fopen("array.txt", "r");

                /* verifica errori in apertura */
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }

                /* legge il numero di elementi del vettore */
  fscanf(fd, "%d\n", &n);


                /* alloca il vettore */
  vett=malloc(n*sizeof(int));


                /* legge il vettore */
  for(i=0; i<=n-1; i++)
    fscanf(fd, "%d\n", &vett[i]);


                /* chiude il file */
  fclose(fd);


                /* stampa l'array */
  for(i=0; i<=n-1; i++)
    printf("%d\n", vett[i]);

  return 0;
}
```

Da notare che il programma non controlla se effettivamente sul file ci sono `n` elementi dopo il primo. Questo potrebbe però accedere se c'è stato un errore nella realizzazione del file. Inoltre, non c'è nessun controllo sulla conversione, ossia il programma non si accorge se in mezzo al file ci sono per esempio dei caratteri alfabetici, che non sono chiaramente interpretabili come interi.

Per rendere il programma *robusto* (in grado di rilevare errori nei dati di input) occorre fare un controllo sul valore di ritorno della funzione `fscanf`.

Il programma dovrebbe inoltre fare un controllo sul valore di ritorno della funzione `malloc`, per controllare se effettivamente siamo riusciti ad allocare gli elementi richiesti (la allocazione fallisce se non c'è più memoria a disposizione). Il programma modificato leggiarrayrob.c è il seguente.

```
/*
  Legge un array da file.
*/

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

int main() {
  int n;
  int *vett;
  FILE *fd;
  int res;
  int i;

                /* apre il file */
  fd=fopen("array.txt", "r");

                /* verifica errori in apertura */
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }

                /* legge il numero di elementi del vettore */
  res=fscanf(fd, "%d\n", &n);
  if(res!=1) {
    printf("Non riesco a leggere il numero di elementi del vettore\n");
    exit(1);
  }


                /* alloca il vettore */
  vett=malloc(n*sizeof(int));
  if(vett==NULL) {
    printf("Non riesco ad allocare memoria\n");
    exit(1);
  }


                /* legge il vettore */
  for(i=0; i<=n-1; i++) {
    res=fscanf(fd, "%d\n", &vett[i]);
    if(res!=1) {
      printf("Non riesco a leggere l'elemento %d-esimo del vettore\n", i);
      exit(1);
    }
  }


                /* chiude il file */
  fclose(fd);


                /* stampa l'array */
  for(i=0; i<=n-1; i++)
    printf("%d\n", vett[i]);

  return 0;
}
```

Lettura array con allocazione di memoria

Lettura array con allocazione di memoria

Una variante del programma di lettura di array da file è la seguente: supponiamo che gli elementi dell'array siano allocati su file in ordine, ma che non ci sia un primo elemento che dice quanto è grande l'array. In altre parole, il file contiene il primo elemento, il secondo, ecc, fino all'ultimo elemento e poi il file finisce.

Per un essere umano non è difficile capire quanti elementi ci sono sul file. Se per esempio il file contiene:

```
-12
3
32
12
```

Allora è chiaro che il vettore deve essere composto da quattro elementi. Il problema è fare in modo che questo numero venga calcolato dal programma.

Una possibile strategia è la seguente: apriamo il file in lettura, e cominciamo a leggere interi fino alla fine del file. Se ogni volta che leffiamo un intero incrementiamo un contatore, allora alla fine della scansione del file questo contatore contiene il numero di interi che si trovano nel file.

Una volta noto questo numero si può procedere come nel programma precedente: si alloca un vettore di `n*sizeof(int)` elementi. Ora dobbiamo di nuovo leggere il file dall'inizio, per cui lo chiudiamo e lo riapriamo di nuovo. A questo punto possiamo leggere gli elementi del file uno per volta. Il programma finale leggiarrayeof.c è qui sotto.

```
/*
  Legge un array da file, fino all'end-of-file.
*/

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

int main() {
  int n;
  int *vett;
  FILE *fd;
  int res;
  int i;

                /* apre il file */
  fd=fopen("array.txt", "r");


                /* verifica errori in apertura */
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


                /* legge il numero di elementi del vettore */
  n=0;
  while(1) {
    res=fscanf(fd, "%d\n", &i);
    if( res==EOF )
```

```
      break;
    else
      n++;
  }

  printf("Il file contiene %d elementi\n", n);


                /* chiude il file */
  fclose(fd);


                /* riapre il file */
  fd=fopen("array.txt", "r");

  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


                /* alloca il vettore */
  vett=malloc(n*sizeof(int));


                /* legge il vettore */
  for(i=0; i<=n-1; i++)
    fscanf(fd, "%d\n", &vett[i]);


                /* chiude il file */
  fclose(fd);


                /* stampa l'array */
  for(i=0; i<=n-1; i++)
    printf("%d\n", vett[i]);

  return 0;
}
```

Questo programma non controlla se la operazione di lettura fallisce a causa della fine del file oppure a causa di caratteri non interpretabili come interi. Modificare il programma in modo da renderlo robusto.

Lettura array con riallocazione di memoria

Lettura array con riallocazione di memoria

Vediamo ora una variante del programma di lettura di array da file, in cui si fa una sola scansione del file.

Dal momento che non sappiamo quanti elementi stanno nel file, facciamo una prima ipotesi, per esempio che ci siano 1024 elementi. Allochiamo quindi un vettore che contiene 1024 interi, e iniziamo la lettura.

Quando si arriva a scrivere l'ultimo elemento allocato del vettore, sappiamo che non c'è più spazio nel vettore per contenere altri elementi. Quello che ci serve è un vettore più grande. Possiamo usare le funzione `realloc` per ottenere un vettore più grande, senza perdere i dati già letti.

In practica, ci serve sempre sapere quanti elementi sono stati allocati nel vettore. Usiamo una variabile intera `dim` per memorizzare questo numero. In questo modo, possiamo sempre sapere se abbiamo ancora spazio oppure no. Nel secondo caso, oltre ad allocare nuova memoria, dobbiamo anche aggiornare il valore di `dim`, dal momento che il numero di elementi a disposizione è cambiato. Questo ci permette di capire quando anche la nuova memoria è finita e ne serve ancora altra.

Il programma completo rialloca.c è qui sotto.

```
/*
  Legge un array da file, fino all'end-of-file.
Usa la realloc nel caso in cui non ci sia piu'
spazio nel vettore.
*/

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

int main() {
  int n;
  int *vett;
  int dim;
  FILE *fd;
  int res;
  int i;


                /* allocazione iniziale del vettore */
  dim=1024;
  vett=malloc(dim*sizeof(int));


                /* apre il file */
  fd=fopen("array.txt", "r");
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


                /* legge il vettore */
  n=0;
  while(1) {
    res=fscanf(fd, "%d\n", &vett[n]);
    if( res!=1 )
      break;

    n++;
    if( n>dim-1 ) {
      dim+=1024;
      vett=realloc(vett, dim*sizeof(int));
    }
  }


                /* stampa l'array */
  for(i=0; i<=n-1; i++)
    printf("%d\n", vett[i]);


                /* chiude il file */
```

```
  fclose(fd);

  return 0;
}
```

In pratica, gli array con riallocazione funzionano sempre nello stesso modo: abbiamo una variabile intera che rappresenta il numero di elementi allocati, e un puntatore alla zona di memoria in cui sono memorizzati gli elementi. Ogni volta che ci si accorge che lo spazio nel vettore non basta più, si modifica la variabile intera con la nuova dimensione, e si fa la riallocazione. In questo modo, quando anche la nuova memoria non è più sufficiente, possiamo fare una nuova riallocazione, ecc.

Serie di Fibonacci

Serie di Fibonacci

Si risolva il seguente problema: sia dato da tastiera un numero intero `n`. Calcolare i valori della serie di Fibonacci fino all' `n`-esimo elemento, e memorizzarli in un vettore. La serie di Fibonacci è caratterizzata cosí: i primi due elementi della serie valgono 1. Ogni successivo elemento della serie si ottiene sommando i due precedenti elementi.

Il problema si risolve leggendo il numero `n` da tastiera, e allocando un vettore di `n` elementi interi. I primi due elementi della serie devono valere 1, e mettiamo 1 nei primi due elementi del vettore. Per calcolare i valori successivi, facciamo un ciclo che parte da 2 e arriva al valore di `n` meno uno (infatti, per memorizzare `n` valori, usiamo gli indici da 0 a `n-1`. Per ognuno di questi indici, l'elemento del vettore si calcola semplicemente sommando i due elementi precedenti.

Questo è il programma completo fibonacci.c

```
/*
  Calcola gli elementi della serie di
Fibonacci e li mette in un vettore.
*/

#include<stdlib.h>

int main() {
  int *fib;
  int n;
  int i;


                /* lettura del valore di n */
  scanf("%d", &n);


                /* allocazione vettore */
  fib=malloc(n*sizeof(int));


                /* calcolo serie */
  fib[0]=1;
  fib[1]=1;

  for(i=2; i<=n-1; i++)
    fib[i]=fib[i-1]+fib[i-2];
```

```
                /* stampa serie */
  for(i=0; i<=n-1; i++)
    printf("%d\n", fib[i]);

  return 0;
}
```

Consideriamo ora la seguente variante: il numero letto da tastiera non indica più il numero di elementi da calcolare, ma il massimo valore che ci interessa. In altre parole, leggiamo un numero intero `max` da tastiera, e il ciclo di calcolo degli elementi della serie si interrompe quando si trova un valore che supera `max`. Detto in un altro modo ancora: quando `fib[n]` supera `max` si esce dal ciclo.

In questo caso, non sappiamo in partenza quanti elementi servono per memorizzare tutta la serie. Infatti, il numero `max` non è il massimo indice del vettore. Il massimo indice si può trovare solo dopo aver calcolato tutti gli elementi fino a quello che supera `max`.

Risolviamo il problema facendo una allocazione iniziale di dieci elementi. Se a un certo punto ci rendiamo conto che non sono sufficienti per memorizzare tutto l'array, facciamo una riallocazione con `realloc`. Dal momento che non sappiamo quante iterazioni sono necessarie, usiamo un ciclo while. All'interno di questo ciclo facciamo il calcolo di un elemento del vettore, e incrementiamo il valore della variabile `n` che ci dà l'indice dell'elemento che stiamo calcolando. Se questo numero raggiunge la dimensione del vettore (numero di elementi correntemente allocati), allora vuol dire che il prossimo elemento che verrà calcolato sarà fuori dalla zona di memoria allocata, per cui dobbiamo fare un riallocazione.

Di seguito si riporta il programma completo refib.c

```
/*
  Calcola gli elementi della serie di
Fibonacci e li mette in un vettore.
Riallocazione del vettore.
*/

#include<stdlib.h>

int main() {
  int dim;
  int *fib;

  int max;
  int n;
  int i;


                /* lettura del valore di n */
  scanf("%d", &max);


                /* allocazione vettore */
  dim=10;
  fib=malloc(dim*sizeof(int));


                /* calcolo serie */
  fib[0]=1;
  fib[1]=1;

  n=1;
```

```
  while(fib[n]<max) {
    n++;                /* nuovo indice */

    if(n>=dim) {        /* se il vettore non basta, rialloca */
      dim+=10;
      fib=realloc(fib, dim*sizeof(int));
    }

    fib[n]=fib[n-1]+fib[n-2];   /* calcolo elemento */
  }


               /* stampa serie */
  for(i=0; i<=n-1; i++)
    printf("%d\n", fib[i]);

  return 0;
}
```

Salva valori di una funzione

Salva valori di una funzione

Un possibile uso dei vettori è quello di memorizzare i valori che una funzione assume in un certo intervallo. Si supponga per esempio di voler stampare alcuni valori di una funzione in un certo intervallo, e poi di volere il valore medio, minimo e massimo. Il modo più semplice è quello di scrivere prima le istruzioni di stampa di una funzione, poi scrivere delle istruzioni che calcolano la media, poi quelle che calcolano il massimo, ecc.

Questo tipo di approccio ha lo svantaggio che i valori della funzione vengono calcolati una prima volta per la stampa, una seconda volta per fare la somma, ecc. Questo non è un problema per le funzioni viste fino ad ora, che sono facili da valutare per il calcolatore. D'altra parte, esistono funzioni la cui valutazione può richiedere molto tempo.

Per evitare di ripetere più volte le stessa operazioni di calcolo della funzione, è possibile memorizzare alcuni valori calcolati in un vettore, per poi poterli usare in seguito quando servono senza dover ogni volta valutare di nuovo la funzione.

Per fare questo, il primo passo è decidere quali sono i valori che potrebbero essere utili in seguito. Usiamo una variabile `maxindice` che dice che gli elementi `f(0) ... f(maxindice)` saranno molto usati nel seguito. Nel seguito vediamo solo il programma con allocazione iniziale, ma è facile da modificare nel caso in cui si voglia fare una riallocazione.

Dal momento che i valori di *f(x)* con *x* intero che va da `0` a `maxindice` saranno molto usati, è utile calcolare questi valori una volta sola all'inizio e memorizzarli in un vettore. Dal momento che da `0` a `maxindice` ci sono `maxindice+1` numeri interi, il vettore deve avere dimensione `maxindice+1`. Se la funzione dà risultati reali, il tipo del vettore deve essere `float`:

```
  float *fvett;
  int maxindice=100;

  fvett=malloc((maxindice+1)*sizeof(int));
```

La successiva operazione è quella di calcolare il valore di *f(x)* con *x* che va da `0` a `maxindice`, e memorizzare questi valori nel vettore. Per fare questo è necessaria una variabile intera `x` e un ciclo for.

```
  int x;

        /* inizializza il vettore */
  for(x=0; x<=100; x=x+1)
    fvett[x]=x*x/2+sqrt(abs(x))-12;
```

A questo punto, ogni volta che ci serve il valore di *f(x)*, e *x* ha una valore intero compreso fra `0` e `maxindice`, possiamo utilizzare i valori memorizzati nel vettore. Per esempio, se vogliamo la media dei valori della funzione fra 10 e 110, calcoliamo la somma e dividiamo per il numero di punti:

```
                /* valore medio della funzione da 10 a 110 */
  somma=0;
  for(x=10; x<=110; x=x+1)
    if( (x>=0) && (x<=maxindice) )
      somma=somma+fvett[x];
    else
      somma=somma+x*x/2+sqrt(abs(x))-12;

  printf("La media fra 10 e 110 vale %f\n", somma/101);
```

Queste istruzioni non differiscono da quelle di calcolo di una media. L'unica differenza è che ogni volta che si vuole il valore *f(x)*, prima di tutto si controlla se *x* sta fra `0` e `maxindice`. Se ci sta, si usa il valore memorizzato nel vettore. Se non c'è, si calcola la funzione usando la sua definizione.

Il programma completo salvafunzione.c contiene anche la stampa dei valori della funzione fra -200 e +200, sempre cercando di usare i valori memorizzati.

```
/*
  Salva alcuni valori di una funzione in un vettore,
  per poterli usare di nuovo quando servono.
*/

#include<stdlib.h>
#include<math.h>


int main () {
  float *fvett;
  int maxindice=100;

  int x;
  float somma;


                /* allocazione */
  fvett=malloc((maxindice+1)*sizeof(int));


                /* inizializza il vettore */
  for(x=0; x<=maxindice; x++) {
    fvett[x]=x*x/2+sqrt(abs(x))-12;
  }


                /* valore medio della funzione da 10 a 110 */
  somma=0;
  for(x=10; x<=110; x++)
```

```
    if( (x>=0) && (x<=maxindice) )
      somma=somma+fvett[x];
    else
      somma=somma+x*x/2+sqrt(abs(x))-12;

  printf("La media fra 10 e 110 vale %f\n", somma/101);


                /* stampa i valori della funzione da -200 a +200 */
  for(x=-200; x<=200; x++)
    if( (x>=0) && (x<=maxindice) )
      printf("%f\n", fvett[x]);
    else
      printf("%f\n", x*x/2+sqrt(abs(x))-12);

  return 0;
}
```

stampare l’inverso di un file