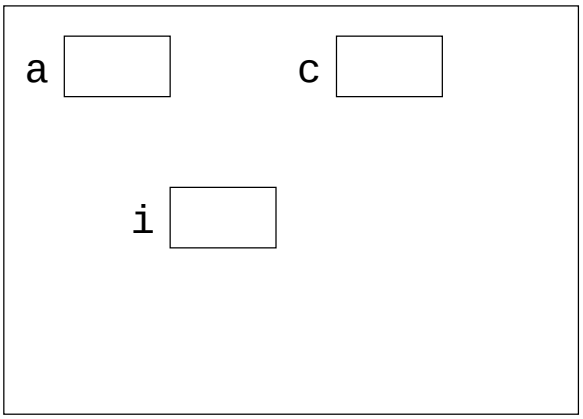
Memoria e puntatori

In questo capitolo vediamo come è fatta, e come viene usata, la memoria. La memoria si può vedere come un array di byte, in cui ogni variabile occupa un certo numero di byte consecutivi. In C, è possibile determinare sia quanto spazio occupa una variabile, sia la sua posizione in questo vettore. In particolare, è possibile memorizzare la posizione (indirizzo) di una variabile in un'altra variabile. Le variabili che contengono indirizzi di memoria sono dette variabili puntatori.

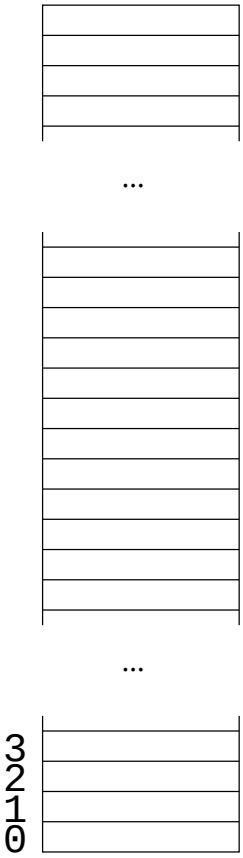
La memoria

1. come si rappresenta la memoria ad alto livello
2. rappresentazione più precisa, come vettore di celle
3. un esempio di alcune variabili in memoria

La memoria di un calcolatore è un dispositivo in cui è possibile memorizzare dei dati. Ogni volta che si crea (dichiara) una nuova variabile, viene riservata un'area di memoria in cui viene memorizzato il valore della variabile. In altre parole, una parte della memoria viene impiegato per la memorizzazione del valore della variabile.



Per i programmi più semplici, si può pensare alla memoria come se fosse una lavagna: ogni volta che si dichiara una variabile, si disegna un quadrato; all'interno di questo quadrato si può scrivere (memorizzare) un valore.



Il modello semplificato di memoria come lavagna funziona bene solo per alcuni tipi di programmi. Esistono delle tecniche di programmazione che richiedono un modello più dettagliato di come è fatta la memoria.

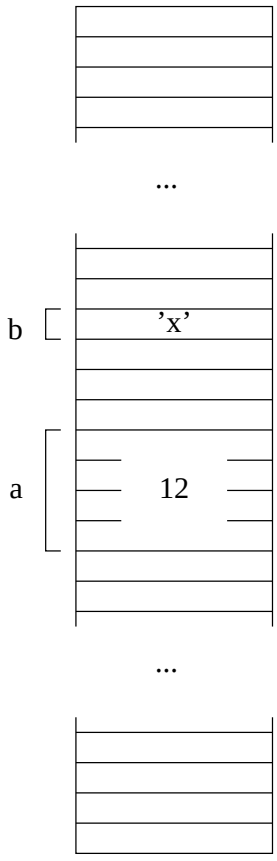
Per gli argomenti che si vedranno in questo corso, è sufficiente dire che la memoria è un vettore di byte. Non ci interessa la sua dimensione.

Va notato che anche questo modello della memoria è una semplificazione, che comunque è sufficientemente preciso per gli argomenti trattari in questo corso.

Nel seguito, rappresentiamo la memoria come nella figura accanto, in cui ogni rettangolo è un byte. Gli indici di questo vettore verranno scritti solo quando necessari.

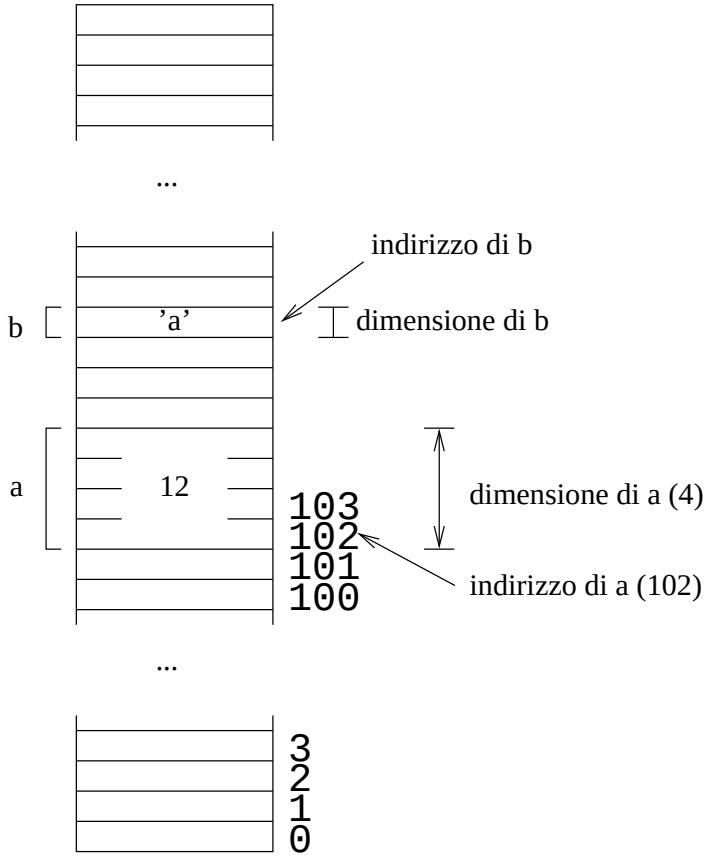
Ogni volta che si dichiara una variabile, viene allocata (riservata) una zona di memoria per essa. La grandezza di questa zona dipende dal tipo della variabile: per esempio, le variabili di tipo `char` occupano un solo byte, mentre gli interi ne occupano 4.

Il disegno mostra come potrebbe essere strutturata la memoria se un programma usa una variabile intera `a` e una variabile carattere `b`: la variabile `a` occupa quattro byte consecutivi nella memoria, mentre la variabile `b` ne occupa uno solo. Ogni volta che si assegna ad `a` un valore, questo viene scritto nei byte che le sono assegnati. Ogni volta che il valore di `a` viene usato (per esempio per fare una stampa del valore di `a`, come `printf("%d\n",a);`, o per fare un calcolo come `b=a+c;`), il calcolatore va a cercare il valore in quei quattro byte.



Indirizzo, byte occupati e valore di una variabile

1. variabile = insieme di celle consecutive
2. per sapere quali celle sono occupate da una variabile, ci basta sapere quante sono e l'indirizzo della prima
3. ci sono casi in cui serve sapere queste cose
4. operatori `&` e `sizeof`
5. indirizzo e numero di byte si possono anche stampare
6. differenza fra valore e indirizzo
7. `sizeof` dipende solo dal tipo; per questa ragione, si può usare anche sul tipo



Le variabili C sono zone di memoria. In altre parole, ogni variabile è un insieme di locazioni all'interno della memoria. Il numero di byte occupati da una variabile dipende dal suo tipo: un intero occupa quattro posizioni, un carattere una posizione, un `double` otto, ecc.

Se vogliamo sapere quali byte una variabile occupa, ci servono due numeri: il primo è la posizione del primo byte occupato, il secondo è il numero di byte occupati. Il primo numero viene detto *indirizzo* della variabile. Se per esempio la variabile `a` occupa in memoria le posizioni dalla 1243 alla 1247, allora il suo indirizzo è il primo di questi numeri, cioè 1243; questa variabile occupa quattro locazioni, quindi il numero di byte occupati da `a` è 4.

Nei programmi fatti fino a questo momento, non era necessario sapere quali celle una variabile occupa. Esistono però delle situazioni in cui è invece necessario. Per questo, il C mette a disposizione delle primitive che permettono di trovare l'indirizzo e il numero di byte occupati dalle variabili. Queste nozioni sono necessarie per esempio nel caso in cui il programma deve gestire un numero di dati non noto a priori, come si vedrà a proposito degli array e delle liste.

indirizzo
: per sapere l'indirizzo di una variabile si usa l'operatore unario `&`; in altre parole, se `a` è una variabile, allora `&a` è il suo indirizzo (la prima posizione di memoria occupata da essa);

numero di byte occupati
: il numero di byte occupati da una variabile si trova con `sizeof`; quindi per esempio `sizeof(a)` è il numero di byte occupati dalla variabile `a`.

Il programma seguente dichiara tre variabili di diversi tipi, assegna dei valori, e poi stampa i loro indirizzi e il numero di byte che occupano. Da notare che l'indirizzo di una variabile, cosí come il numero di byte occupati, sono dei normali numeri, e si possono quindi stampare. Nel caso dell'indirizzo, si è scelto di stamparlo in esadecimale (usando `%x`) ma anche la stampa in decimale avrebbe funzionato (da notare che gli indirizzi sono numeri unsigned, e quindi è comunque più appropriato stamparli usando `%u` che `%d`).

```
/*
  Stampa indirizzo, occupazione di memoria e valore
  di alcune variabili.
*/

int main(void) {
  int a;
  char b;
  float c;

  a=12;
  b='a';
  c=0.1243;

  printf("L'indirizzo di a e' %x, occupa %d bytes, il suo valore e' %d\n",
         &a, sizeof(a), a);

  printf("L'indirizzo di b e' %x, occupa %d bytes, il suo valore e' %c\n",
         &b, sizeof(b), b);

  printf("L'indirizzo di c e' %x, occupa %d bytes, il suo valore e' %f\n",
         &c, sizeof(c), c);


  return 0;
}
```

È importante notare la differenza tra il *valore* di una variabile e il suo *indirizzo*. L'indirizzo di una variabile è l'inizio della zona di memoria occupata da una variabile, mentre il valore di una variabile è il contenuto di tale zona.

Altra osservazione: tutte la variabili di un certo tipo occupano esattamente lo stesso numero di byte. Quindi, se `a` e `x` sono interi, si può essere certi che `sizeof(a)` è uguale a `sizeof(x)`. Per questa ragione, `sizeof` è stata definita in modo che possa avere come parametro sia il nome di una variabile che il nome di un tipo. È quindi possibile per esempio calcolare e stampare il valore di `sizeof(int)`, che coincide necessariamente con i valori di `sizeof` di ogni variabile di tipo `int`.

Il programma dimensione.c stampa il numero di byte occupati dalle variabili di vari tipi di dato. Il numero di byte occupati da un tipo di dato può cambiare passando a calcolatori di tipo diverso. Per esempio, un intero può occupare due, oppure quattro, oppure otto byte a seconda del tipo di calcolatore/sistema operativo/compilatore usato.

```
/*
  Spazio occupato da variabili di un certo tipo.
*/

int main() {

  printf("Il tipo int occupa %d bytes\n", sizeof(int) );

  printf("Il tipo char occupa %d bytes\n", sizeof(char) );
```

```
  printf("Il tipo float occupa %d bytes\n", sizeof(float) );

  printf("Il tipo double occupa %d bytes\n", sizeof(double) );

  return 0;
}
```

Puntatori

1. gli indirizzi non sono esattamente dei numeri
2. per esempio, `int p, a;` e poi `p=&a;` genera un errore (warning)
3. il *tipo* di un indirizzo non è `int`
4. l'indirizzo di una variabile `int` è di tipo `int *`
5. quindi, `int *p; int a;` e poi `p=&a;` non genera errori

Nella pagine precedente si è detto che l'indirizzo, cosí come lo spazio occupato da una variabile, sono dei semplici numeri. Questo è in realtà vero solo per il numero di byte occupati, che è realmente un numero intero.

Per quello che riguarda l'indirizzo, non è del tutto esatto che si tratta di un numero. Per capire meglio questo fatto, ricordiamo che una variabile, oltre ad avere una zona di memoria associata, ha anche un tipo. Per esempio, due variabili `a` e `b` possono entrambe occupare quattro byte in memoria, ma essere una di tipo `int` e una di tipo `float`. Quello che cambia non è la occupazione di memoria, ma il modo in cui questi quattro byte vengono interpretati.

Questa differenza si riflette anche sui tipi degli indirizzi. Per esempio l'indirizzo di una variabile di tipo `int` non è un intero. È invece una variabile di un nuovo tipo, il *puntatore a intero*. In pratica, questo si può vedere immediatamente se si cerca di compilare il seguente programma nocast.c.

```
/*
  Dimostrazione che l'indirizzo di una variabile
non e' un numero intero: compilando questo
programma si genera un errore (warning) di
incompatibilita' fra tipi.
*/

int main() {
  int a;
  int b;

  b=&a;

  return 0;
}
```

Compilando, si ottiene un messaggio di questo genere:

```
nocast.c: In function 'main':
nocast.c:12: warning: assignment makes integer from pointer without a cast
```

Questo è il tipico errore che si ottiene quando si cerca di assegnare a una variabile un valore di un altro tipo (per esempio, assegnando a una variabile intera un valore reale). È quindi chiaro che la istruzione `b=&a` contiene un tentativo di assegnare a una variabile intera un valore di un tipo non compatibile. Quindi, il tipo della espressione `&a` non è compatibile con gli interi.

La regola generale sui puntatori è la seguente:

l'indirizzo di una variabile di un tipo `T` è di tipo `T *`, che viene detto tipo ''puntatore a `T`''.

Dato che `a` è un intero, il suo indirizzo è di tipo `int *`, ossia di tipo ''puntatore a `int`''. A prima vista, questa differenza di tipi può sembrare una particolarità inutile del C. Il motivo per cui è invece necessaria risulterà chiara nel seguito.

Per ogni tipo che è possibile definire in C, ad esso viene associato il corrispondente tipo puntatore, che è anche esso un tipo. Per esempio, dato che in C esiste il tipo `int`, esiste automaticamente anche il tipo puntatore ad `int`, che si denota con `int *`. È importante notare che questo è un tipo come tutti gli altri. È quindi possibile definire delle variabili di tipo puntatore a intero, nel seguente modo:

```
  int *p;
```

Dal momento che gli indirizzi di variabili sono di tipo puntatore (al tipo della variabile), nella variabile `p` si può mettere l'*indirizzo* di una variabile di tipo int. Questa operazione assegna a una variabile di tipo puntatore a intero `p` il valore dell'indirizzo di una variabile intera, che è anche esso un puntatore a intero. Quindi, la istruzione `p=&a` non genera nessun errore di incompatibilitè fra tipi.

La compilazione del programma nocastpunt.c, programma del resto inutile, non genera nessun errore. Si riporta qui sotto il testo del programma.

```
/*
  Una variabile definita di tipo int * e' un puntatore a
intero. Anche l'indirizzo di una variabile intera e' un
puntatore a intero. Quindi, p=&a non genera nessun errore,
perche' p e &a sono dello stesso identico tipo.
*/

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

  p=&a;

  return 0;
}
```

Se vogliamo memorizzare l'indirizzo di una variabile di tipo `int` in un'altra variabile `p`, quest'ultima deve essere dichiarata non come `int` (anche se gli indirizzi sembrano essere interi), ma come puntatore a intero `int *`. Lo stesso vale per tutti gli altri tipi.

Questo discorso vale anche per gli altri tipi definiti in C. Per esempio, l'indirizzo di una variabile `float` va messo in una variabile di tipo `float *`, mentre l'indirizzo di una variabile `char` va messo in una variabile `char *`. Si noti che, in ogni caso, l'indirizzo di una variabile è comunque un numero intero (anche se la variabile contiene un valore reale!) Il programma seguente altri.c contiene delle variabili puntatore a tipi diversi da `int`

```
/*
  Le variabili float* contengono indirizzi di variabili
float. Lo stesso per char*
*/

int main() {
  float f;
```

```
  float *p;
  char c;
  char *t;

  p=&f;
  c=&t;

  return 0;
}
```

Dichiarazioni di variabili di tipo puntatore

1. quando si dichiarano più variabili puntatori insieme, `*` va ripetuto su tutte
2. tranne quando si usa `typedef`

Rispetto alle variabili di tutti gli altri tipi, le dichiarazioni di variabili di tipo puntatore hanno delle regole diverse. Infatti, mentre per tutti gli altri tipi si può fare:

```
Tipo a, b, c;
```

e questo dichiara tre variabili di tipo `Tipo`, nel caso dei puntatori il risultato è differente. La dichiarazione:

```
int * a, b, c;
```

Ha come risultato che la sola variabile `a` è un puntatore a intero, mentre `b` e `c` sono variabili intere. Per dichiarare tutte e tre le variabili come puntatori, occorre fare:

```
int *a, *b, *c;
```

ossia, si deve mettere l'asterisco `*` davanti a ognuna delle variabili. La regola per la dichiarazione dei puntatori è semplicemente che un asterisco davanti a una variabile dichiara la variabile (e solo quella) come un puntatore. Quindi, se si fa:

```
float *x, y, *z;
```

le variabili `x` e `z` sono puntatori a `float`, dato che sono precedute da un asterisco. La variabile `y` invece, non avendo `*` davanti, non è un puntatore ma una variabile di tipo `float`. Eventuali spazi fra il nome del tipo, l'asterisco, la variabile e le virgole sono ignorati.

Una cosa importante da notare è che nel caso in cui si definisce un tipo puntatore usando la `typedef`, in nuovo tipo segue invece la regola generale. In altre parole, se si fa:

```
typedef int * punt;

punt a, b, c;
```

Allora il tipo `punt` è equivalente al tipo `int *`. L'unica differenza è che la dichiarazione delle variabili `a`, `b` e `c` le definisce tutte e tre del tipo `punt`, ossia sono tutte e tre puntatori a interi.

Valore e indirizzo

1. tutte le variabili hanno valore e indirizzo
2. questo vale anche per i puntatori
3. per i puntatori, sono entrambi indirizzi di memoria
4. sono comunque due cose diverse

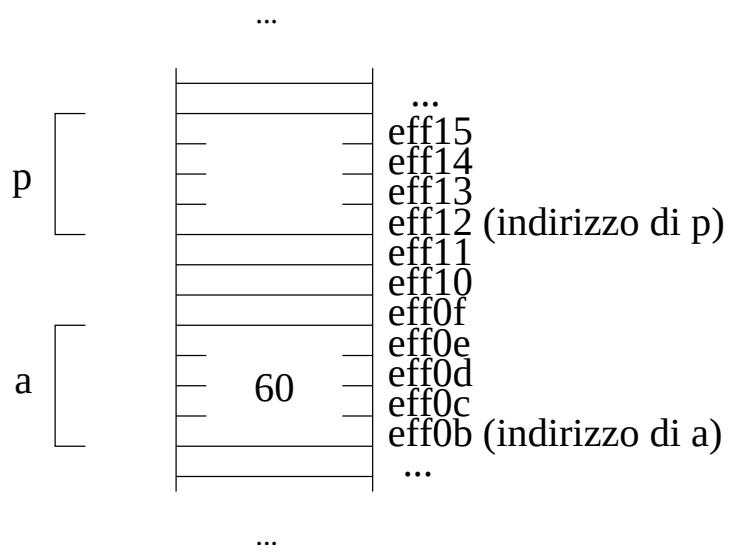
Tutte le variabili hanno un valore e un indirizzo. Questa regola vale anche per le variabili puntatore: anche loro hanno un indirizzo (la zona di memoria in sui si trovano) e un valore (quello che è memorizzato). Quello che le rende diverse è il fatto che il valore in esso memorizzato è a sua volta un indirizzo.

In altre parole, le variabili puntatore sono sempre zone di memoria in cui viene memorizzato un numero. Mentre le variabili intere e reali contengono valori interi e reali, il valore memorizzato in una variabile puntatore è un numero che indica una posizione di memoria.

Anche se il valore di una variabile puntatore e il suo indirizzo sono entrambe posizioni di memoria, sono comunque due cose diverse: l'indirizzo è la posizione iniziale della memoria occupata, mentre il valore è quello che si trova memorizzato, che a sua volta un indirizzo di memoria (in generale, uno diverso). Il seguente programma valind.c illustra la differenza.

```
/*
  Valore e indirizzo sono diversi.
*/

int main() {
  int a=60;
  int *p;

  p=&a;

  printf("&a=%x a=%x\n", &a, a);
  printf("&p=%x p=%x\n", &p, p);

  return 0;
}
```

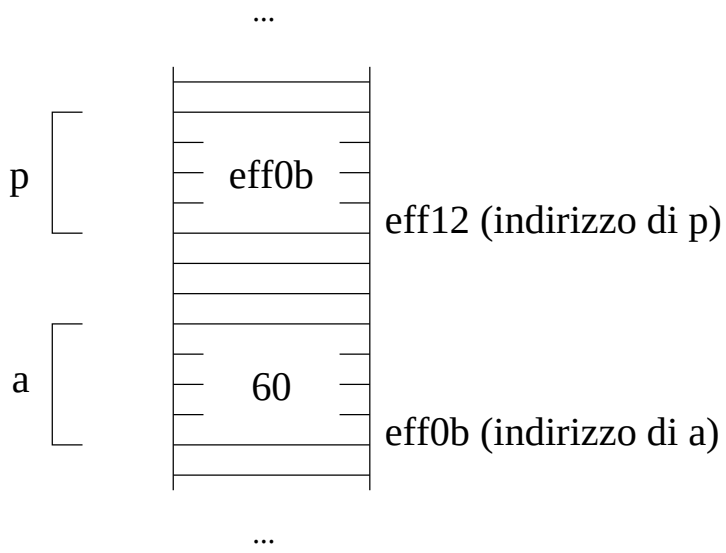
Questo programma stampa indirizzo e valore per le due variabili `a` e `p`. Tutte le variabili hanno un indirizzo (la posizione in memoria) e un valore (quello che c'è memorizzato). Questo vale quindi sia per `a`, che è una variabile intera, che per `p`, che è un puntatore.



Quando si esegue il programma, viene creata una zona di memoria per ognuna delle variabili. In questo caso, viene creata una zona per `a` ed una zona per `p`.

Nella figura a sinistra si vede un possibile stato della memoria dopo che le variabili sono state create. Quando si assegna ad `a` il valore `60`, questo valore viene scritto nella sua posizione di memoria.

Quando si esegue l'istruzione `p=&a`, l'indirizzo di `a` viene scritto in `p`. Quando si fanno queste figure di esempio, è bene scrivere dei valori di esempio per le posizioni di memoria. In questa figura, per esempio, la posizione iniziale di `a` è `eff0b`. Si tratta ovviamente di un valore di esempio: la posizione effettiva della variabile viene determinata soltanto quando si esegue il programma. In ogni caso, il numero che dice la posizione iniziale di `a` viene memorizzato in `p`.



Quello che verrebbe stampato, in questo caso, è:

```
&a=eff0b a=3c
&p=eff12 p=eff0b
```

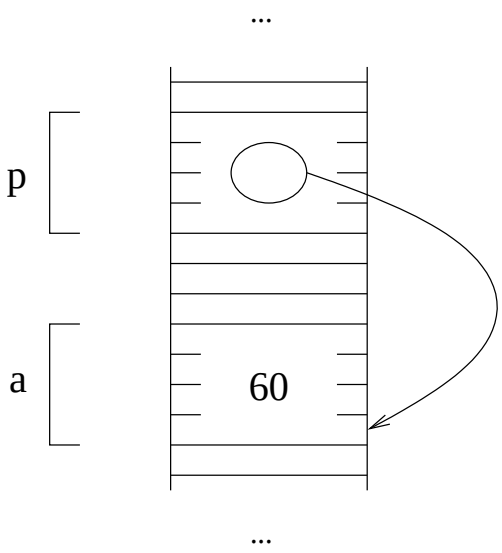
Il primo valore delle due righe è l'indirizzo delle due variabili. I due indirizzi sono necessariamente diversi, dal momento che ogni variabile occupa una sua zona di memoria diversa da quelle delle altre.

Il secondo numero stampato è il valore delle due variabili. Il valore memorizzato in `a`, nel nostro caso, è `60`, che viene stampato in esadecimale. Nella seconda riga, il secondo numero stampato è il valore memorizzato nella variabile `p`. Dato che in `p` abbiamo messo l'indirizzo di `a` (quando abbiamo eseguito l'istruzione `p=&a`) viene stampato appunto l'indirizzo di `a`.

La variabile `p`, come tutte le variabili, ha sia un indirizzo che un valore. Soltanto che, a differenza di `a`, il valore di `p` è a sua volta un indirizzo (in questo caso, quello di `a`). Il fatto che il valore di una variabile puntatore sia un indirizzo può a volte causare confusione. Basta però disegnare lo stato della memoria per chiarire la situazione.

Facciamo ora una precisazione sulla rappresentazione grafica della memoria. Nella figura di sopra abbiamo messo dei valori numerici di esempio per rappresentare gli indirizzi delle variabili (`eff0b` per l'indirizzo di `a` e `eff12` per l'indirizzo di `p`). Va tenuto sempre presente che questi sono solo indirizzi di esempio: se si esegue il programma una seconda volta, potrebbero venire stampati degli indirizzi diversi, per esempio `ffb02` ed `ffb12`. Nel seguito, si mettono comunque dei valori specifici per gli indirizzi, ma va tenuto presente che si tratta solo di valori di esempio.

La figura qui accanto mostra una rappresentazione grafica diversa, che non ha questo problema. Per indicare che un puntatore contiene un certo indirizzo, mettiamo una freccia che termina con la locazione a quell'indirizzo. Per esempio, l'istruzione `p=&` mette in `p` l'indirizzo di `a`. Questo si può rappresentare come una freccia che parte dalle locazioni occupate da `p` e punta alla prima locazione occupata da `a`. Questa rappresentazione grafica permette di far capire quale è l'indirizzo memorizzato in una variabile puntatore senza dover scrivere valori numerici di esempio per gli indirizzi.



Questa rappresentazione risulta particolarmente comoda per visualizzare lo stato della memoria quando si eseguono programmi che contengono molti puntatori.

Usare l'oggetto puntato

1. se `p` è un puntatore, allora `*p` è come se fosse una variabile
2. esempio di uso dell'operatore `*`
3. cosa succede in memoria quando si esegue un programma con puntatori
4. rappresentazione degli indirizzi con freccie

Come si è visto, l'operatore `&` trova l'indirizzo di una variabile. In C esiste anche l'operatore inverso, che permette di accedere alla zona di memoria definita da un puntatore. Questo operatore è l'asterisco `*`. Il simbolo viene quindi usato in due modi distinti: per definire un tipo puntatore a un tipo (esempio: `int *p`), e per definire l'oggetto associato a un indirizzo, di cui ora parliamo. La regola generale è:

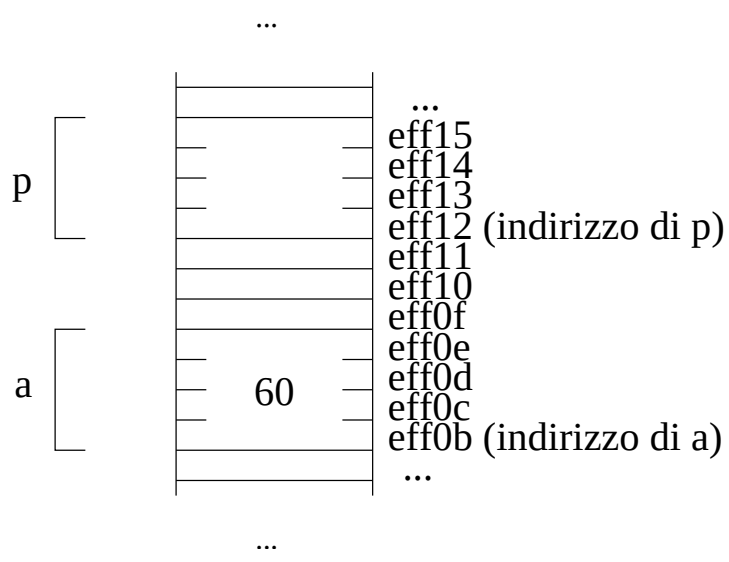
> se `p` è una variabile di tipo puntatore a intero, si può pensare a `*p` come a una variabile di tipo intero.

È quindi possibile per esempio stampare il valore di `*p`, oppure usare questo valore all'interno di espressioni come fosse un intero (per esempio, `12+*p-2` è una espressione perfettamente valida). È anche possibile memorizzare dei valori in questa variabile: per esempio, `*p=34;` è una istruzione valida.

Nel seguente programma varpunt.c, la variabile `a` e la espressione `*p` sono esattamente equivalenti. Le loro zone di memoria sono le stesse, e quindi usare/cambiare il loro valore genera esattamente gli stessi risultati.

```
/*
  Se si assegna a p l'indirizzo di a,
allora *p ed a sono la stessa cosa.
*/

int main() {
  int a=60;
  int *p;       /* qui * indica che p e' un puntatore e non un intero */

  p=&a;         /* l'indirizzo di a va in p */

                /* modifiche ad a sono modifiche a *p */
  a=12;
  printf("a=%d *p=%d\n", a, *p);
                /* qui ----^ l'operatore * indica che la variabile non
                e' p ma la zona di memoria indicata da p */


                /* modifiche a *p sono modifiche ad a */
  *p=24312;
  printf("a=%d *p=%d\n", a, *p);

  return 0;
}
```

Quando si usano i puntatori, è molto facile fare confusione fra oggetti puntati e i loro puntatori. Un puntatore è un indirizzo di memoria, mentre l'oggetto puntato è la zona di memoria che inizia con l'indirizzo, ed è grande quanto basta per contenere il tipo corrispondente. D'altra parte, le variabili di tipo puntatore sono anche esse variabili, ossia zone di memoria. La differenza fra una variabile `int` e una variabile di tipo puntatore a intero è che la prima contiene un valore intero, mentre la seconda contiene un indirizzo, e in particolare l'indirizzo iniziale della zona di memoria associata a un intero.



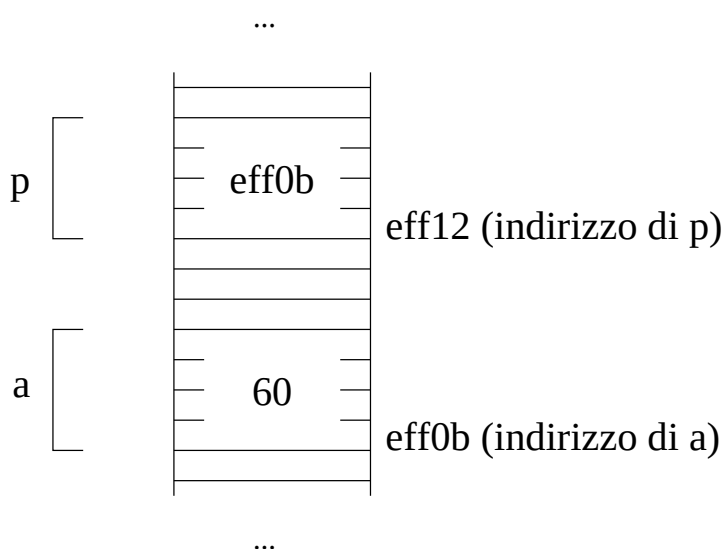
Consideriamo la rappresentazione della memoria come array, e vediamo cosa succede se si esegue il programma di sopra. In questa figura, e nelle successive, rappresentiamo solo la parte della memoria che ci interessa. Nel vettore sono stati messi dei valori numerici per gli indici. Questi valori sono ovviamente dei numeri di esempio: a ogni esecuzione, la posizione in memoria di una variabile può cambiare.

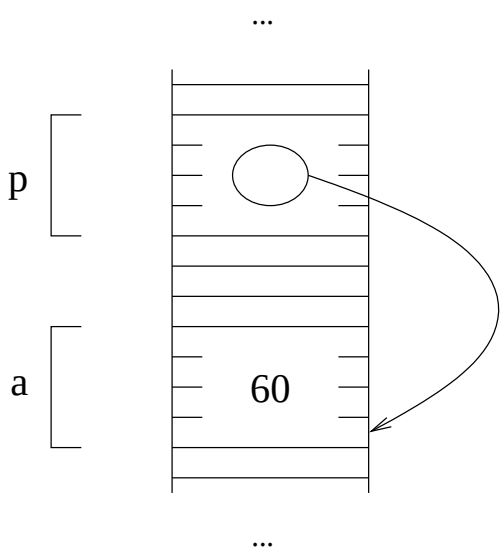
In questo esempio, e nei successivi, si fa l'ipotesi che i puntatori siano rappresentabili con quattro byte. Questo viene fatto solo ai fini dell'esempio: non esiste nessuna garanzia che lo spazio occupato da un puntatore abbia questo valore (in generale, fare delle assunzioni sullo spazio occupato dalle variabili di un certo tipo è un errore di programmazione).

Nella dichiarazione di `a` viene anche assegnato il valore `60`, che viene quindi memorizzato nella corrispondente area di memoria. Non viene invece dato nessun valore iniziale a `p`, per cui il suo valore non è determinato.

La assegnazione dell'indirizzo di a a p fa sí che nella variabile p venga messa la prima posizione di memoria occupata da a. Nel caso dell'esempio, in p si mette il valore ef00b.

Nella figura a lato si vede lo stato della memoria dopo aver eseguito la istruzione p=&a. L'espressione *p rappresenta la zona di memoria che comincia con l'indirizzo memorizzato in p. In questo caso, la variabile p contiene il valore eff0b, quindi *p rappresenta la zona di memoria che comincia con eff0b. Inoltre, p è una variabile di puntatore a intero, quindi *p rappresenta un intero, che è grande quattro byte. Si può pensare che *p è una variabile intera (grande quindi quattro byte) la cui zona di memoria inizia da eff0b.





Nella figura a lato si vede lo stesso stato della memoria usando la rappresentazione con freccie. Il risultato finale è ovviamente lo stesso: dato che la freccia indica che la variabile p contiene l'indirizzo dia, quando si modifica/stampa *p viene modificato/stampato il valore memorizzato in a.

Nella rappresentazione con freccie, *p indica quindi la zona di memoria in cui termina la freccia che inizia in p.

Tutto questo ragionamento permette di concludere che, ovunque si usa *p, questo è equivalente ad a, nel programma di sopra. Va notato che, se a un certo punto si cambia il valore di p, questo non funziona più. Per esempio, se si fa p=&b dove b è un'altra variabile intera, allora *p diventa equivalente a b.

Tipo dei puntatori

1. perchè i puntatori non sono interi?
2. perchè altrimenti non so in che modo `*p` va interpretato (è un intero? un carattere?)
3. alternativa possibile ma non usata: `*` con due argomenti

Nelle pagine precedenti abbiamo visto come i puntatori siano effettivamente dei numeri. Si distinguono dai numeri interi solo per il fatto che il loro valore indica l'indirizzo di una locazione, ossia indica una posizione della memoria.

Ci si potrebbe a questo punto chiedere perchè

1. i tipi puntatori sono di tipo diverso dagli interi?
2. i puntatori non hanno tutti lo stesso tipo?

In effetti, il tipo di un puntatore è una cosa che può venire facilmente scavalcata: per esempio, un programma in cui `x` è un intero e `p` è un puntatore può contenere la istruzione `x=p;` oppure `p=x;`. Il compilatore produce un messaggio di avvertimento (warning), ma la compilazione ha successo. Lo stesso vale per la distinzione fra puntatori a due tipi diversi: se `p` è un puntatore a intero e `t` è un puntatore a reale, allora le istruzioni `p=t` e `t=p` sono ammesse, anche se si genera un warning in compilazione. Il programma tipipunt.c contiene alcuni esempi di assegnazioni fra tipi diversi.

```
/*
  Assegnazioni fra tipi sbagliati.
*/

int main() {
  int x;
  int *p;
  float *t;


  x=p;          /* assegnazione a variabile int * di valore int */

  t=x;          /* assegnazione a variabile float * di valore int */

  p=t;          /* assegnazione a variabile int * di valore float *
                */

  return 0;
}
```

Si noti anche che i messaggi di errore possono facilmente venire eliminati con il cast.

A questo punto, ci si chiede come mai i puntatori non sono semplicemente di tipo `int`. Il motivo è che in questo modo, non sarebbe chiaro quale è il tipo della espressione `*p`. Infatti, se `p` è semplicemente un intero, allora `*p` potrebbe essere a sua volta un intero, oppure un carattere, oppure un reale, ecc.

In altre parole, se i puntatori fossero interi, allora le istruzioni `p=&c`, `p=&x` e `p=&f` sarebbero tutte accettabili anche se `p` fosse intero, ma `c`, `x` e `f` sono un carattere, un intero e un reale. A questo punto non sarebbe più possibile scrivere semplicemente `*p`, perchè `p` potrebbe puntare a un carattere, a un intero, oppure a un reale. Scrivendo `*p` non sarebbe possibile capire quale è il tipo di questa espressione, e quindi non si capirebbe nemmeno quale è la zona di memoria a cui il puntatore si riferisce (se fosse un puntatore a carattere, allora sarebbe una zona grande 1, se fosse un puntatore a intero la zona sarebbe grande 4, ecc).

Proviamo a immaginare una variante del C in cui i puntatori siano semplicemente di tipo intero. Consideriamo il seguente codice.

```
/*
  Quello che segue non e' codice C corretto.
*/

int main() {
  int px;
  float g;

  px=925434;

  g=*px / 2;

  return 0;
}
```

La espressione `*px / 2` dice che devo prendere l'oggetto puntato da `px` e dividere per due questo valore. Però non dice quale è il tipo di `*px`. Questa espressione potrebbe quindi venire interpretata in (almeno) tre modi:

1. prendi il byte che sta all'inidirizzo scritto in `px`, e dividilo per due;
2. prendi i quattro byte a partire dall'indirizzo che sta scritto in `px`, e interpreta questi quattro byte come un intero; dividilo per due usando la divisione fra interi;
3. prendi gli stessi quattro byte, e interpretali come un numero `float`; dividi questo numero usando la divisione fra reali.

Il risultato sarebbe diverso: nel primo caso si prende il primo byte invece dei primi quattro; il secondo e il terzo caso sono diversi perchè numeri reali e numeri interi, anche se occupano lo stesso spazio, sono rappresentati in modo diverso (in più, il risultato è anche diverso perchè sono diverse le regole della divisione fra interi e fra reali).

Tutto questo serve a dire:

> il puntatore `p` ha un tipo, altrimenti sarebbe impossibile capire il tipo di `*p`, e quindi capire quanto è grande e come va interpretata la zona di memoria puntata da `p`.

Una possibile alternativa alla soluzione dei tipi sui puntatori sarebbe stata quella di definire l'operatore di oggetto puntato `*` come un operatore binario, il cui primo argomento è il puntatore e il secondo è il tipo dell'oggetto puntato. In altre parole, i puntatori non avrebbero bisogno dei tipi se, per prendere l'oggetto puntato da una variabile `p`, dovessi fare `*(p,int)` se voglio un intero, `*(p,float)` se voglio un float, ecc. Non è però questa la soluzione seguita in C.

Copia del valore e copia dell'indirizzo

1. differenza fra `b=a` e `p=&a`, in pratica
2. `b=a` significa: copia il valore di `a` in `b`; non ci sono più relazioni fra `a` e `b` dopo la copia
3. `p=&a` copia l'indirizzo: anche se è una copia, ora `*p` e `a` sono la stessa cosa
4. cosa succede in memoria

Il seguente programma cambia.c fa capire la differenza fra assegnare il valore di una variabile (`b=a`) e assegnare il suo indirizzo (`p=&a`).

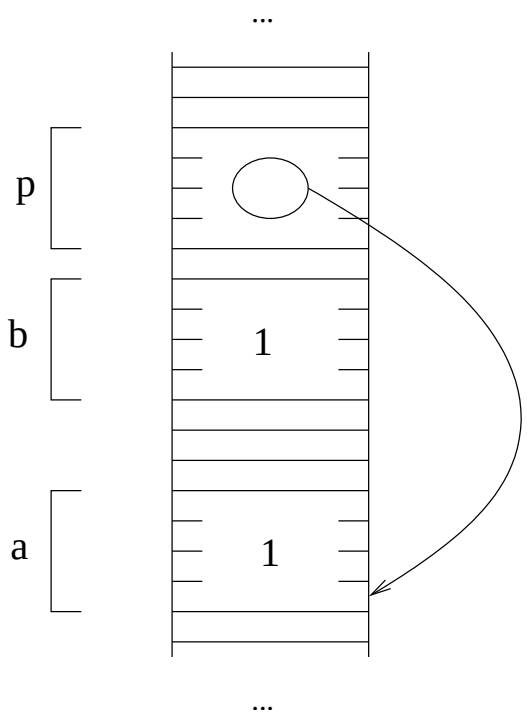
```
/*
  Effetti del cambiamento di una variabile.
*/

int main(void) {
  int a;
  int b;
  int *p;

  a=1;                  /* a vale 1 */

  b=a;                  /* il valore di a viene copiato in b */

  p=&a;                 /* l'indirizzo di a viene messo in p */

  a=12;                 /* a viene cambiato */

                        /* quanto valgono b e *p a questo punto? */
  printf("*p vale %d\n", *p);
  printf("b vale %d\n", b);

  return 0;
}
```

Assegnare a `b` il valore di `a` significa semplicemente che il valore di `a` viene copiato nella zona di memoria di `b`. Da questo momento in poi, non esiste nessuna associazione fra `a` e `b`. In particolare, se si cambia il valore di `a`, questo non si riflette sul valore di `b`. È come fotocopiare un documento e poi fare delle modifiche solo sull'originale: chiaramente, la copia non viene modificata automaticamente.

Nel caso della assegnazione dell'indirizzo, quando si fa `p=&a`, da questo momento in poi, `*p` e `a` sono realmente la stessa zona di memoria. Quindi, da questo momento in poi, ogni modifica su `a` cambia anche `*p` e viceversa. Nell'esempio della documento fotocopiato, è come se usassimo due nomi diversi per indicare lo stesso documento: dato che il documento è sempre lo stesso, quando modifico uno dei due sto modificando anche l'altro. Questa associazione permane fino a che il *valore* di `p` non viene cambiato.

Attenzione! La seguente affermazione è sbagliata:

> affermazione errata: dopo aver fatto `a=12` il valore di `*p` viene aggiornato di conseguenza in modo automatico



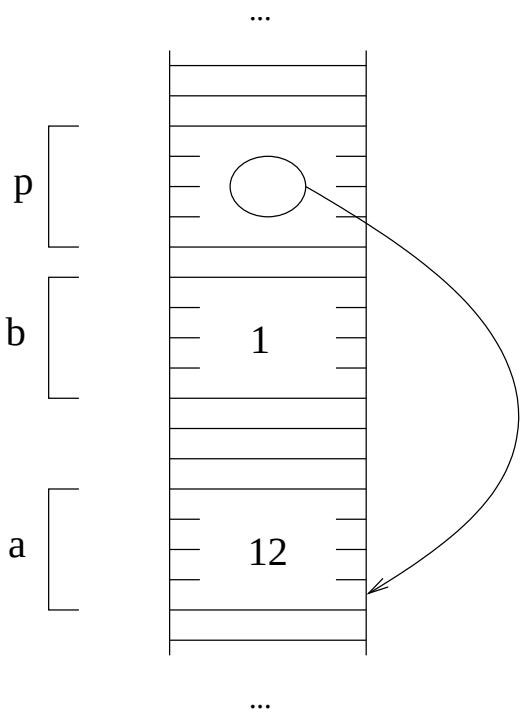
Infatti, questa affermazione presuppone che `a` e `*p` siano due variabili memorizzate in due zone di memoria diverse, e che poi è il calcolatore che si preoccupa di far sí che i loro valori siano sincronizzati. Questo non è assolutamente vero: quando si memorizza `12` in `a`, questo valore non viene copiato in `*p`. Quello che invece succede è che `*p` ha la stessa zona di memoria di `a` e quindi il suo valore non può che essere lo stesso.

La figura qui accanto mostra lo stato della memoria subito dopo la assegnazione `p=&a`. In questo momento, i valori di `a` e `b` coincidono semplicemente perchè il valore di `a` è stato copiato in `b` e non è stato più modificato; invece, il valore di `*p` e di `a` coincidono perchè sono esattamente la stessa area di memoria.

Quando si fa la modifica `a=12` del valore di `a`, la parte di memoria modificata è solo quella in cui è memorizzato `a`. La situazione diventa quindi come nella figura qui a fianco.

È chiaro che il valore di `b` non è stato alterato, visto che è memorizzato in una zona di memoria differente. D'altra parte, il valore di `p` è l'indirizzo di `a`, e quindi `*p` e `a` stanno nella stessa zona di memoria. Quindi, il valore di `*p` è quello memorizzato in questa zona, e quindi `12`.

In conclusione, possiamo dire che la differenza fra assegnare il valore di una variabile come in `b=a` e assegnare il suo indirizzo come in `p=&a` è che, nel primo caso, una volta copiato il valore le due variabili vanno ognuna per la sua strada, nel secondo caso l'elemento puntato `*p` coincide con la variabile, e quindi cambia insieme alla variabile.



Puntatori indefiniti

1. i puntatori sono inizialmente indefiniti, come tutte la variabili
2. cosa succede quando si usa un puntatore indefinito
3. non si devono usare puntatori indefiniti

Negli esempi visti fino ad ora, i puntatori venivano usati soltanto per memorizzare indirizzi di variabili. In generale, un puntatore è una variabile che contiene un indirizzo di memoria.

Quando si definisce una variabile di tipo puntatore, per esempio con `int *p;`, si crea una variabile il cui contenuto è un indirizzo di memoria. Se non si assegna un valore a questa variabile, il suo contenuto è indefinito, quindi non è possibile sapere a priori quale indirizzo è scritto in `p`. Usare il valore di `*p`, oppure memorizzare un valore in `*p` produce un risultato indefinito, ossia non è possibile sapere a priori cosa succede (il risultato può cambiare di volta in volta). Questo avviene perchè il valore iniziale (indefinito) di `p` può essere l'indirizzo di una qualsiasi zona di memoria, che può essere o no associata a un'altra variabile del programma, e può anche essere una posizione di memoria a cui il programma non può accedere.

Per usare una variabile puntatore, è necessario che contenga l'indirizzo di una zona di memoria su cui siamo sicuri che:

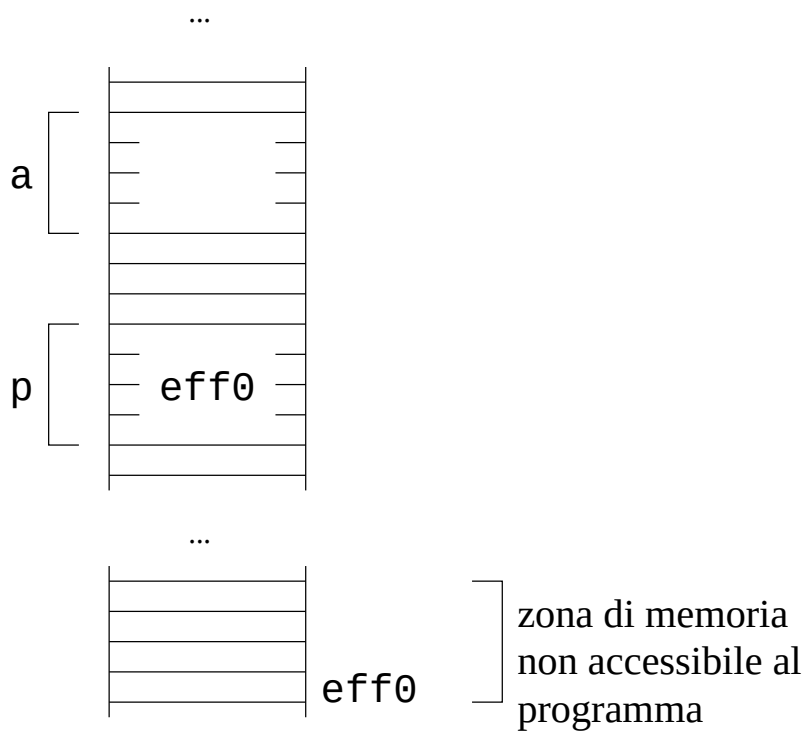
1. il sistema operativo ci permette di accedere;
2. non viene modificata dal programma in modo inaspettato.

Consideriamo il programma nomalloc.c riportato qui sotto, che non segue queste regole.

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

  a=3;

  *p=12;

  printf("%d\n", a);

  return 0;
}
```

Il programma non rispetta la regola sul valore di un puntatore, che dovrebbe essere l'indirizzo di una zona di memoria in cui siamo certi di poter accedere senza problemi. Al contrario, il valore di `p` è indeterminato come tutte le variabili che non sono state inizializzate.

Consideriamo ora cosa può succedere quando si esegue questo programma.



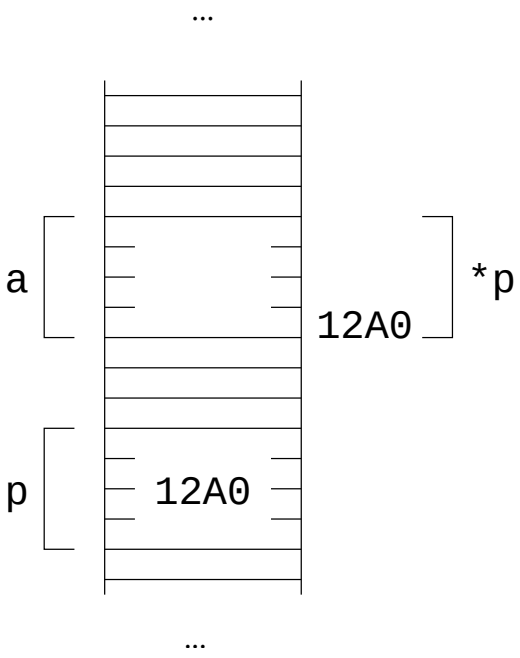
La figura qui accanto mostra un possibile caso in cui il valore iniziale di `p` (su cui, come si è detto, non si può avere alcun controllo) è l'indirizzo di una zona di memoria a cui il programma non può accedere.

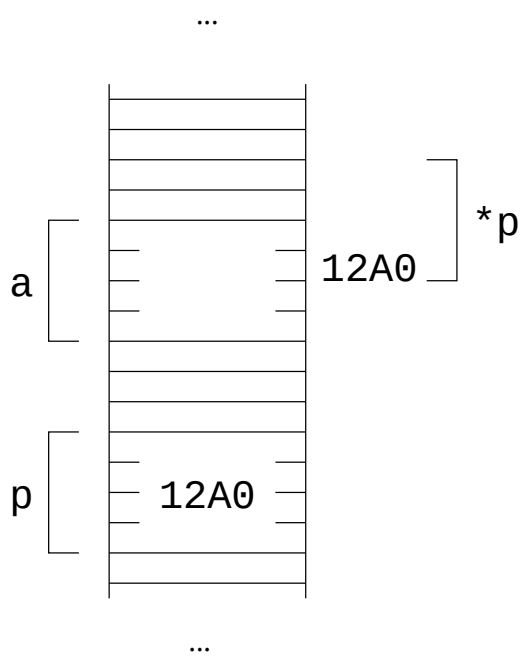
Quando si esegue l'istruzione `*p=12`, viene fatto un tentativo di mettere il valore `12` nella zona di memoria il cui indirizzo si trova in `p`. A seconda del tipo di sistema operativo che si sta usando, può venire generato un errore in esecuzione oppure si può produrre un blocco dell'intero sistema operativo (nel qual caso, il computer va riavviato).

La figura qui accanto mostra un altro possibile caso: quello in cui il valore contenuto in `p` casualmente risulta essere l'indirizzo di un'altra variabile del programma. Si tratta chiaramente di un caso molto poco probabile.

Quello che succede è che `*p=12` mette il valore `12` nella zona di memoria della variabile `a`. Quando poi si va a stampare `a`, viene quindi stampato il valore `12`.

Dal momento che la variabile `a` conteneva il valore `3`, e il programma non esegue operazioni di assegnamento su `a`, ci si aspetterebbe la stampa del valore `3`. La stampa di `12` è quindi un risultato non atteso.





Questo caso è simile al precedente, soltanto che la zona di memoria il cui indirizzo iniziale si trova in `p` si sovrappone soltanto parzialmente con la zona di memoria in cui è memorizzato `a`. In questo caso, viene stampato un valore che dipende da come i numeri sono rappresentati in memoria, ma tipicamente non è nè `12` è `3`.

Da notare che tutte e tre le situazioni di sopra sono possibili, e non è nemmeno possibile prevedere a priori quale delle tre si verifica in una certa esecuzione. Il problema è quindi chiaro: i programmi che si scrivono devono comportarsi sempre nello stesso modo, e non fare ogni volta una cosa diversa.

Allocazione di memoria

1. quali zone di memoria si possono usare (riservate ad uso esclusivo del programma)
2. funzione `malloc`: dà una zona di memoria con queste caratteristiche
3. esempio di creazione di memoria per contenere un intero
4. vale ancora la regola che `*p` è una variabile intera, ma si può usare solo se questa zona di memoria è stata riservata

Un puntatore è una variabile che contiene un indirizzo di memoria. Finora, abbiamo usato i puntatori solo per memorizzare in essi le posizioni di altre variabili. In realtà, è possibile memorizzare in un puntatore un valore qualsiasi. D'altra parte, quando si usa `*p` dove `p` è un puntatore, occorre essere certi che la zona di memoria puntata da `p` si possa utilizzare, ossia:

1. il sistema operativo ci permette di accedere a questa zona;
2. il contenuto di questa zona non viene modificata dal programma in modo inaspettato.

L'unico modo per realizzare queste due condizioni visto fino ad ora è stato quello di memorizzare in `p` l'indirizzo di una variabile con istruzioni del tipo di `p=&a`. In questo caso, infatti, si è sicuri che la zona di memoria è accessibile al programma, dato che è la zona di memoria di una sua variabile; inoltre, non può venire modificata dal programma, a meno che non si facciano delle modifiche su `a`.

Esiste però un altro meccanismo che soddisfa questi due requisiti, ed è quello di chiedere al sistema operativo di riservare una nuova zona di memoria. In altre parole, si chiama una funzione che crea una nuova zona di memoria e ne restituisce l'indirizzo iniziale. Questa funzione garantisce che:

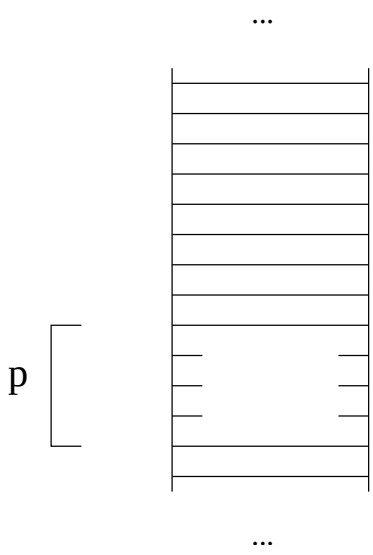
1. la zona di memoria è accessibile al programma;
2. se si creano nuove variabili (per esempio con una chiamata di funzione), non saranno memorizzate in questa zone;
3. ogni nuova chiamata alla funzione crea una zona di memoria nuova, distinta dalle precedenti.

Il primo punto garantisce che il programma può effettivamente usare queste locazioni di memoria. Il secondo e il terzo punto dicono che non ci saranno altre variabili o altri puntatori che sono associati automaticamente alla stessa zona di memoria. Questo serve a garantire che la zona di memoria non verrà cambiata inaspettatamente dalla modifica di una variabile che non avevamo messo in relazione con quella zona.

La funzione che crea una nuova zona di memoria si chiama `malloc`. Ha un argomento, che è il numero di byte che la zona da allocare deve contenere. Per esempio `malloc(10)` assegna al programma una nuova zona di memoria grande 10 byte. Il valore di ritorno di questa funzione è l'indirizzo iniziale di questa zona. Quindi, `malloc(10)`, oltre a creare una zona di 10 byte, restituisce l'indirizzo del primo di questi byte, cioè l'indirizzo più basso in questa zona.

Il programma alloca.c crea una zona di memoria larga abbastanza da contenere un intero. L'indirizzo di ritorno viene assegnato alla variabile di tipo puntatore a intero `p`.

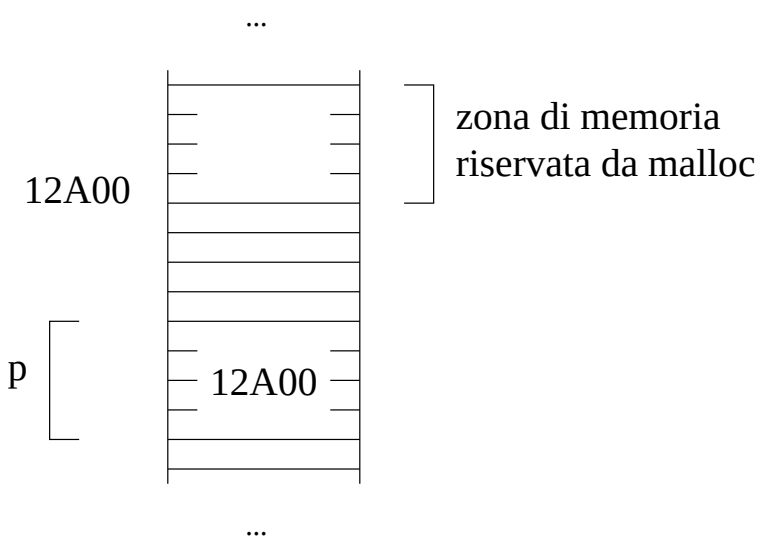
```
/*
  Allocazione di memoria.
*/

#include<stdlib.h>

int main(void) {
  int *p;

  p=malloc(sizeof(int));

  *p=12;

  (*p)++;

  printf("*p vale %d\n", *p);

  return 0;
}
```



Qui accanto vediamo lo stato iniziale della memoria: la variabile puntatore `p` non contiene nessun valore significativo, e nel programma non sono definite altre variabili.

La prima istruzione del programma è la chiamata alla funzione `malloc`. In particolare, l'argomento è `sizeof(int)`, quindi la istruzione `p=malloc(sizeof(int))` crea una zona di memoria abbastanza grande da contenere un intero, e l'indirizzo iniziale di questa zona viene restituito e memorizzato nella variabile puntatore `p`.

Nella figura si vede lo stato della memoria dopo questa chiamata: la funzione `malloc` ha riservato una zona di memoria di quattro byte (ipotizziamo come sempre che un intero è rappresentato con quattro byte) per il programma; la variabile `p` contiene l'indirizzo della prima locazione di questa zona.



Possiamo ora usare la notazione freccia per indicare quale è l'indirizzo memorizzato nella variabile `p`. Si ricordi che la notazione con la freccia è semplicemente un trucco grafico per rendere più chiare la figure che riguardano i puntatori, ma che quello che in effetti si trova scritto nelle variabili puntatore è un indirizzo, ossia un numero, e che la variabili puntatore, per il resto, si comportano come tutte le variabili normali.

Un'osservazione importante: la variabile puntatore `p` contiene l'indirizzo di una zona di memoria grande quanto basta per contenere un intero. Inoltre, la variabile `p` è di tipo puntatore a intero. Quindi, si può ancora pensare a `*p` come alla variabile intera la cui zona di memoria è quella identificata dalla punta della freccia del disegno di sopra.

Il altre parole, dopo aver fatto `p=malloc(sizeof(int))`, è come se avessimo creato una nuova variabile di tipo intero `*p`, che possiamo quindi usare come una qualsiasi altra variabile intera: possiamo per esempio assegnare ad essa un valore con `*p=12`, incrementarla con `(*p)++`, e stampare il suo valore con `printf("*p vale %d\n", *p);` (il motivo delle parentesi per `(*p)++` sarà chiarito più avanti).

Il punto fondamentale da ricordare è che è possibile creare una zona di memoria sia dichiarando una variabile che chiamando la funzione `malloc`. In quest'ultimo caso, il risultato della funzione va memorizzato in un puntatore.

Riassumendo, la regola generale è: se `p` è una variabile puntatore a intero, allora `*p` è come se fosse una variabile intera. Questa variabile si può però usare soltanto se in `p` c'è un indirizzo di memoria di una zona che si può usare. Questo avviene soltanto se:

- la zona di memoria è stata riservata con `malloc`, oppure
- la zona di memoria corrisponde a una variabile.

Il seguente programma duemalloc.c mostra due possibili modi di mettere in una variabile puntatore degli indirizzi di zone di memoria che si possono usare.

```
/*
  Allocazione di memoria.
*/

#include<stdlib.h>

int main(void) {
  int *p, *q;
  int a;
```

```
  p=malloc(sizeof(int));
  *p=12;
  (*p)=(*p)+12;

  q=&a;
  *q=4;
  (*q)=(*q)+4;

  printf("Valori di *p e *q: %d e %d\n", *p, *q);

  return 0;
}
```

L’indirizzo che viene memorizzato in `p` è il valore di ritorno di `malloc`, quindi l’indirizzo di una zona di memoria che è riservata dalla funzione `malloc`. Nella variabile `q` viene invece memorizzato l’indirizzo della variabile `a`. Si noti che questa operazione fa sí che le modifiche a `*q` siano anche modifiche ad `a`, dato che la zona di memoria è la stessa.

Rispetto all’esempio di prima nomalloc.c, in cui la variabile puntatore è indeterminata, qui la modifica ad `a` è prevedibile, ed avviene tutte le volte (la cosa da evitare è quella in cui il comportamento di un programma dipenda da valori iniziali non determinati).

il valore `NULL`

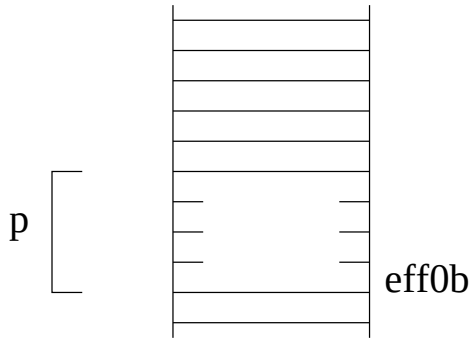
Zone di memoria inaccessibili

Si consideri il seguente programma inacc.c. Cosa viene stampato?

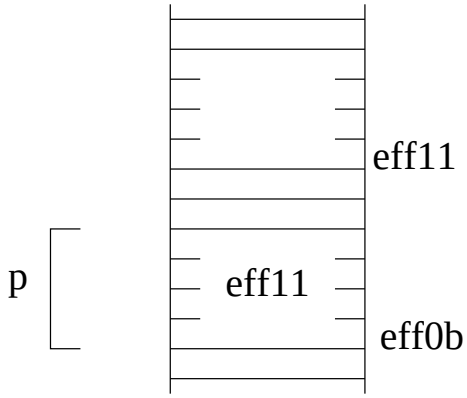
```
#include<stdlib.h>

int main() {
  int *p;

  p=malloc(sizeof(int));
  *p=12;

  p=malloc(sizeof(int));

  printf("%d\n", *p);

  return 0;
}
```

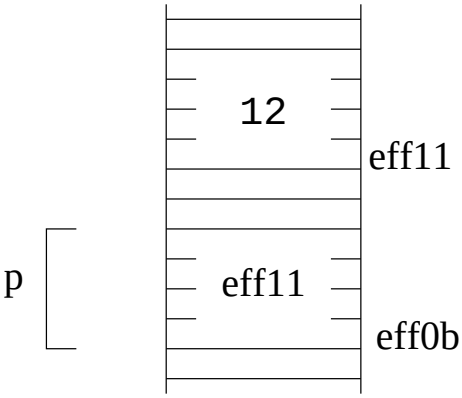
Per trovare la soluzione a problemi di questo genere, occorre disegnare l’evoluzione dello stato della memoria quando si esegue il programma. Si usano ovviamente dei valori di esempio per le posizioni di memoria, dal momento che i valori esatti vengono determinati solo quando si esegue il programma.

La fingura qui accanto mostra lo stato della memoria dopo la creazione della variabile: l'unica zona di memoria attualmente riservata è quella che corrisponde a `p`. Il valore in essa contenuta è tuttora indefinito.



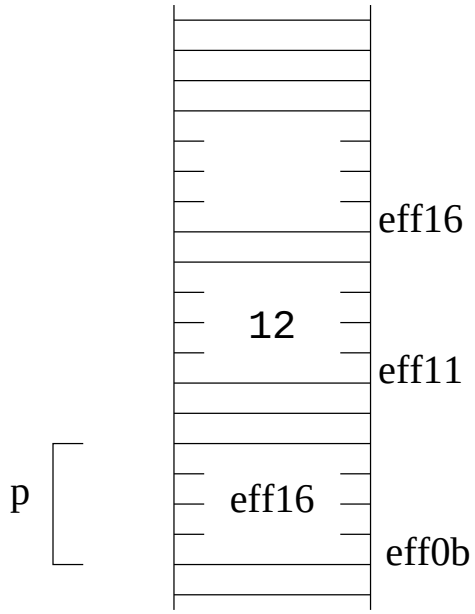
Dopo aver effettuato l'operazione di allocazione, viene creata una zona di memoria, e il suo indirizzo viene memorizzato in `p`. Lo stato della memoria è quindi come quello qui a sinistra. L'istruzione `*p=12` mette il valore `12` nella zona di memoria che era stata allocata. Lo stato della memoria dopo questa operazione è riportato qui a destra.





A questo punto viene fatta la seconda `malloc`. La regola della allocazione di memoria dice che la zona di memoria che viene restituita è una zona che il programma può usare, e che non è già usata. Quindi, la zona di memoria riservata da `malloc` non può essere nè la zona di memoria di una variabile nè una zona che è stata già allocata in precedenza.

La situazione che si viene a creare è quindi che abbiamo una *seconda* zona di memoria riservata, il cui indirizzo iniziale viene scritto in `p`. In altre parole, `p` contiene l'indirizzo della zona di memoria che è stata allocata con la seconda chiamata alla funzione `malloc`.



Cosa viene stampato? L'istruzione `printf("%d\n", *p)` stampa il contenuto della zona di memoria che è stata allocata dalla seconda chiamata alla funzione `malloc`. D'altra parte, nessun valore è stato memorizzato in questa zona. Il valore che viene stampato dipende quindi dai valori iniziali della memoria, ossia è indefinito.

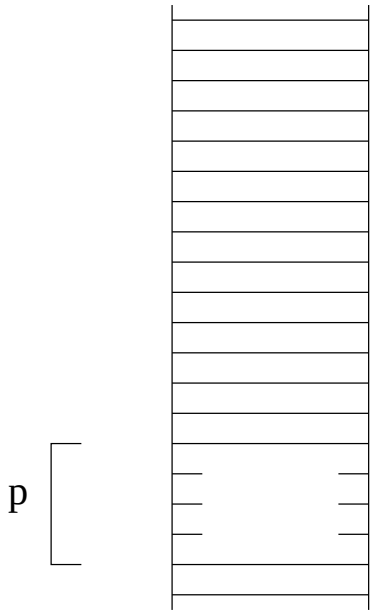
Quello che vale per le variabili, ossia che il loro valore è indefinito finchè un valore non viene memorizzato in esso, continua a valere.

Un'altra osservazione importante che si puè fare su questo programma è che non è più possibile accedere alla prima zona di memoria, quella creata dalla prima chiamata di `malloc`. Infatti, l'unico modo per accedere a una zona di memoria è quello di conoscere il suo indirizzo, ma l'indirizzo di questa zona era stato memorizzato in `p` e poi sovrascritto. Il programma non può quindi sapere quale è l'indirizzo di questa zona, e non può quindi accedere ad essa.

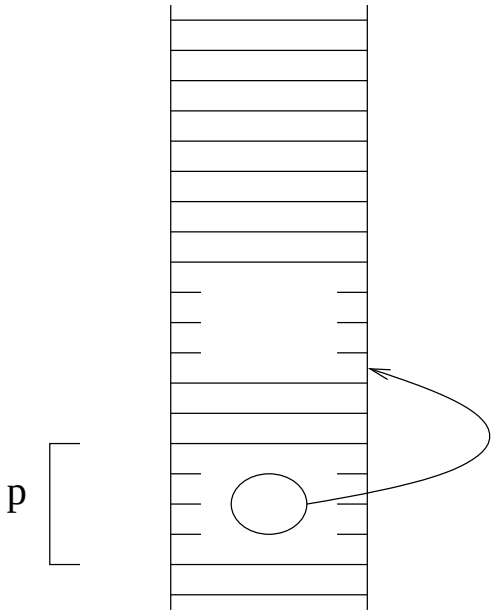
Si noti che è possibile accedere a una zona di memoria scrivendo nel programma `p=0xeff11`: l'indirizzo `eff11` è infatti soltanto un valore di esempio: il valore di ritorno effettivo della prima chiamata a `malloc` viene determinato soltanto in esecuzione, e più esecuzioni dello stesso programma in genere producono risultati diversi.

Tutto questo dice che, se il valore di ritorno della chiamata a `malloc` viene perso, non c'è più modo per accedere alla zona di memoria che era stata riservata.

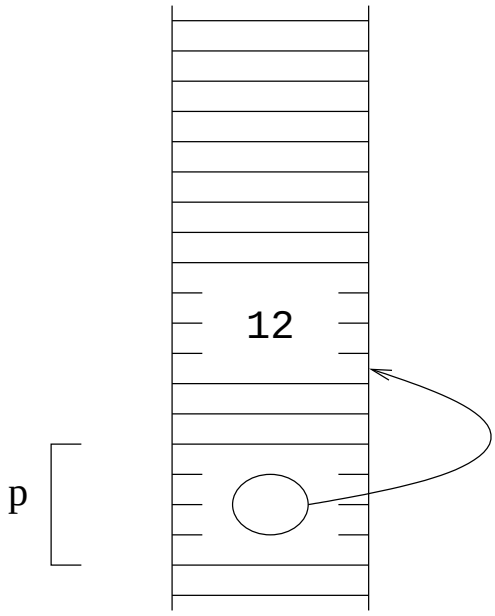
Nella rappresentazione con freccie, l'evoluzione della memoria è la seguente.



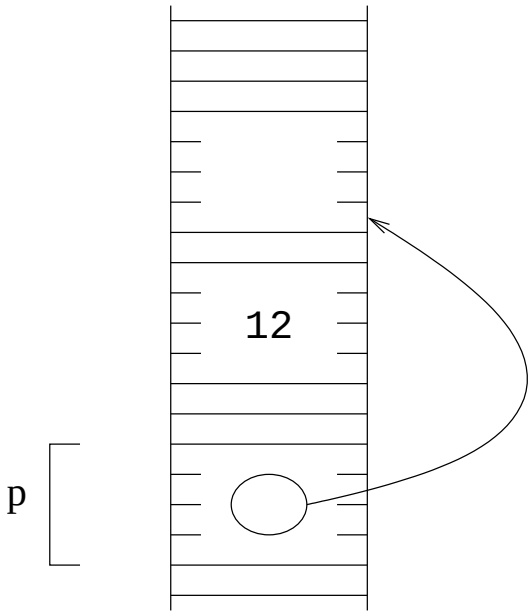
Creazione della variabile `p`



Dopo la prima `malloc`



Dopo l'istruzione `*p=12`



Dopo la seconda `malloc`

Passaggio di un puntatore a una funzione

1. si può passare l'indirizzo di una variabile a una funzione
2. effetti della modifica a) del valore della zona di memoria puntata, e b) del valore del parametro formale stesso
3. cosa succede in memoria

La prima applicazione dei puntatori è quello del passaggio dei parametri a una funzione. Quando si passa un numero a una funzione, quello che succede in effetti è che:

1. si crea una nuova variabile con il nome del parametro formale;
2. in questa nuova variabile si copia il valore del parametro attuale;
3. si esegue il corpo della funzione.

Questo significa che, se si chiama una funzione passando una variabile come parametro, quello che viene in effetti usato dalla funzione è il valore della variabile, che viene copiato in una zona di memoria nuova (il parametro formale). Se quindi si modifica il parametro formale, queste modifiche vengono fatte nella nuova zona di memoria, distinta da quella del parametro attuale. Quindi, il programma che ha chiamato la funzione non vede le modifiche, semplicemente perchè le modifiche non sono state fatta in una zona di memoria che corrisponde a una sua variabile.

Vediamo ora cosa succede se si passa un puntatore ad una funzione. Consideriamo il seguente programma passaggio.c, in cui la funzione `esempio` prende come parametri un intero e un puntatore a intero.

```
/*
  Una funzione che riceve un puntatore e un intero.
*/

#define NULL 0

void esempio(int a, int *b) {
  a=12;

  *b=3;

  b=NULL;
}

int main() {
  int x;
  int y;
  int *z;

  x=10;
  y=43;
  z=&y;

  printf("Valori: x=%d y=%d z=%x\n", x, y, z);

  esempio(x, z);

  printf("Valori: x=%d y=%d z=%x\n", x, y, z);

  return 0;
}
```

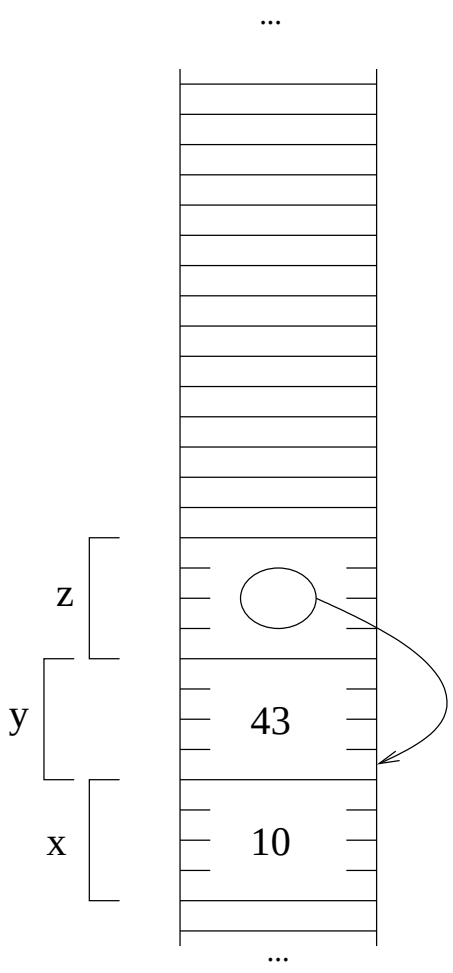
La funzione `esempio` fa tre cose: cambia il valore della variabile intera `a=12;`, cambia il valore dell'oggetto puntato dal puntatore `*b=3;`, e cambia anche il valore del puntatore stesso `b=NULL`.

Compilando ed eseguendo il programma si ottiene la seguente stampa:

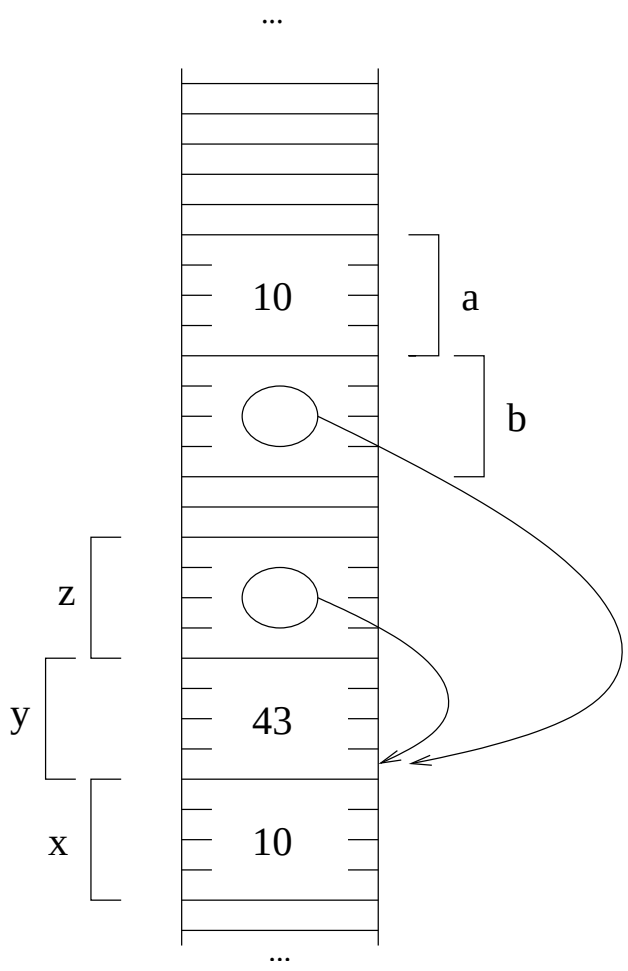
```
Valori: x=10 y=43 z=bffff448
Valori: x=10 y=3 z=bffff448
```

Questo significa che i valori di x e z non sono cambiati, mentre quello di y sí. Per capire questo comportamento, vediamo lo stato della memoria passo per passo.

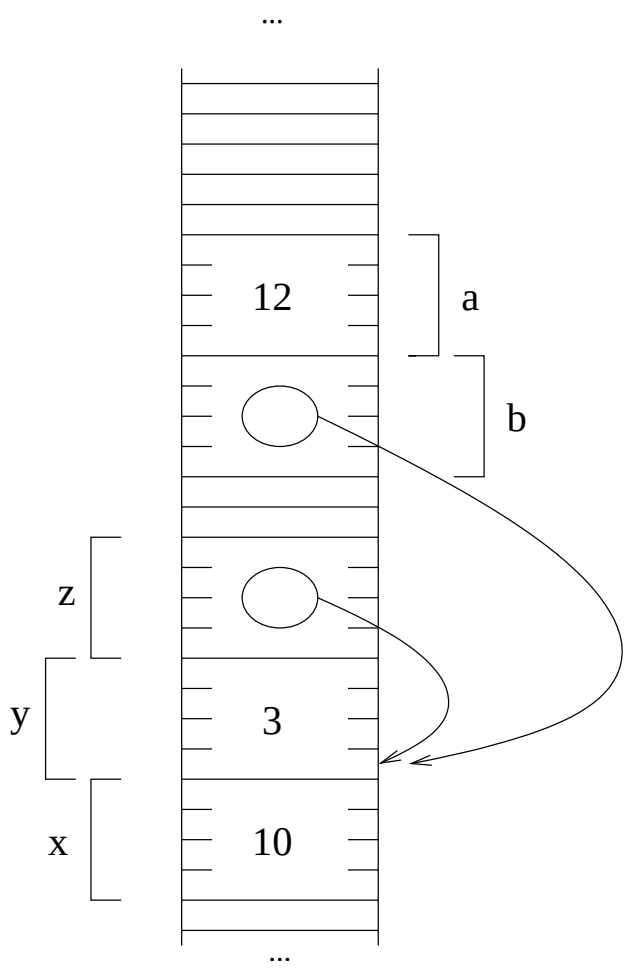
Stato della memoria prima della chiamata a funzione: x, y e z sono tre variabili, ognuna associata a una zona di memoria. Il valore di z è l'indirizzo di y. In altre parole, nella variabile z è memorizzata la posizione di memoria in cui y inizia.



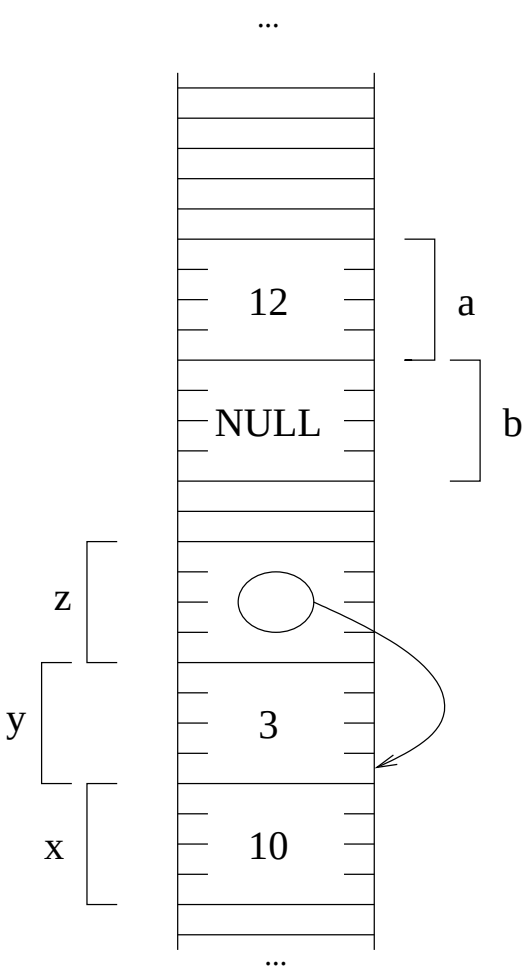
Stato della memoria all'inizio della esecuzione della funzione. Sono state create due nuove variabili `a` e `b`, in cui sono stati copiati i valori degli argomenti con cui è stata chiamata la funzione. In questo caso, la funzione è stata chiamata con `esempio(x,z)`. Quindi, nella variabile `a` è stato messo il valore di `x`, cioè `10`. Nella variabile `b` è stato messo il valore di `z`. Si noti che il valore di `z` non è `43`; il valore di `z` è l'indirizzo a cui è memorizzata la variabile `y`. Questo valore viene copiato in `b`, che quindi ora contiene l'indirizzo della variabile `y`.



Durante la esecuzione della funzione, viene messo il valore `12` nella variabile `a`. La seconda istruzione mette `3` nella zona di memoria puntata da `b`. Dal momento che `b` punta alla zona di memoria in cui è memorizzato `y`, la situazione alla fine delle prime due istruzioni è quella riportata qui accanto.



L'ultima istruzione della funzione mette `NULL` nel puntatore `b`. Alla fine della esecuzione della funzione la situazione è quindi quella accanto. A questo punto, la funzione termina e le variabili `a` e `b` vengono deallocate.



Vediamo ora cosa è successo alle variabili del programma principale. Le variabili `x` e `z` non sono state cambiate. L'unica variabile che ha subito un cambiamento è `y`.

La spiegazione è ora chiara: le variabili `x` e `z` non sono state cambiate semplicemente perchè i loro valori sono stati copiati in `a` e `b`, e i cambiamenti sono stati fatti sulle zone di memoria che corrispondono alle nuove variabili `a` e `b`. Il valore di `y` è invece diverso. Questo avviene perchè la funzione `esempio` conosce l'indirizzo in cui è memorizzata, perchè è quello che sta scritto in `b` (mentre per esempio di `x` conosce solo il valore).

Se l'indirizzo di una variabile è noto, è possibile cambiare il suo valore semplicemente con una istruzione `*indirizzo=nuovo_valore;`. Questo è esattamente quello che si è fatto con la istruzione `*b=3`: si mette il valore 3 nella memoria puntata da `b`. Dato che `b` contiene l'indirizzo di `y`, questa istruzione mette 3 nella variabile `y`.

Si noti che l'istruzione `*b=3` non altera il valore di `b`, ma solo quello della memoria puntata da `b`. Le modifiche che vengono poi fatte a `b` (ossia `b=NULL;`) sono modifiche a una variabile locale di `esempio`, per cui non vengono viste dal programma principale.

Volendo riassumere, diamo la regola generale sul passaggio dei parameteri:

1. quando si passa una variabile, il suo valore viene copiato, per cui le modifiche fatte all'interno della procedura non hanno effetto sulle variabili del programma principale;
2. quando si passa un indirizzo, la funzione è in grado di modificare il contenuto della zona di memoria che parte da quell'indirizzo; le modifiche sono quindi modifiche a zone di memoria in cui ci sono variabili del programma principale.

Volendo essere ancora più pratici, la regola è che se si passa una variabile normalmente allora le modifiche fatte nella funzione non hanno effetto; se si passa un indirizzo con `&variabile` e poi si usa `*` nella funzione, le modifiche sono viste dal programma principale. Questa regola si ottiene semplicemente applicando la regola generale di creazione di nuove variabili nelle funzione e il significato dei puntatori. È sempre bene però ricordare il perchè le cose funzionano in questo modo, altrimenti situazioni più complesse (per esempio, puntatori a puntatori) potrebbero risultare incomprensibili.

Puntatori a puntatori

1. i puntatori sono variabili come tutte le altre
2. quindi, si può determinare il loro indirizzo
3. differenza fra indirizzo e valore

Abbiamo già visto come l'indirizzo di una variabile sia un numero, che è quindi possible memorizzare in una variabile. Si è anche visto come il tipo di un indirizzo dipende dal tipo dell'oggetto puntato. Una variabile di tipo puntatore è anche essa una variabile, per cui è una zone di memoria, per cui si può trovare il suo indirizzo usando l'operatore `&`.

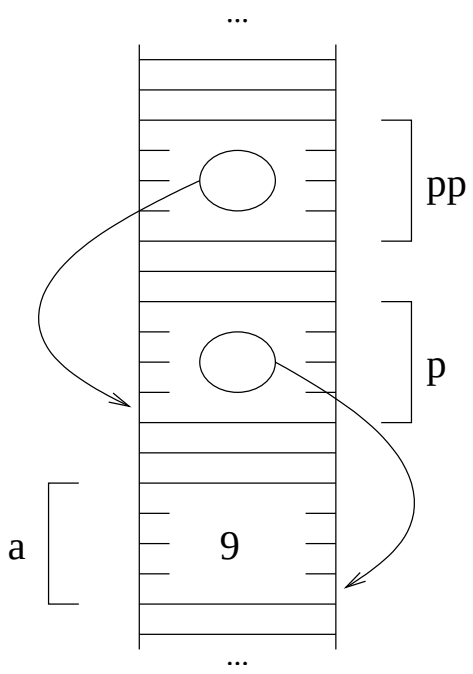
Per essere precisi, il puntatore è una variabile con un tipo, per cui il suo indirizzo è (seguendo la regola generale) un puntatore a un puntatore. Per memorizzare per esempio l'indirizzo di un puntatore a intero in una variabile, questa deve essere si tipo puntatore a puntatore a intero. Questo tipo si scrive `int **`. Il seguente programma puntpunt.c memorizza in una variabile `p` l'indirizzo della variabile intera `a`, e poi trova l'indirizzo di `p` e lo memorizza nella variabile `pp`.

```
/*
  Un esempio di puntatore a puntatore.
*/

int main() {
  int a;
  int *p;
  int **pp;

  a=9;

  p=&a;

  pp=&p;

  printf("Indirizzo di pp=%x, valore=%x\n", &pp, pp);

  printf("Indirizzo di  p=%x, valore=%x\n", &p, p);

  printf("Indirizzo di  a=%x, valore=%x\n", &a, a);

  return 0;
}
```

Quello che si ottiene eseguendo il programma è una stampa del genere:

```
Indirizzo di pp=bffff444, valore=bffff448
Indirizzo di  p=bffff448, valore=bffff44c
Indirizzo di  a=bffff44c, valore=9
```

A parte il valore di `a`, tutti gli altri sono indirizzi, per cui il loro valore non è noto a priori. In un'altra esecuzione del programma si potrebbero ottenere numeri diversi. Quello che però vale sempre è che il valore della variabile `pp` coincide con l'indirizzo della variabile `p`, e che il valore di `p` coincide con l'indirizzo di `a`.



La figura qui accanto mostra una rappresentazione della memoria: `p` contiene l'indirizzo di `a`, e a sua volta `pp` contiene l'indirizzo di `p`.

Dato il valore di `pp` è chiaro che è possibile accedere al valore di `a`: basta seguire i puntatori, ossia prima trovare il valore di `*pp`, che è l'indirizzo di `a`, e questo permette di trovare il valore di `a` usando ancora l'operatore `*`. Quindi, dato `pp`, il valore di `a` si può trovare con `**pp`.

In questo modo si può anche assgnare un valore alla variabile `a` usando `pp`: basta usare una istruzione del tipo `**p = ...` .