Puntatori e Heap

1.Cosa Sono i Puntatori ?

I puntatori sono fondamentalmente delle variabili, come quelle intere, reali e carattere. Tuttavia, l'unica differenza consiste nel fatto che essi non contengono una valore numerico od alfanumerico, ma un puntatore (ossia un indirizzo) alla locazione di memoria dove è memorizzato un certo valore.

Come viene definito un puntatore nel linguaggio C ? E' molto semplice. La definizione di una variabile puntatore avviene nello stesso modo con cui si definisce una variabile, eccetto che dobbiamo aggiungere un'asterisco tra il tipobase e il nome della variabile puntatore. Il simbolo asterisco permette di definire una variabile puntatore e specificare il tipo di dato che è contenuto nell'area di memoria "puntata" dalla variabile puntatore. In altri termini, quando si definisce una variabile puntatore bisogna fin dall'inizio specificare che tipo di dato (ad esempio intero, reale o carattere) verrà contenuto nell'area di memoria il cui indirizzo è contenuto nella variabile puntatore.

Per esempio, il codice seguente crea due puntatori, entrambi puntano ad un intero:

```
int *pNumberOne, *pNumberTwo;
```

Adesso facciamo puntare questi puntatori a qualcosa:

```
int NumberOne, NumberTwo;

pNumberOne = &NumberOne;
pNumberTwo = &NumberTwo;
```

L'operatore & (ampersand o e-commerciale) deve essere letto come "l'indirizzo di" (Address-of) e restituisce l'indirizzo di memoria della variabile e non la variabile. Nell'esempio precedente *pNumberOne* è assegnata con l'indirizzo della variabile *NumberOne*, in altre parole punta a tale variabile.

Ora, se desideriamo riferirci all'indirizzo *di NumberOne*, possiamo usare *pNumberOne*. Se desideriamo riferirci al valore di *NumberOne* da *pNumberOne*, dovremmo scrivere **pNumberOne*.

L'operatore * deve essere letto come "il contenuto della locazione di memoria puntata da". L'operatore * prende il nome di *operatore di dereference*.

1.1.Che cosa abbiamo imparato finora, un esempio

Vediamo un primo esempio per mettere in pratica quanto fino finora.

```
#include <stdio.h>

/* definisco le variabili: */
int nNumber;
int *pPointer;

int main(void)
{

      /* ora, diamogli un valore:*/
      nNumber = 15;
      pPointer = &nNumber;

      /* stampiamo il valore di nNumber: */
      printf( "\n   nNumber e' uguale a :  %d  ",nNumber);

      /* ora, alteriamo nNumber per mezzo di pPointer: */
      *pPointer = 25;

      /* proviamo che nNumber è stato cambiato e stampiamolo nuovamente:*/
      printf("\n   nNumber e' uguale a:  %d \n  ",nNumber);
}
```

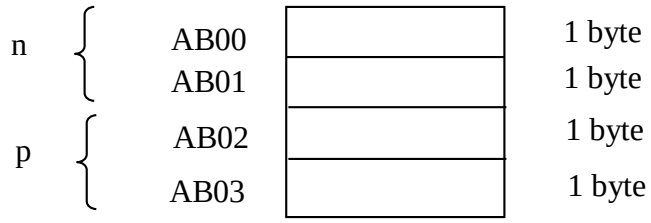
Leggi, scrivi in un editor e compila il precedente codice d'esempio; assicurati di aver capito perché funziona.

1.2.Cosa succede nella memoria ?

Consideriamo il seguente esempio:

```
int n=4;
int *p;

int main(void)
{
      p=&n;
      *p=3;
}
```

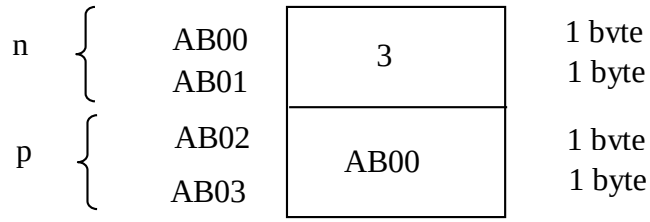
Cerhiamo di capire cosa succede nella memoria del computer. Come verrà detto più avanti, la memoria che andremo ad esplorare è l'Area Dati della memoria assegnata al programma utente.
Le definizioni int n=4 e int *p, determinano la presenza di due locazioni di memoria allocate nell'area dati, come mostrato nella seguente figura.



Come si vede, è stato ipotizzato che entrambe le variabili occupino 2 bytes; in realtà nella maggior parte degli ambienti di compilazione in ANSI C, gli *int* occupano 4 bytes e i puntatori sono indirizzi anch'essi di 4 byte. Solo per semplicità, nel seguito si supporrà che la lunghezza di una variabile puntatore sia di 2 bytes. Nella figura sono anche indicati gli indirizzi (in esadecimale) di ciascun byte. Si ricordi che l'indirizzo della variabile *n* è per definizione l'indirizzo del primo byte, cioè AB00. Analogamente l'indirizzo di *p* è AB02. Si ricordi ancora che tali indirizzi vengono fissati sia in fase di compilazione (viene fissato un indirizzo relativo a partire da un indirizzo base o di impianto) sia in quella di esecuzione (dove viene deciso l'indirizzo di impianto). Le istruzioni:

```
p=&n;
*p=3;
```

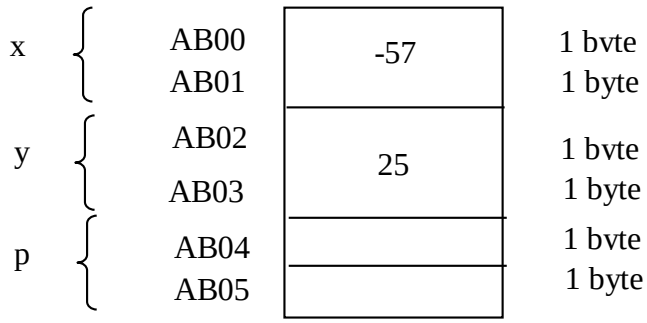
hanno il seguente effetto in memoria:



Si consideri il seguente esempio:

```
int x=-57, y=25;
int *p;

int main(void)
{
      p=&y;
      x=*p;
}
```

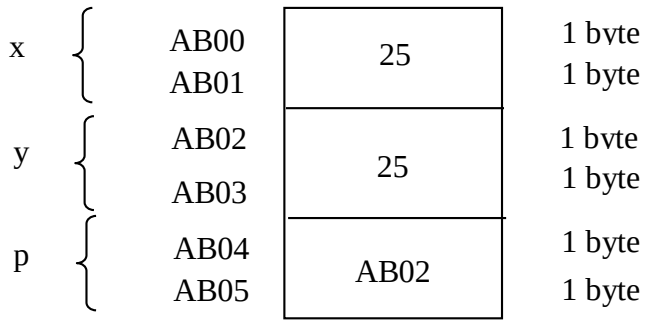
Cerhiamo di capire cosa succede nella memoria del computer. Le definizioni delle variabili x,y,p, determinano la presenza di tre locazioni di memoria allocate nell'area dati, come mostrato nella seguente figura.



Le istruzioni:

```
p=&y;
x=*p;
```

hanno il seguente effetto in memoria:



2.Allocazione Dinamica

L'allocazione dinamica è un concetto chiave. E' usata per allocare la memoria, ossia riservarla per contenere determinati valori, ed assegnarne l'indirizzo di partenza ad un puntatore. Il seguente codice dimostra come allocare la memoria destinata a contenere un numero intero:

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

int *pNumber;

int main(void)
{
      pNumber = (int *) malloc(sizeof(int));
}
```

La prima linea dichiara il puntatore *pNumber*. Nel *main*, l'unica istruzione presente, alloca la memoria per un intero ed assegna a *pNumber* tale indirizzo (fa puntare *pNumbe*r alla memoria appena allocata).

Nel seguito è proposto un altro esempio, questa volta utilizzando un char:

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

char *pString;

int main(void)
{
      pString = (char *) malloc(sizeof(char));
}
```

La formula è la stessa, l'unica differenza è il tipobase.

In entrambi gli esempi, è stato fatto uso della funzione di libreria *malloc()*. Essa è definita nel file *stdlib.h*, che dovrà dunque essere sempre incluso nel programma. La funzione *malloc()*, richiede come parametro attuale, la dimensione dello spazio di memoria che deve essere allocato; negli esempi mostrati, tale dimensione è fornita tramite il comando *sizeof(tipobase)*, che fornisce il numero di byte occupato dal tipobase. La funzione *malloc()* restituisce un generico puntatore all'area di memoria allocata; nel caso in cui non riesca ad allocare memoria, la funzione ritorna *NULL*. Come si vede dagli esempi, il valore del puntatore ritornato dalla funzione *malloc()* deve essere sottoposto ad una funzione di cast, convertendolo nel tipo del puntatore che si desidera avere (puntatore ad *int* e puntatore a *char* nei due esempi).

La caratteristica fondamentale dell'allocazione dinamica, consiste nel fatto che la memoria allocata tramite la funzione *malloc()* viene rilasciata (resa disponibile per altri usi) solo quando essa viene espressamente rilasciata tramite l'uso di una opportuna funzione di libreria, *free()*, o a quando l'intero programma utente non termina. Quindi, se consideriamo l'esempio successivo:

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

int *pPointer;

int main(void)
{
      pPointer = (int *)malloc(sizeof(int));
      *pPointer = 25;
      printf("\nValore di *pPointer = %d ", *pPointer);
}
```

la memoria allocata rimane intatta fino a quando il programma non termina.

Quindi la memoria allocata con il comando *malloc* è lasciata intatta, non viene mai cancellata automaticamente, ossia la memoria rimarrà occupata per tutta l'esecuzione del programma. In questo modo, se la memoria che abbiamo allocato non ci serve più, si sprecherà lo spazio che altre parti del nostro programma potrebbero usare.

Una soluzione a ciò è il rilascio della memoria precedentemente allocata tramite la funzione *free()*. L'uso è semplicissimo, basti utilizzare la funzione *free()*, come segue:

```
free(pPointer);
```

ossia specificando la variabile puntatore relativa all'area di memoria precedentemente allocata. Si noti che la funzione *free()* dealloca l'area puntata dal puntatore, ma non annulla il puntatore, ossia esso non viene posto a NULL, ma viene lasciato al valore attuale.

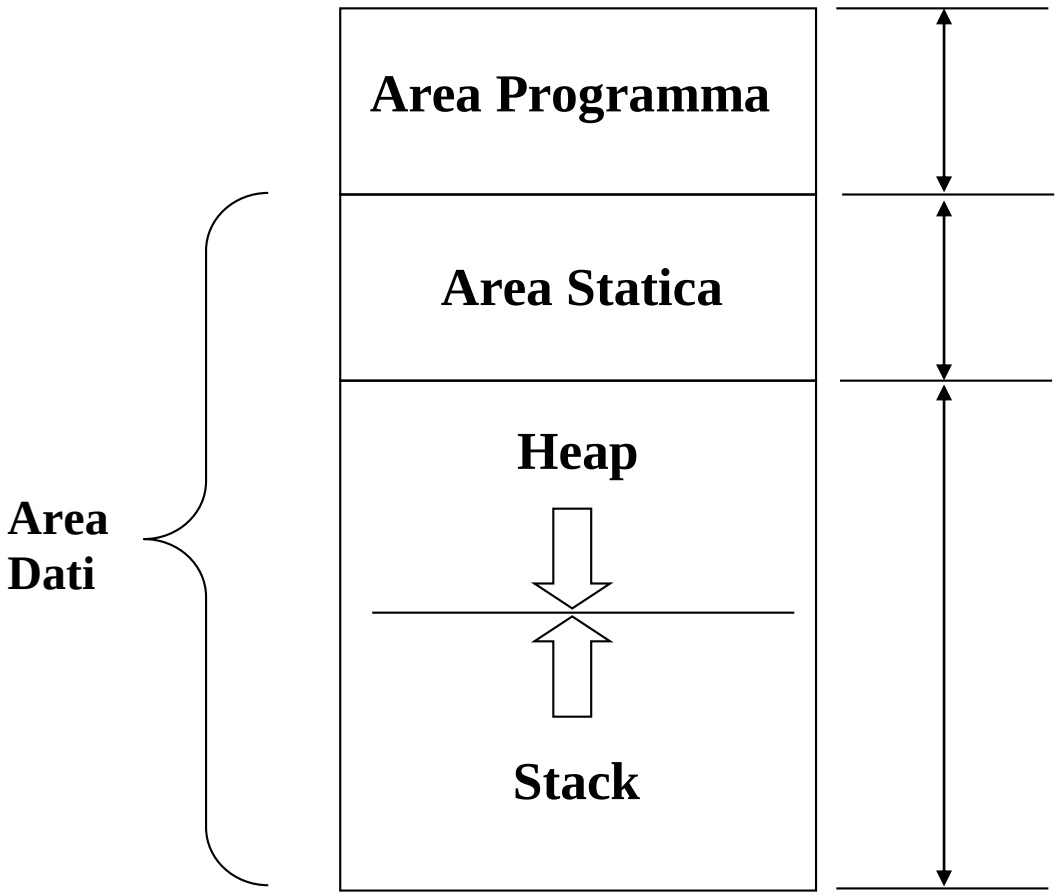
Come mostrato negli esempi precedenti, le funzioni *malloc()* e *free()* sono definite in *stdlib.h*, che bisogna dunque includere sempre nei programmi.

2.1.Dove viene allocata la Memoria ?

Cerchiamo di capire adesso quale è lo spazio di memoria riservato per l'allocazione/rilascio di memoria dinamica. Per capirlo basta pensare che a ciascun programma utente viene assegnata una porzione di memoria, che è automaticamente divisa in quattro porzioni:

- area del codice contiene il codice del programma
- area statica che contiene le variabili statiche
- area heap disponibile per allocazioni dinamiche
- area stack contiene i record di attivazione delle funzioni (variabili locali e parametri). Per capire meglio il contenuto dell'area stack si veda la dispensa "Record di Attivazione" relativa al Passaggio dei Parametri.

Delle quattro aree sopramenzionate, è chiaro che l'area heap viene utilizzata per l'allocazione di memoria dinamica. Si consideri la seguente figura:



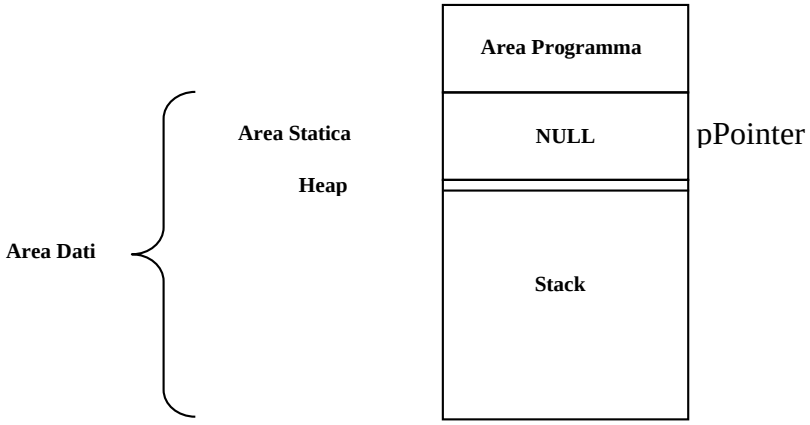
Come è visibile dalla figura, le dimensioni delle aree programma e dati sono fisse e sono decise in fase di compilazione. Infatti, tali dimensioni corrispondono al numero di righe comando del programma e al numero di variabili statiche definite in esso. Le altre due aree (Heap e Stack) non hanno dimensione fissa, ma l'area complessiva (pari alla somma delle due) è anch'essa fissa e decisa in fase di esecuzione.

Il fatto che l'Heap e lo Stack non abbiano dimensione fissa significa che tale dimensione varia nel tempo a seconda dell'utilizzo delle due aree. In particolare per quanto riguarda l'area Heap, essa crescerà quando verranno allocate variabili dinamiche. La sua dimensione si ridurrà, invece, quando una o più variabili dinamiche verranno deallocate.

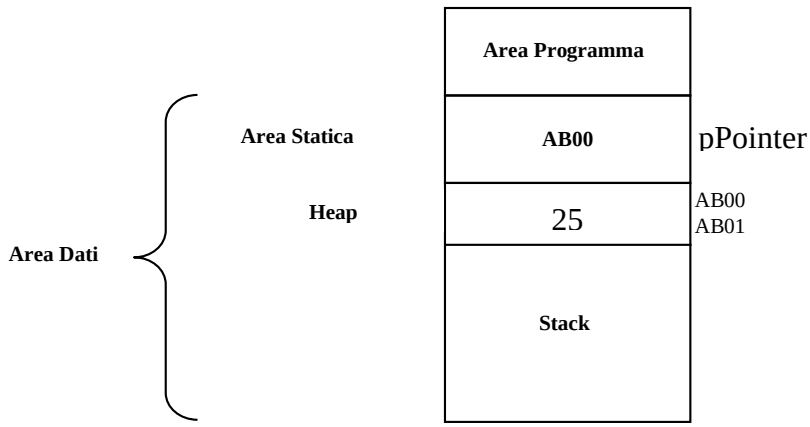
Si consideri il programma già visto in precedenza:

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

int *pPointer;

int main(void)
{
      pPointer = (int *)malloc(sizeof(int));
      *pPointer = 25;
      printf("\nValore di *pPointer = %d ", *pPointer);
}
```

Appena il programma viene caricato in memoria per essere eseguito la memoria apparirebbe come mostrato dalla seguente figura:



A seguito dell'esecuzione del main(), l'area Heap cresce per poter allocare un intero, che viene posto pari al valore 25 dall'istruzione *pPointer=25.



3.Allocazione Dinamica di un Vettore

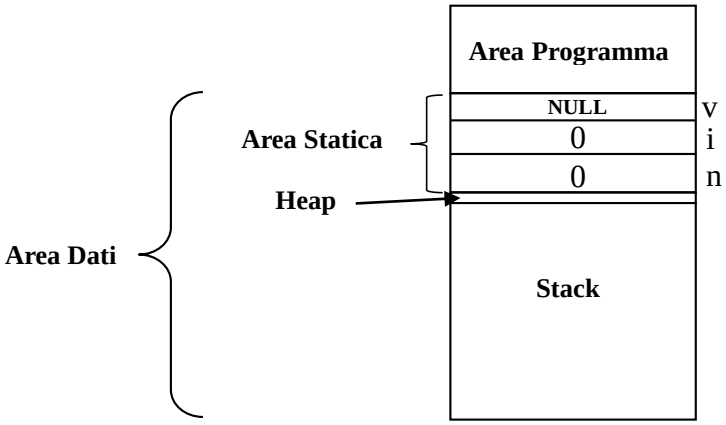
L'allocazione dinamica di un vettore viene realizzata utilizzando una variabile puntatore al tipo di appartenenza di ciascun elemento del vettore (se si vuole allocare un vettore di float, bisogna utilizzare un puntatore ad un float). L'allocazione viene realizzata utilizzando la funzione *malloc()*, ma specificando la dimensione totale del vettore ossia il prodotto del numero di elementi da allocare per la dimensione di ciascun elemento.

Si consideri il seguente programma:

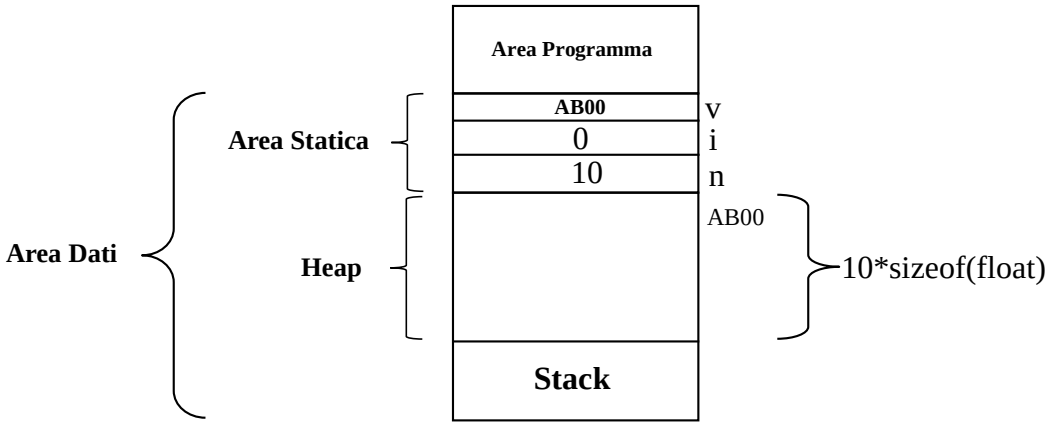
```
#include<stdio.h>
#include<stdlib.h>

float *v;
unsigned int i,n;

int main(void)
{

  do {
    printf ("Inserisci il valore di n > 1  ");
    scanf("%u",&n);
  } while (n<2);

  v=(float *)malloc(n*sizeof(float));

  for (i=0; i<n; i++) {
      printf ("\nInserisci un valore reale ");
      scanf("%f",&v[i]);
  }

  for (i=0; i<n; i++)
      printf ("\nIl valore dell'elemento di indice %d e' %f \n",i,v[i]);

  free(v);
}
```

Come si vede il programma utilizza la variabile statica *v* (puntatore a float). Nel main viene chiesta all'utente la dimensione del vettore da allocare (*n* è la variabile contenete la dimensione). Il programma richiede che n sia maggiore di 1 (vettori di dimensione 0 o 1 non hanno senso!). La funzione *malloc()* permette di allocare una dimensione di memoria contigua pari al prodotto di *n* per la dimensione di un float. Il programma continua con il riempimento del vettore allocato e con la visualizzazione del suo contenuto. Si noti che il programma finisce con *free(v)*, che provoca la deallocazione dell'area occupata dal vettore. Anche in questo caso, il comando *free()* non pone a NULL la variabile v, che rimane al suo attuale valore.

Per quanto riguarda l'uso della memoria, la seguente figura mostra l'area statica e l'Heap appena il precedente programma viene caricato nell'area programma per essere eseguito.



Appena il programma viene eseguito, supponendo che l'utente (ossia chi esegue in quel momento il programma) inserisce 10, l'area dati appare come mostrato nella seguente figura.



Come si vede l'Heap è stato allargato per allocare uno spazio di memoria contigua avente dimensione pari al prodotto tra 10 e la dimensione di un *float*. Nella figura è stato supposto che tale area inizi dall'indirizzo AB00.