

ALLOCAZIONE DINAMICA: I PUNTATORI

Sappiamo che una variabile è un'area di memoria a cui viene dato un nome, con la dichiarazione

```
int num;
```

facciamo in modo di riservare un'area di memoria che viene individuata dal nome num per memorizzare un dato di tipo integer e successivamente possiamo accedere al valore memorizzato mediante il suo nome. Se in seguito eseguiamo questa istruzione

```
num = 20;
```

il computer memorizza il valore intero 20 in un'area di memoria identificata dal nome num. Naturalmente il calcolatore non ha scritto nella propria memoria il nome simbolico della variabile ma per individuare la posizione della variabile "Num" nella memoria RAM assegna un numero univoco, chiamato INDIRIZZO. Possiamo quindi dire che una generica variabile è caratterizzata da 4 elementi:

nome: num	A00E	...
tipo: int	A010	20
valore: 20	A012	...
indirizzo: A010		

Per conoscere l'indirizzo di una variabile nel linguaggio C si utilizza l'operatore indirizzo "&" per esempio la scritta **&num** ci restituisce l'indirizzo di memoria della variabile di nome num

```
printf ("%p",&num);
```

L'indicatore di conversione "% p" serve a stampare l'indirizzo in un formato esadecimale

fa apparire sullo schermo l'indirizzo della locazione di memoria dove è memorizzata la variabile di nome num, nell'esempio di sopra, A010.

Il linguaggio C ha tra i suoi tipi di dati standard le variabili di tipo puntatore; in una variabile di tipo puntatore è possibile memorizzare l'indirizzo di una locazione di memoria, un puntatore, cioè, contiene l'indirizzo di memoria di un'altra variabile (variabile puntata)



I PUNTATORI SONO VARIABILI I CUI VALORI SONO INDIRIZZI DI LOCAZIONI IN CUI SONO MEMORIZZATE ALTRE VARIABILI.

Dato che il puntatore è un tipo di dato standard come l'int, il char, etc ... per utilizzare le variabili di tipo puntatore dobbiamo dichiararla.

- Sintassi
`tipo *nomeVariabile;`
- Esempi
`int *punt;`
`char x, *p, *q, y;`
x e y sono variabili di tipo char
p e q sono variabili di tipo *puntatore-a-char*

x e y sono variabili di tipo char, p e q sono variabili di tipo puntatore, contengono un numero e questo numero è l'indirizzo di memoria riservato per contenere una variabile di tipo char (p e q sono variabili di tipo puntatore a char), infine punt è una variabile di tipo puntatore ad un intero.

I puntatori sono fondamentalmente delle variabili, come quelle intere, reali e carattere. Tuttavia, l'unica differenza consiste nel fatto che essi non contengono un valore numerico od alfanumerico, ma un

puntatore (ossia un indirizzo) alla locazione di memoria dove è memorizzato un certo valore. Tale valore può essere di qualunque tipo, ad esempio intero, reale o carattere.

Vediamo due esempi per capire meglio la differenza tra variabile e variabile puntatore:

Esempio: `int a = 5;`

Proprietà della variabile a :	nome:	a	A00E	...
	tipo:	int	A010	5
	valore:	5	A012	...
	indirizzo:	A010		

Finora abbiamo usato solo le prime tre proprietà. Come si usa l'indirizzo?

`&a` ... **operatore indirizzo** "&" applicato alla variabile **a**
⇒ ha valore 0xA010 (ovvero, 61456 in decimale)

Gli indirizzi si utilizzano nelle variabili di tipo puntatore, dette anche **puntatori**.

Esempio: `int *pi;`

Proprietà della variabile pi :	nome:	pi
	tipo:	puntatore ad intero (ovvero, indirizzo di un intero)
	valore:	inizialmente casuale
	indirizzo:	fissato una volta per tutte

Con la scrittura

`int *pi;`

effettuiamo la **dichiarazione** della variabile "**pi**" di tipo **PUNTATORE AD UN INTERO** che, naturalmente, **AVRÀ UNA SUA COLLOCAZIONE TRA LE LOCAZIONI DI MEMORIA** come tutte le altre variabili (nell'esempio di figura **pi** contiene A200).

A00E	...	
A010	20	num
A012	...	
	...	
A200	?	pi
A202	...	

Con la dichiarazione della variabile puntatore **pi**, come vediamo sopra, **non definiamo** il contenuto di **pi** (nella figura di sopra abbiamo messo un "?") ma solo dove essa si trova in memoria (il suo indirizzo). Si possono avere puntatori a qualsiasi tipo di variabile.

La definizione di una variabile puntatore, come vediamo, avviene nello stesso modo con cui si definisce una variabile, eccetto che dobbiamo aggiungere un asterisco tra il tipobase e l'identificatore del puntatore (ossia 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.

La dichiarazione di un puntatore include IL TIPO dell'oggetto a cui il puntatore punta.

- L'operatore **&** (operatore unario, o monadico) fornisce l'indirizzo di una variabile.
- L'operatore ***** (operatore indiretto, o non referenziato) da' **IL CONTENUTO** dell'oggetto a cui punta un **puntatore**.

L'operatore **&** (E-commerciale) deve essere letto come "l'indirizzo di" (Address-of) e restituisce l'indirizzo di memoria della variabile e non la variabile. Quindi una variabile puntatore può essere inizializzata usando l'operatore **&** di indirizzo

```
int *pi, num; //dich. congiunta variabile punta_intero e intero
num = 20;
pi = &num; //contiene l'indirizzo della variabile num
```

Dopo aver eseguito il segmento di codice di sopra la situazione è la seguente:

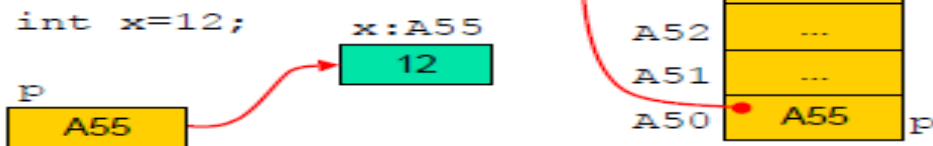
nella cella A200 (in fase di **inizializzazione** viene assegnato l'**indirizzo A200** alla **variabile puntatore pi**) viene memorizzato l'indirizzo di memoria dove è contenuta la variabile **num**, cioè A100 (in fase di **inizializzazione** viene assegnato l'**indirizzo A010** alla **variabile num**). Ora la variabile **pi** punta alla variabile **num** e graficamente si rappresenta con



■ Esempio

p è un puntatore e "punta" alla variabile x che in questo esempio ha indirizzo di memoria A55)

```
int x=12;
```



Ad un puntatore si possono assegnare solo indirizzi di memoria o il valore NULL che indica che il puntatore non punta a nulla. La costante 0 in un contesto dove è atteso un puntatore (inizializzazione e assegnamento di un puntatore, confronto con un puntatore) viene convertita in NULL dal compilatore.

L'asterisco usato nella dichiarazione serve a definire il tipo di dati, quindi, la dichiarazione `int *p`, rappresenta la dichiarazione di una variabile di tipo `int` * (puntatore ad intero).

L'**operatore di indirizzo** '&' ("ampersand") calcola l'indirizzo di memoria di una variabile (si dice che ne restituisce il puntatore); nella maggior parte degli ambienti di compilazione in ANSI C, i puntatori occupano 4 bytes.

`int *p = &x;` **inizializza p con l'indirizzo di x**

OPPURE

`int *p;`

`p = &x;` **assegna a p l'indirizzo di x**

facciamo un primo esempio per capire meglio variabile e variabile puntatore

```
#include <stdio.h>
main()
{
    int b,c; /* definisco 2 variabili intere */
    int *a; /* definisco a come puntatore a intero */
    b=10; /* assegno alla variabile b il valore 10 */
    a=&b; /* assegno alla variabile puntatore a l'indirizzo di memoria della variabile b */
    c=*a; /* assegno alla variabile c il valore contenuto all'indirizzo di memoria che è puntato da a
           quindi siccome a contiene l'indirizzo in memoria della variabile b
           c sarà uguale a b */
    printf("a = %d\n",a);
    printf("b = %d\n",b);
    printf("c = %d\n",c);

    printf("\n\n");
    system("PAUSE");
}
```

Se lanciamo il programma otteniamo

```
a = 2293620
b = 10
c = 10

Premere un tasto per continuare . . . _
```

Vediamo che **a è un puntatore** e contiene l'indirizzo di memoria a cui punta (a punta all'indirizzo di memoria 2293620), **b** e **c** sono variabili numeriche e contengono due numeri; **b** contiene il valore numerico 10 che gli è stato assegnato dall'istruzione

b=10;

c contiene il valore contenuto all'**indirizzo di memoria** che è specificato da **a** quindi siccome **a** contiene l'**indirizzo in memoria** di **b**, **c** sarà uguale a **b** quindi conterrà anch'esso 10

In conclusione

<code>printf("X = %d\n", *a);</code>	stampa contenuto della variabile puntata dal puntatore *a
<code>printf("Y = %p\n", a);</code>	stampa indirizzo di memoria memorizzato nel puntatore *a

vediamo ancora questo esempio per capire meglio

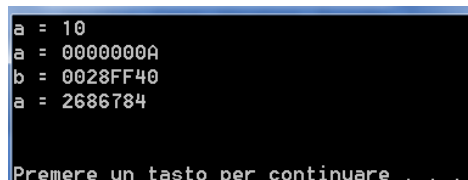
```
#include <stdio.h>
main()
{
int *a, b; /* definisco a come puntatore a intero e b variabile intera */

b=10; /* assegno alla variabile b il valore 10 */
a=&b; /* assegno alla variabile puntatore a l'indirizzo di memoria della variabile b */

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

printf("\n\n");
system("PAUSE");
}
```

Se lanciamo il programma otteniamo



```
a = 10
a = 0000000A
b = 0028FF40
a = 2686784
Premere un tasto per continuare . . .
```

OPERATORE DI DEFERENZIAZIONE

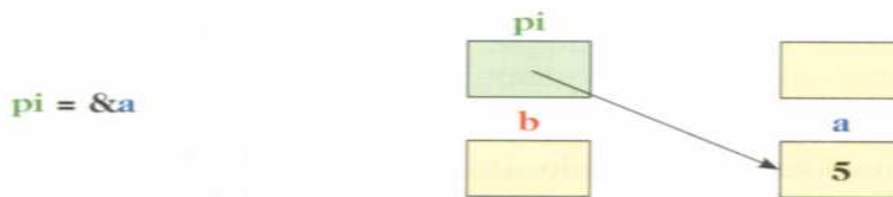
Come di già detto, l'operatore ***** ha la seguente capacità, **se viene applicato ad una variabile puntatore, restituisce il valore memorizzato nella variabile a cui punta**. Quindi riassumendo

- **pi** memorizza l'**indirizzo di memoria** di una variabile (si dice pi punta a una variabile)
- ***pi** restituisce il **valore memorizzato nella variabile** a cui punta pi

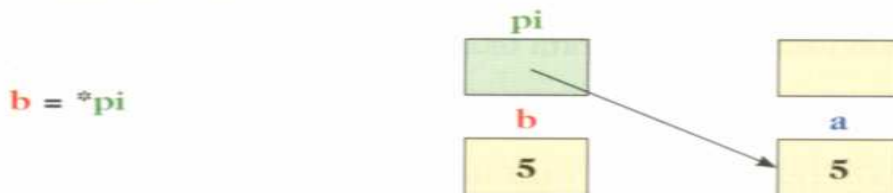
L'operatore * viene chiamato operatore di deferenziiazione

Vediamo la figura di sotto per cercare di capire meglio

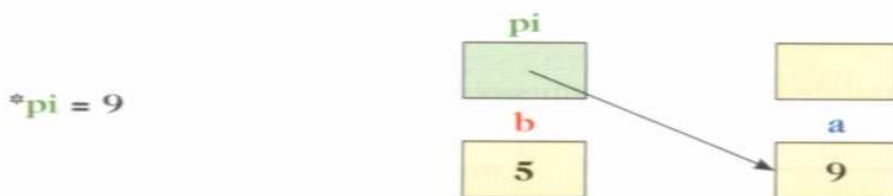
A nella variabile `pi` viene messo l'indirizzo della variabile `a`



B nella variabile `b` viene messo il valore della variabile indirizzata da `pi`



C nella variabile indirizzata da `pi` viene il valore 9



Vediamo che:

- `&a` è un puntatore, cioè un **indirizzo di memoria**
- `*pi` è un **valore intero**, cioè **il valore presente nella cella di memoria indirizzata da `pi`**, ed essendo un intero posso operare su di esso con le operazioni su numeri interi, per esempio

```
*pi = *pi + 4;
```

Vediamo adesso un esempio di programma in C in cui utilizziamo l'operatore di dereferenziazione “`*`”

```
/* Scopo: relazione tra gli operatori di dereferenziazione * e indirizzo & */
```

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

```
main()
```

```
{
```

```
    int num=7,*pi; // dic. var. intera num e punta _intero pi
```

```
    pi = &num;    //al puntatore assegno l'indirizzo
```

```
    // modalità di visualizzazione dell'indirizzo della variabile num
```

```
    printf("indirizzo di num = 0x%X\n", &num); // indirizzo di mem della variabile num
```

```
    printf("valore di punta = 0x%X\n", pi); // indirizzo di mem memorizzato in pi
```

```
    printf("valore di *&punta = 0x%X\n", &*pi); // indirizzo di mem della variabile puntata dal puntatore pi
```

```
    putchar('\n');
```

```
    // modalità visualizzazione del valore della variabile num
```

```
    printf("valore di num = %d\n", num);
```

```
    printf("valore di *punta = %d\n", *pi);
```

```
    printf("valore di *&num = %d\n", *&num);
```

```
    printf("\n\n");
```

```
    system("PAUSE");
```

```
}
```

Se lanciamo il programma otteniamo

```
LA FUNZIONE printf - 2017/18 - Esercizio 1 - deferenzazione.exe
indirizzo di num = 0x22FF74
valore di punta = 0x22FF74
valore di &*punta = 0x22FF74

valore di num = 7
valore di *punta = 7
valore di *&num = 7

Premere un tasto per continuare . . .
```

Per migliorare l'output abbiamo utilizzato il formato esadecimale “0x%X” per visualizzare il contenuto dei puntatori, avremmo potuto usare, come prima, il formato “%p” ottenendo

```
LA FUNZIONE printf - 2017/18 - Esercizio 1 - deferenzazione.exe
indirizzo di num = 0022FF74
valore di pi = 0022FF74
valore di &*pi = 0022FF74

valore di num = 7
valore di *pi = 7
valore di *&num = 7

Premere un tasto per continuare . . .
```

Bisogna prestare attenzione a non confondere i due utilizzi dell'operatore *

- * in una dichiarazione serve per **dichiarare** una variabile puntatore
int *pi;
- * in un'espressione è l'operatore di deferenzazione e **corrisponde ad un valore**
***pi = *pi + 4;**
printf("%d\n", *pi);

LA FUNZIONE sizeof ()

Il linguaggio C ha nella sua libreria la funzione sizeof() che restituisce il numero di byte di occupazione di memoria di una variabile, l'abbiamo di già incontrata ed utilizzata nella gestione dei file; questa funzione è pure applicabile anche al tipo puntatore. Vediamo un esempio dove creiamo tre puntatori a variabili di tipo diverso (char, int, double) e visualizziamo le rispettive occupazioni di memoria.

```
/* Scopo: sizeof per tipi puntatore */
#include <stdio.h>
main ()
{
    char *pchar;
    int *pint;
    double *pdouble;
    // visualizza le dimensioni dei PUNTATORI
    printf("sizeof(char *) = %d, sizeof(int *) = %d, sizeof(double *) = %d\n\n",
        sizeof(char*), sizeof(int*), sizeof(double*));
    // visualizza le dimensioni delle VARIABILI
    printf("sizeof(char) = %d, sizeof(int) = %d, sizeof(double) = %d\n",
        sizeof(char), sizeof(int), sizeof(double));
    printf("\n\n");
    system("PAUSE");
}
```

Mandando in esecuzione il programma otteniamo il seguente output

```
sizeof(char *) = 4, sizeof(int *) = 4, sizeof(double *) = 4
sizeof(char) = 1, sizeof(int) = 4, sizeof(double) = 8

Premere un tasto per continuare . . .
```

Possiamo osservare che i puntatori hanno tutti la medesima dimensione (4byte), anche se puntano a variabili di tipo diverso, questo è un risultato prevedibile in quanto i puntatori contengono un indirizzo di memoria indipendentemente dal tipo di variabile che puntano. Diverso è invece il risultato per quanto

riguarda lo spazio di memoria occupato dalle variabili (la dimensione delle variabili); sappiamo infatti che una variabile di tipo intero occupa 4 byte, una variabile di tipo double 8 byte, una variabile di tipo char 1 byte, e questo è quello che otteniamo.

OPERAZIONI CON I PUNTATORI

Le operazioni che possono essere fatte sul tipo puntatore prendono il nome di aritmetica dei puntatori. Con queste operazioni è possibile effettuare gli accessi a strutture di dati omogenei conservati in posizioni contigue della memoria in modo semplice e diretto. Vediamo adesso alcuni esempi per capire meglio.

OPERAZIONE DI SOMMA:

Cominciamo con l'operazione di somma: con l'operazione di somma di un intero ad un puntatore effettuiamo un'operazione tra due tipi diversi, lo scopo è quello di ottenere il posizionamento del puntatore all'elemento successivo di memoria. Vediamo un esempio dove sommiamo a pa (puntatore a int) il numero 1 con il relativo output

```
int *pa;
pa = malloc(sizeof(int)); // allocazione intero
printf("indirizzo: 0x%x\n", pa);
pa++; // punta alla cella int successiva
printf("indirizzo: 0x%x\n", pa);
```

```
indirizzo: 0x3e2468
indirizzo: 0x3e246c
```

In generale se pa è un puntatore ad un certo tipo il suo valore è un certo indirizzo, eseguendo $pa + num$ viene aggiunto num volte l'indirizzo necessario per accedere correttamente alla posizione di una variabile del tipo di partenza num posizioni più avanti nella memoria.

Questa operazione ha "senso" solo per la manipolazione diretta degli array, in quanto gli elementi sono memorizzati sequenzialmente: se p (puntatore a intero) contiene l'indirizzo della prima cella di un array di interi, p+1 produce l'indirizzo della seconda cella, p+2 l'indirizzo della terza e via dicendo.

L'aritmetica dei puntatori permette, pertanto, di recuperare i dati da un array con una notazione alternativa rispetto a quella tradizionale che si basa sugli indici: l'istruzione

$*(v1+i)$

equivale a

$v1[i]$

dato che con $*v1$ indichiamo l'indirizzo di base (di partenza) del vettore che è anche l'indirizzo del primo elemento $v1[0]$, pertanto $*(v1+4) = v1[4]$ e così via.

SOTTRAZIONE DI UN INTERO AD UN PUNTATORE:

E' l'operazione opposta alla somma, il valore nel puntatore è l'indirizzo di num posizioni precedenti a quella iniziale all'interno della memoria

```
int *pi;
pi = malloc(sizeof(int)); // allocazione intero
printf("indirizzo: 0x%x\n", pi);
pi=pi-2; // punta a due interi precedenti
printf("indirizzo: 0x%x\n", pi);
```

```
indirizzo: 0x3e24d0
indirizzo: 0x3e24c8
```

OPERATORE DI DIFFERENZA TRA DUE PUNTATORI:

Per eseguire la differenza tra due puntatori è necessario che i due operandi siano puntatori allo stesso tipo, come risultato si ottiene la differenza di elementi del tipo base che sono interposti tra i due indirizzi.

Viene cioè effettuata la differenza aritmetica tra gli indirizzi diviso per il numero di byte di occupazione di memoria del tipo base.

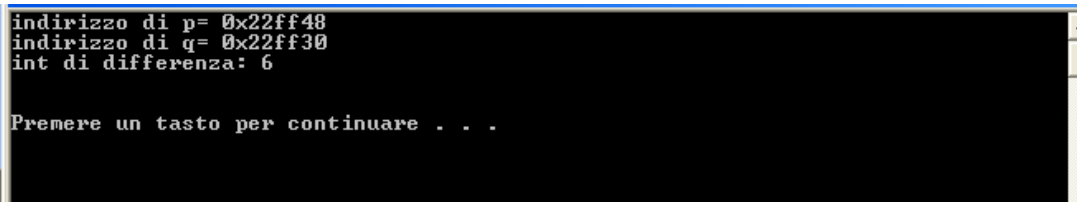
Anche l'operazione di differenza tra puntatori è significativa solo se i due operandi contengono gli indirizzi di due celle del medesimo array e se il tipo base dell'array coincide con quello dei due puntatori.

In questo caso, infatti, la differenza tra i due puntatori corrisponde al numero di celle dell'array che separano la cella puntata del puntatore di valore minore da quella del puntatore di valore maggiore

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    //differenza tra puntatori
    int *p, *q, x;
    int a[10]={0};
    q = a;
    p = &a[6];
    x=p-q;
    printf("indirizzo di p= 0x%x\n", p);
    printf("indirizzo di q= 0x%x\n", q);
    printf("int di differenza: %d\n", x);

    printf("\n\n");
    system("PAUSE");
}
```

Mandando in esecuzione il programma otteniamo il seguente output



```
indirizzo di p= 0x22ff48
indirizzo di q= 0x22ff30
int di differenza: 6

Premere un tasto per continuare . . .
```


VARIABILI DINAMICHE

Prima di andare avanti con i puntatori introduciamo il concetto di variabile dinamica:

Variabili Dinamiche

In C e' possibile classificare le variabili in base al loro tempo di vita.

Due categorie:

- variabili **automatiche**
- variabili **dinamiche**

Variabili automatiche:

- L'allocazione e la deallocazione di variabili automatiche e' effettuata **automaticamente** dal sistema (senza l'intervento del programmatore).
 - Ogni variabile automatica ha un **nome**, attraverso il quale la si puo' riferire.
 - Il programmatore non ha la possibilita' di influire sul tempo di vita di variabili automatiche.
- ➔ tutte le variabili viste finora rientrano nella categoria delle **variabili automatiche**.

Variabili Dinamiche

Variabili dinamiche:

- Le variabili dinamiche devono essere allocate e deallocate **esplicitamente** dal programmatore.
- L'area di memoria in cui vengono allocate le variabili dinamiche si chiama **heap**.
- Le variabili dinamiche non hanno un **identificatore**, ma possono essere riferite soltanto attraverso il loro indirizzo (mediante i **puntatori**).
- Il tempo di vita delle variabili dinamiche e' l'intervallo di tempo che intercorre tra l'allocazione e la deallocazione (che sono impartite esplicitamente dal programmatore).

Variabili Dinamiche in C

Il C prevede funzioni standard di allocazione e deallocazione per variabili dinamiche:

- **Allocazione:** malloc
- **Deallocazione:** free

malloc e free sono definite a livello di sistema operativo, mediante la libreria standard <stdlib.h> (da includere nei programmi che le usano).

Variabili Dinamiche

Allocazione di variabili dinamiche:

La memoria dinamica viene allocata con la funzione standard malloc. La **sintassi** da usare è:

```
punt = (tipodato *)malloc(sizeof(tipodato));
```

dove:

- **tipodato** è il tipo della variabile puntata
- **punt** è una variabile di tipo **tipodato ***
- **sizeof()** è una funzione standard che calcola il numero di bytes che occupa il dato specificato come argomento
- è necessario convertire esplicitamente il tipo del valore ritornato (*casting*):

```
(tipodato *) malloc(..)
```

Significato:

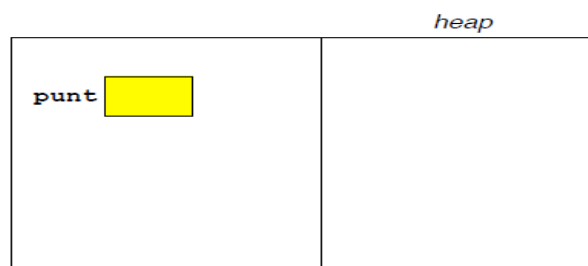
La **malloc** provoca la creazione di una variabile dinamica nell'**heap** e restituisce l'indirizzo della variabile creata.

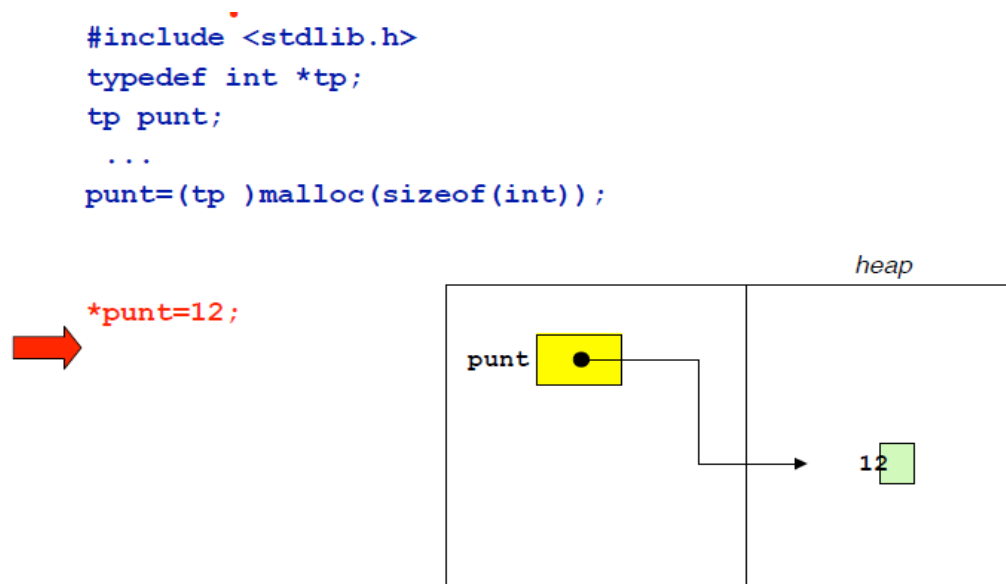
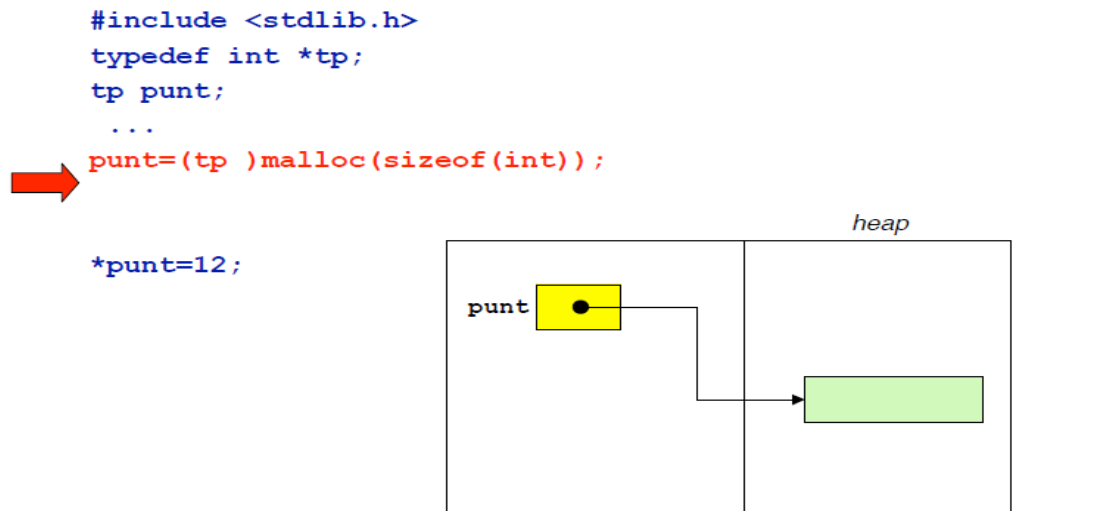
Vediamo questo esempio per capire meglio

Esempio:

```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
punt=(tp )malloc(sizeof(int));
```

```
*punt=12;
```





Variabili dinamiche

Deallocazione:

Si rilascia la memoria allocata dinamicamente con:

```
free(punt);
```

dove `punt` e' l'indirizzo della variabile da deallocare.

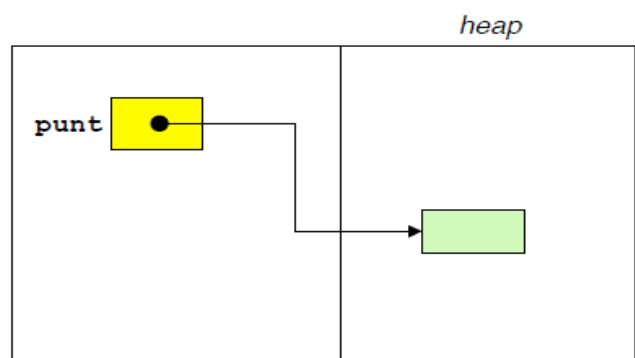
→ Dopo questa operazione, la cella di memoria occupata da `*punt` viene liberata: **punt non esiste piu'*.

Esempio:

```

#include <stdlib.h>
typedef int *tp;
tp punt;
...
punt=(tp )malloc(sizeof(int));
*punt=12;
...<uso di punt>...
free(punt);

```



Si rilascia la memoria allocata dinamicamente con:

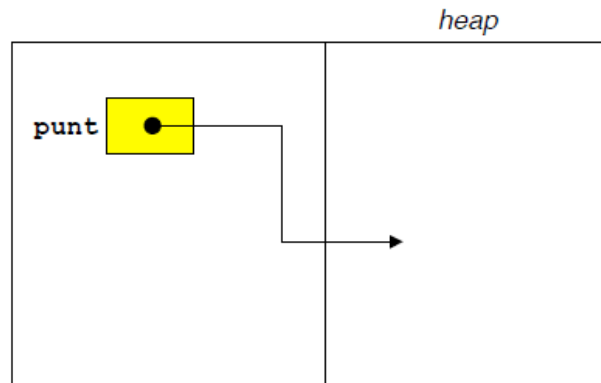
`free (punt) ;`

dove `punt` e' l'indirizzo della variabile da deallocare.

→ Dopo questa operazione, la cella di memoria occupata da `*punt` viene liberata: `*punt non esiste piu'`.

Esempio:

```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
punt=(tp )malloc(sizeof(int));
*punt=12;
...<uso di punt>...
free (punt) ;
```



Vediamo adesso alcuni esempi per capire meglio l'uso dei puntatori e delle istruzioni di sopra.

Il metodo standard per creare un `array` di dieci interi:

```
int array[10];
```

Tuttavia, se si vuole assegnare un simile array in modo dinamico, si può utilizzare il seguente codice:

```
/* Alloca spazio per un array con 10 elementi di tipo int. */
int *ptr = (int *)malloc(10 * sizeof (int));
if (ptr == NULL) {
    /* La memoria non può essere allocata, il programma dovrebbe gestire l'errore in modo appropriato. */
} else {
    /* Allocazione avvenuta con successo. Prosegui con il programma. */
    free(ptr); /* Abbiamo finito di lavorare con l'array, e liberiamo il puntatore associato. */
    ptr = NULL; /* Il puntatore non può essere più usato fino ad un riassegnamento effettuato con malloc. */
}
```

Il seguente esempio riguarda l'accesso in memoria mediante puntatori

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ // accesso mediante puntatore
    int *pi, dim, x;
    printf("Inserisci il numero di elementi: ");
    scanf ("%d",&dim);

    /* alloco una quantità di memoria necessaria per memorizzare dim numeri interi */
    /* il puntatore pi punterà esattamente all'inizio di questa area */
    pi=(int*)malloc(dim*sizeof(int));

    if (pi == NULL) { // verifica corretta allocazione
        printf("Non ho abbastanza memoria per l'allocazione\n");
        system("PAUSE");
        exit(1); //terminazione errata
    }
```

```

for(x=0;x<dim;x++)
{
    printf ("\ninserisci il numero %d :", x+1);
    scanf ("%d",&pi[x]); // inserisco i numeri a partire da &pi[0]
}
for (x=0;x<dim;x++) // li visualizzo direttamente con *(pi+offset)
    printf("\n numero %d =%d",x+1,*(pi+x));
printf("\n\n");
system("PAUSE");
return 0; //terminazione corretta
}

```

Li inserisco in un modo, li visualizzo in un altro modo

Mandando in esecuzione il programma otteniamo il seguente output

```

Inserisci il numero di elementi: 4
inserisci il numero 1 :45
inserisci il numero 2 :2
inserisci il numero 3 :16
inserisci il numero 4 :21

numero 1 =45
numero 2 =2
numero 3 =16
numero 4 =21

Premere un tasto per continuare . . .

```

Come secondo esempio vediamo l'**allocazione di un vettore** mediante puntatori

```

#include <stdio.h>
#include <stdlib.h>
int *vetAlloc(int numEle, int dim)
{
    void *p;
    /* p punta al vettore di numEle elementi */
    p=malloc(numEle*dim);
    if (p==NULL){
        printf("errore di memoria!\n");
        exit(-1);
    }
    return p;
}

int main()
{ // accesso mediante puntatore
    int *pi, num1, num2, dim, x;
    printf("Inserisci il numero di elementi da memorizzare: ");
    scanf ("%d",&dim);
    pi = (int *)vetAlloc(dim,sizeof(int));
    if (pi == NULL) { // verifica corretta allocazione
        printf("Non ho abbastanza memoria per l'allocazione\n");
        system("PAUSE");
        exit(1); //terminazione errata
    }
    for(x=0;x<dim;x++)
    {
        printf ("\ninserisci il numero %d :", x+1);
    }
}

```

```

scanf ("%d",&pi[x]); // inserisco i numeri a partire da &pi[0]
}
for (x=0;x<dim;x++) // li visualizzo direttamente con *(pi+offset)
    printf("\n numero %d =%d",x+1,*(pi+x));
printf("\n\n");
system("PAUSE");
return 0; //terminazione corretta
}

```

Mandando in esecuzione il programma otteniamo il seguente output

```

Inserisci il numero di elementi da memorizzare: 3
inserisci il numero 1 :34
inserisci il numero 2 :5
inserisci il numero 3 :12
numero 1 =34
numero 2 =5
numero 3 =12
Premere un tasto per continuare . . .

```

Come ultimo esempio guardiamo questo programma sulle matrici allocate dinamicamente

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    int **matrice, righe, colonne, x,y;
    righe=3; // 3 righe
    colonne =5; // 5 colonne

    // Alloco il vettore delle righe: ogni elemento è un puntatore
    matrice =(int**) malloc(righe*sizeof (int *));

    // per ogni riga alloco le colonne composte da interi
    for (x=0;x<righe;x++)
        matrice[x]=(int*)calloc(colonne,sizeof(int));

    // scrivo nelle celle delle matriche per riga
    for (y=0;y<righe;y++) // per ogni riga
        for (x=0;x<colonne;x++) // per ogni colonna
            *(matrice[y]+x)=(y+1)*(x+1);
    printf("\n\n visualizzo indirizzi base e inizio di ogni riga ");
    for (x=0;x<righe;x++) //visualizza puntatori alle righe
        printf("\n %p %p",matrice+x,*(matrice+x));

    printf("\n\n visualizzo il contenuto e l'indirizzo di ogni cella \n");
    for (y=0;y<righe;y++){ // per ogni riga
        for (x=0;x<colonne;x++) // visualizza contenuto di ogni riga
            printf(" %2d ",*(matrice[y]+x)); //visulizza il dato
        for (x=0;x<colonne;x++)
            printf(" %p",matrice[y]+x); //visulizza l'indirizzo della cella
        printf("\n");
    }
}

```

****matrice** è un modo per dichiarare un puntatore ad una matrice

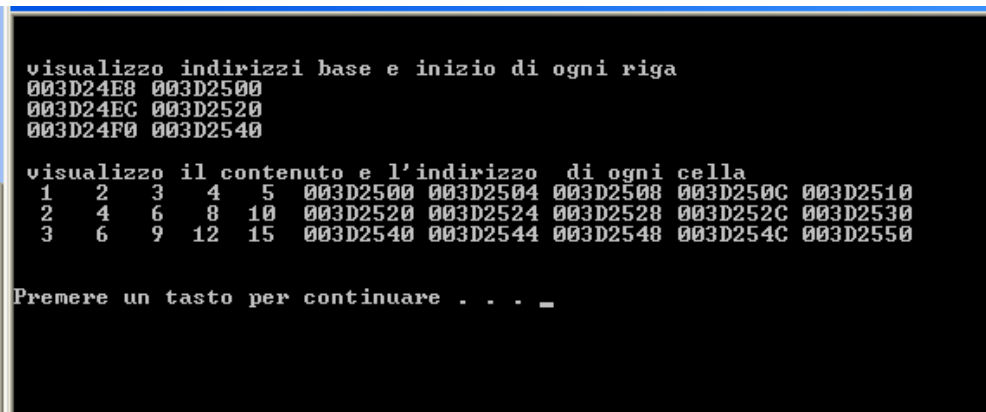
```

// deallocazione della memoria
// per ogni riga dealloco le colonne
for (x=0;x<righe;x++)
    free (matrice[x]);
// dealloco il vettore delle righe
free (matrice);

printf("\n\n");
system("PAUSE");
}

```

Mandandolo in esecuzione otteniamo il seguente output



```

visualizzo indirizzi base e inizio di ogni riga
003D24E8 003D2500
003D24EC 003D2520
003D24F0 003D2540

visualizzo il contenuto e l'indirizzo di ogni cella
1 2 3 4 5 003D2500 003D2504 003D2508 003D250C 003D2510
2 4 6 8 10 003D2520 003D2524 003D2528 003D252C 003D2530
3 6 9 12 15 003D2540 003D2544 003D2548 003D254C 003D2550

Premere un tasto per continuare . . . _

```

APPROFONDIMENTI

Vediamo infine di approfondire e capire meglio le funzioni “malloc” (derivato dai termini *memory allocation*), “calloc”, “free” e “realloc” per l’allocazione dinamica della memoria

La funzione “malloc” serve a chiedere una nuova zona di memoria da usare. Quello che succede quando viene chiamata la funzione è che il calcolatore individua una zona di memoria libera, ossia una zona che non è attualmente occupata da variabili o usata in altro modo, e ne restituisce l’indirizzo iniziale. Da questo momento in poi, la zona di memoria non è più libera, ma è attualmente in uso. Questa zona non deve quindi venire più trattata come una zona libera, per cui non deve essere usata per altre variabili e non deve venire restituita da successive chiamate della funzione “malloc”. Se così non fosse, si potrebbero verificare situazioni in cui due puntatori, che ci aspettiamo non essere correlati, 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.

La “calloc” dopo aver allocato memoria, inizializza lo spazio di memoria indirizzato a 0. L’unica differenza tra malloc e calloc è che malloc alloca semplicemente mettendo nei vari campi di un vettore (per esempio) valori casuali. Mentre calloc alloca dinamicamente e inoltre inizializza tutti i campi del vettore a 0.

La funzione “realloc” si usa invece per cambiare (in genere aumentare) la dimensione di un’area di memoria precedentemente allocata, la funzione vuole in ingresso il puntatore restituito dalla precedente chiamata ad una “malloc” ad esempio quando si deve far crescere la dimensione di un vettore. In questo caso se è disponibile dello spazio adiacente al precedente la funzione lo utilizza, altrimenti rialloca altrove un blocco della dimensione voluta, copiandoci automaticamente il contenuto; lo spazio aggiunto non viene inizializzato.

Si deve sempre avere ben presente il fatto che il blocco di memoria restituito da “realloc” può non essere un’estensione di quello che gli si è passato in ingresso; per questo si dovrà sempre eseguire la riassegnazione di ptr al valore di ritorno della funzione, e reinizializzare o provvedere ad un adeguato aggiornamento di tutti gli altri puntatori all’interno del blocco di dati ridimensionato.

Funzione `malloc`

La funzione `malloc` è dichiarata in `<stdlib.h>` con prototipo:

```
void * malloc(size_t);
```

- prende come parametro la dimensione (numero di byte) della zona da allocare (`size_t` è il tipo restituito da `sizeof` e usato per le dimensioni in byte delle variabili — ad esempio potrebbe essere `unsigned long`)
- alloca (riserva) la zona di memoria
- restituisce il puntatore iniziale alla zona allocata (è una funzione che restituisce un puntatore)

N.B. La funzione `malloc` restituisce un puntatore di tipo `void*`, che è compatibile con tutti i tipi puntatore.

Esempio:

```
float *p;  
p = malloc(4);
```

Uso tipico di `malloc` è con `sizeof(tipo)` come parametro.

Esempio:

```
#include <stdlib.h>

int *p;
p = malloc(sizeof(int));
*p = 12;
(*p)++; /* N.B. servono le parentesi */
printf("*p vale %d\n", *p);
```

- attivando `malloc(sizeof(int))` viene allocata una zona di memoria adatta a contenere un intero; ovvero viene creata una nuova variabile intera
- il puntatore restituito da `malloc` viene assegnato a `p`
⇒ `*p` si riferisce alla nuova variabile appena creata

La zona di memoria allocata attraverso `malloc` si trova in un'area di memoria speciale, detta **heap** (o **memoria dinamica**).

⇒ abbiamo **4 aree di memoria**:

- zona programma: contiene il codice macchina
- stack: contiene la pila dei RDA
- statica: contiene le variabili statiche
- heap: contiene dati allocati dinamicamente

Funzionamento dello heap:

- gestito dal sistema operativo
- le zone di memoria sono **marcate libere o occupate**
 - marcata libera: può venire utilizzata per la prossima `malloc`
 - marcata occupata: non si tocca

Potrebbe **mancare la memoria** per allocare la zona richiesta. In questo caso `malloc` restituisce il puntatore `NULL`.

⇒ Bisogna sempre verificare cosa restituisce `malloc`.

Esempio:

```
p = malloc(sizeof(int));
if (p == NULL) {
    printf("Non ho abbastanza memoria per l'allocazione\n");
    exit(1);
}
...
```

La costante `NULL`

- è una costante di tipo `void*` (quindi compatibile con tutti i tipi puntatore)
- indica un puntatore che non punta a nulla ⇒ non può essere dereferenziato
- ha tipicamente valore 0
- definita in `<stdlib.h>` (ed in altri file header)

