Programmazione I

dott. Sabrina Senatore

Dipartimento di Informatica

Programmazione I

I puntatori

I puntatori

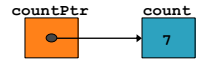
- I puntatori
  - Si usano per simulare la chiamata per riferimento
  - Stretta correlazione con vettori e stringhe

- In generale, in C ogni variabile ha due tipi di valori: una locazione e un valore contenuto in quella locazione

Le variabili puntatore

- Le variabili normali contengono uno specifico valore (riferimento diretto)

- I puntatori contengono l'indirizzo di una variabile che ha uno specifico valore (riferimento indiretto)



- Deriferimento – riferimento a un valore per mezzo di un puntatore

Dichiarazione di una variabile puntatore 1/2

- Il tipo puntatore è un classico esempio di *tipo derivato*: infatti non ha senso parlare di tipo puntatore in generale, ma occorre sempre specificare a quale tipo esso punta.
- Dichiarazione di una variabile riferimento o puntatore:

```
tipo_base *var_punt;
```

dove:

- **tipo_base** è uno dei tipi fondamentali già introdotti: **char, int, float e double.**
- **var_punt** è definita come una variabile di tipo puntatore a **tipo_base**
- **var_punt** è creata per mantenere l'indirizzo di variabili di tipo **tipo_base**

Dichiarazione di una variabile puntatore 2/2

- Dichiarazioni di puntatori:
  - ***** usato con variabili puntatore

    ```
    int *myPtr;
    ```

  - Puntatori multipli richiedono ***** multipli

    ```
    int *myPtr1, *myPtr2;
    ```

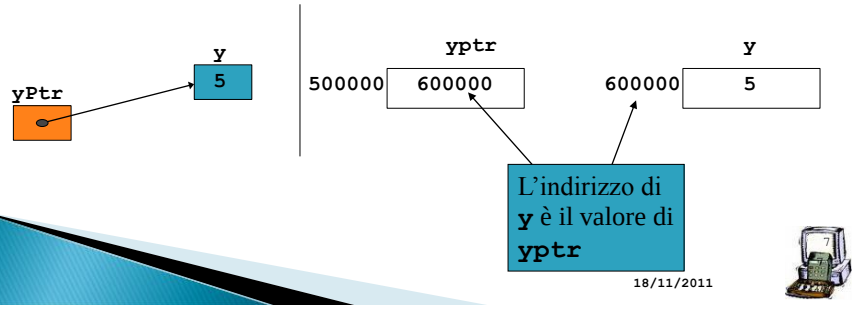
  - E' possibile dichiarare puntatori a qualsiasi tipo di dato
  - I puntatori possono essere inizializzati a 0, a **NULL** (costante simbolica definita in molti file di intestazione) oppure a un indirizzo
    - 0 o **NULL** – non fa riferimento a nessun dato (si preferisce **NULL**)

Inizializzazione di puntatori

- **&** (operatore di indirizzo)
  - Restituisce l'indirizzo dell'operando

```
int y = 5;
int *yPtr;
yPtr = &y;
/* a yPtr viene assegnato l'indirizzo di y */
```

  - **yPtr** "punta a" **y**



Operatore di indirizzo &

- L'operando dell'operatore di indirizzo (&) deve essere una variabile

```
&x    /*x deve essere una variabile*/
```

- Notare:

```
&val          ammesso
&(val+1)      non ammesso
&val = m;     non ammesso
```

- *L'indirizzo di memoria di una variabile non può essere modificato in un'istruzione, ma è consentito usarlo solo come riferimento (è predeterminato e non modificabile)*

L'operatore di dereferenziazione *

- L'operazione di derefenziazione è da considerarsi "inversa" a quella di indirizzo.
- Se assegniamo a un puntatore p l'indirizzo di una variabile a,

```
p = &a;
```

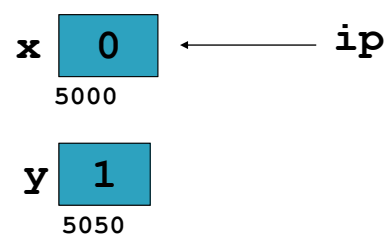
allora

```
*p equivale ad a
```


L'operatore di dereferenziazione *

- L'operatore unario * è detto anche operatore di indirezione.
- Applicato ad un puntatore, consente l'accesso all'oggetto puntato.

```
int x=1, y=2;
int *ip;

ip = &x;
y = *ip;
*ip = 0;
```



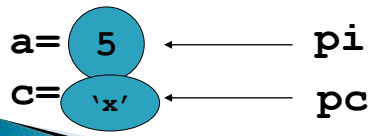
Inizializzazione di un puntatore

- Ci sono due modi di inizializzare un puntatore:
  - un indirizzo
  - un altro puntatore

```
int i   = 10;
int *p1 = &i;
int *p2 = p1;
```


Esempio

```
int a;
char c;
int *pi; /*pi è una variabile di tipo puntatore a int */
char *pc; /* pc è una variabile di tipo puntatore a char */
pi = &a; /* pi è inizializzata con l'indirizzo di a */
pc = &c; /* pc è inizializzata con l'indirizzo di c */
a = 5;   /* ora pi punta all'intero 5 */
c = 'x';  /* ora pc punta al carattere 'x' */
```



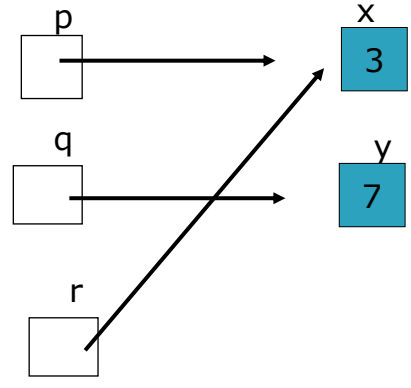
Riassumendo su & e *

- L'operatore **&** ha significato "**indirizzo di**", infatti &x si legge "indirizzo di x"

- *NB:L'operatore * ha tre significati diversi:*
  1. *Operatore di **moltiplicazione** 5*x*
  2. ***Dichiarazione** di puntatore int *x*
  3. *Operatore di **de-riferimento** *y*
     e va applicato a variabili puntatori

Qualche esempio di Puntatori

```
int x =3, y=7;
int *p = &x;
int *q = &y
```




```
int *r = p;
```


Specifica di conversione %p

- La specifica di conversione `%p` restituisce la locazione di memoria a cui si riferisce, convertendola in un formato definito dall'implementazione (molto spesso un intero esadecimale)

```
int *ptr;
        int x = 1123;
        ptr = &x;
        printf("l'indirizzo di x è %p\n", &x);
        printf("il valore di ptr è %p\n", ptr);
```

```
l'indirizzo di x è 001F2BB4
il valore di ptr è 001F2BB4
```

```
1    /* Fig. 7.4: fig07_04.c
2       Usando gli operatori  & e * */
3    #include <stdio.h>
4
5    int main()
6    {
7       int a;
8       int *aPtr;     /* aPtr è un puntatore a
9
10      a = 7;
11      aPtr = &a;     /*  'indirizzo di a è assegnato a a
12
13      printf( "L'indirizzo di a è %p"
14              "\nil valore di aPtr è %p", &a, aPtr );
15
16      printf( "\n\nIl valore di a è %d"
17              "\nil valore di *aPtr è %d", a, *aPtr );
18
19      printf( "\n\nDiostrazione che  * e &  "
20              "sono complementari \n&*aPtr = %p"
21              "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23      return 0;
24
25   } /* end main */
```

L'indirizzo di **a** è il valore di **aPtr**.

L'operatore ***** restituisce ciò a cui punta l'operando. **aPtr** punta ad **a**, così ***aPtr** restituisce **a**.

Si noti come ***** e **&** sono inversi

```
1    /* Fig. 7.4: fig07_04.c
2       Usando gl
3    #include <st
4
5    int main()
6    {
7       int a;
8       int *aPtr
9
10      a = 7;
11      aPtr = &a;    /*  'indirizzo di a è assegnato a aPtr*/
12
13      printf( "L'indirizzo di a è %p"
14              "\nil valore di aPtr è %p", &a, aPtr );
15
16      printf( "\n\nIl valore di a è %d"
17              "\nil valore di *aPtr è %d", a, *aPtr );
18
19      printf( "\n\nDimostrazione che  * e &  "
20              "sono complementari \n&*aPtr = %p"
21              "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23      return 0;
24
25   } /* end main */
```

```
L'indirizzo di a è 0012FF7C
Il valore di aPtr è 0012FF7C

Il valore di  a è 7
Il valore di *aPtr è 7

Dimostrazione che * e & sono complementari.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```


Esercizio: caccia all'errore

```
#include <stdio.h>
int main(){
  int n = 27;
  double x = 2.71;
  int *p = &n;  //inizializzazione
  double *q;    //dichiarazione
  q = &x;       //assegnamento
  q = &n;
  …
```

Tipi incompatibili: n è una variabile intera, mentre q è un puntatore a reali

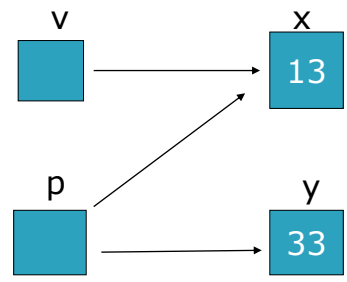
Esercizio

- Scrivere un programma, che dato un array di 100 numeri interi, assegna un valore a due variabili puntatore che puntano all'elemento dell'array contenente, rispettivamente, il valore più basso e il valore più alto.


```
#define DIM_ARRAY 100
main ( ) {
 int i;
 int array[DIM_ARRAY];
 int *punta_a_minore;
 int *punta_a_maggiore;
 punta_a_minore = &array[0];
 i = 1;
 while (i< DIM_ARRAY)
 {
    if (array[i] < *punta_a_minore)
        punta_a_minore = &array[i];
    i = i+1;
 }
 punta_a_maggiore = &array[0];
 i = 1;
 while (i< DIM_ARRAY)
 {
   if (array[i] > *punta_a_maggiore)
        punta_a_maggiore = &array[i];
   i = i+1;
 }
return 0;
}
```


Esercizio

```
…
int x=13, *v =&x;
int y=33, *p=&y;
…
p=v;
printf("%d", *p);
```



- Quale sarà il risultato di questa porzione di codice?
  1. Non compila
  2. Stampa 13
  3. Stampa l'indirizzo della varaibile che contiene 33
  4. Stampa 33

Esercizio

```
int *pi;          /* dichiarazione di un puntatore ad intero */
int a = 5, b;     /* dichiarazione di variabili intere */
pi = &a;          /* pi punta ad a */
b = *pi;         /* assegna a b il valore della variabile puntata
                    da pi, ovvero il valore di a, ovvero 5 */
*pi = 9;         /* assegna 9 alla variabile puntata da pi */
```

- Quanto vale **a**?

a=9

- Notare che se **pi** è di tipo **int ***, allora ***pi** è di tipo **int**.

Puntatori generici

- E' possibile dichiarare una variabile di tipo puntatore senza specificare il tipo:

```
void *pa, *pb;
```

- Tale dichiarazione indica che il tipo di dato a cui il puntatore punta e' indeterminato.
- Ad un puntatore di tipo void può essere assegnato l'indirizzo di qualsiasi tipo di dato.
- Può essere pensato come puntatore a un tipo generico

Esempio

```
...
int *p, i;
float x;
void *p_void;
...
p_void = &i;
p = (int *) p_void;  // casting
p_void = &x;
```


Aritmentica dei puntatori 1/2

- Operazioni aritmetiche ammissibili sui puntatori
  - Incremento/decremento del puntatore
    - ++ opp. --
  - Aggiungere o sottrarre un intero a un puntatore
    - + opp. += , - opp. -=
    - ***Se p è un puntatore, p + 3***
      - indica il terzo oggetto che segue quello puntato da p
      - il risultato è un valore di indirizzo
      - Il compilatore C considera le dimensioni del tipo di oggetto memorizzato: non effettua la semplice somma p+3 ma p + 3*n, dove n è il numero di byte usati per memorizzare l'oggetto puntato.
- Si può sottrarre un puntatore da un altro puntatore

Nota: p un puntatore → p + 3

- Se p è un puntatore a interi (supponiamo che il compilatore riservi 4 byte per int)
  - p + 3 equivale a p + 3*4 = p +12
  - p = 1000 ---> 1012
- Se p fosse dichiarato di tipo char (1 byte per i char), allora
  - p = 1000
  - p= p+3---> 1003

Aritmentica dei puntatori 2/2

- Non si possono sommare puntatori tra di loro
- Se i puntatori si riferiscono allo stesso tipo di oggetto, è possibile sottrarre il valore di un puntatore da un altro
  - Il risultato è un valore intero che rappresenta il numero di celle sono comprese tra i due puntatori
- È possibile sottrarre un intero da un puntatore
  - risultato è il valore di un puntatore

Esercizio

```
int *p1, *p2;
int k;
char *p3;

p2=p1 + 3;   .
//AMMESSO
k = p2-p1;
// AMMESSO (k=3);
k = p1-p2;
// AMMESSO (k=-3) valore negativo
p1=p2-2;
// AMMESSO, i puntatori sono dello stesso tipo
p3 = p1-1;
// NON AMMESSO, i tipi sono diversi
k = p1 - p3;
// NON AMMESSO, i tipi sono diversi
```


L'accesso agli elementi di un array mediante puntatore – I

- Si accede agli elementi di un array mediante l'impiego di un indice, relativo all'array oppure mediante l'uso dei puntatori.
- Si può osservare che il nome stesso di un array non seguito da un indice viene interpretato come un puntatore all'elemento iniziale dell'array

arr è equivalente a &arr[0]

arr[n] equivale a *(arr+n)

- Il compilatore C trasforma il nome di array in un puntatore al primo elemento dell'array e quindi gli indici vengono interpretati come spostamenti dalla posizione di indirizzo base (offset)

Variabili puntatore e nomi di array

```
float arr[7], *p;
p=arr;
    /* AMMESSO equivale a p=&arr[0] */
arr=p;
 /* NON AMMESSO: non è possibile operare assegnamenti su un
   indirizzo di array */
&p=arr;
/* NON AMMESSO : non è possibile operare assegnamenti su un
   indirizzo di puntatore */
arr++;
   /* NON AMMESSO : non è possibile incrementare un indirizzo di
   array */
arr[1]=*(p+5);
 /*  AMMESSO arr[1] è una variabile */
p++;
/*  AMMESSO è possibile incrementare un puntatore */
```


Puntatori vs. Array

- Se p è un puntatore:

p[0] equivale *p	& p[0] equivale p
p[1] equivale *(p+1)	& p[1] equivale p+1
p[i] equivale *(p+i)	& p[i] equivale p+i

Stringhe e puntatori

- Una stringa è un array di caratteri terminato dalla sequenza di escape \0.

```
char stringa[]="benvenuto";
```

- Similmente, è possibile inizializzare un puntatore a char con una stringa costante:

```
char *ptr = "ancora benvenuto";
```

crea un array di caratteri, inizializzato ad "ancora benvenuto".

Esercizio:

Descrivere il comportamento della seguente porzione di codice.

```
int arr1[10];
char arr2[10];
char *p;
int a;
a = &arr1[5] - &arr1[3];
printf("%d\n", a);
a = &arr1[5] - &arr2[3];
printf("%d\n", a);
p = arr2 + 3;
printf("%d\n", *p);
```

a=2

a non prevedibile

p va a puntare a arr2[3]

Puntatori Vs. Array I

```
int array[10];


for (int i=0; i<10;i++)
{
  array[i]= 0;

}
```

```
int array[10];
int *p;
p = array;
for (int i=0; i<10;i++)
{
   *p = 0;
   p++;
}
```

Puntatori Vs. Array II

```
char arr[]="bell";
int i=0;
while (arr[i] !='\0')
{
  if arr[i]=='e'
        arr[i]='a';
 i++
}
```

```
char arr[]="bell";
char *p=arr;

while(*p !='\0')
{
  if *p=='e'
          *p='a';

  p++;
}
```


Esercizio

```
#include <stdio.h>
int strlen(char *);

int main(){
char stringa[100];
     printf("Inserisci la stringa: ");
     scanf("%s", stringa);
     printf("la lunghezza della stringa: %d", strlen(stringa));

return 0;
}

//strlen: restituisce la lunghezza di una stringa
int strlen(char *s)
{
  int n;
  for (n=0; *s != '\0'; s++)
       n++;
  return n;
}
```


Passaggio di parametri a una funzione per riferimento

Chiamata per riferimento 1/2

- Chiamata per riferimento con puntatori come argomenti
  - Si passa l'indirizzo dell'argomento usando l'operatore **&**
  - Consente di modificare la reale locazione di memoria che contiene il valore passato dalla funzione chiamante.
- *Si noti che gli array non sono passati con* ***&*** *dato il nome dell'array è un puntatore (****arrayName*** *equivale a* ***&arrayName[0]****)*

Chiamata per riferimento 2/2

- Operator *
  - E' usato come alias/nickname (soprannome) per la variabile all'interno della funzione

```
void double( int *number )
 {
     *number = 2 * ( *number );
 }
```

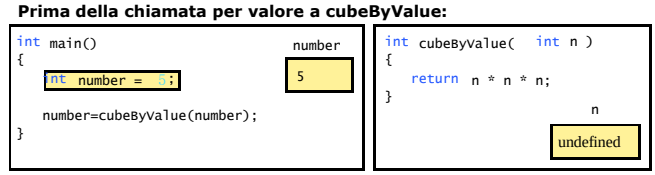
  - *number è usato come nickname per la variabile passata


```
/* Fig. 7.6: fig07_06.c
Calcolo del cubo di una variabile mediante chiamata per valore */
#include <stdio.h>

int cubeByValue( int ); /* prototype */

int main()
{
    int number = 5;

  printf( "The original value of number is %d", number );
  number = cubeByValue( number );
  printf( "\nThe new value of number is %d\n", number );

return 0;
}

   int cubeByValue( int n )
   {
      return n * n * n; /* calcolo del cubo: variabile locale n */
   }
```

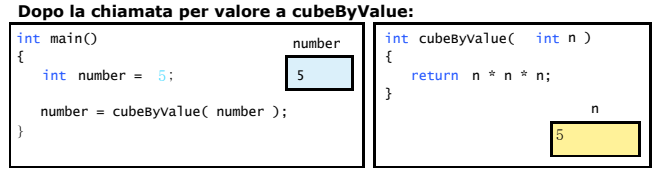
```
The original value of number is 5
The new value of number is 125
```


Passi di una chiamata per valore

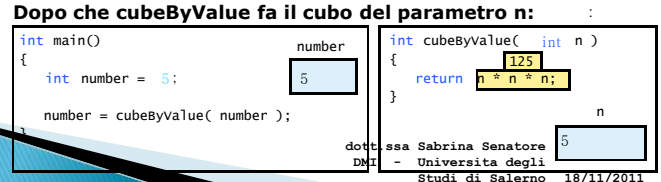
Prima della chiamata per valore a cubeByValue:



Dopo la chiamata per valore a cubeByValue:

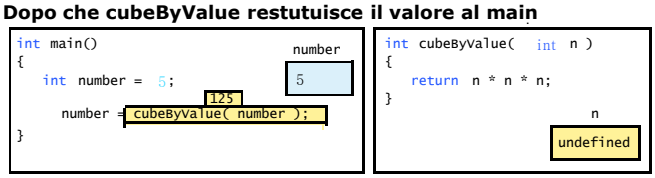


Dopo che cubeByValue fa il cubo del parametro n:

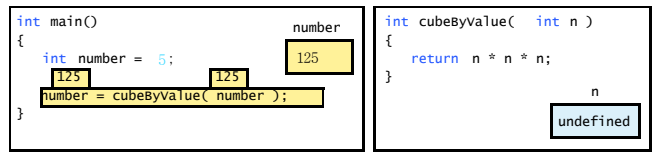


Passi di una chiamata per valore

Dopo che cubeByValue restutuisce il valore al main



Dopo che main completa l'assegnazione a number:



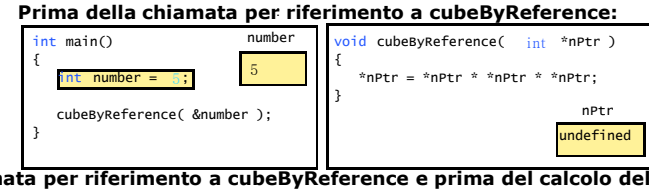
Il cubo di una variabile usando una chiamata per riferimento con un argomento puntatore */

```
#include <stdio.h>
void cubeByReference( int * ); /* prototype */
int main()
 {
 int number = 5;

 printf( ”Il valore originale di number è %d", number );
 cubeByReference( &number );
 printf( "Il nuovo valore di number è %d\n", number );
 return 0;
 }

 void cubeByReference( int *nPtr )
 {
 *nPtr = *nPtr * *nPtr * *nPtr; /* il cubo di number nel main */
 }
```

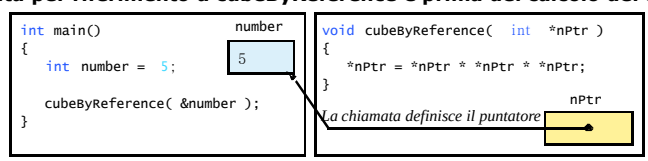
```
The original value of number is 5
The new value of number is 125
```


Passi di una chiamata per riferimento

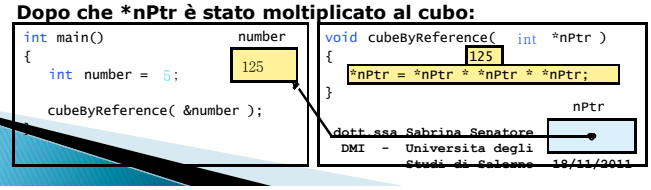
Prima della chiamata per riferimento a cubeByReference:



Dopo la chiamata per riferimento a cubeByReference e prima del calcolo del cubo di *nPtr:



Dopo che *nPtr è stato moltiplicato al cubo:



```
1 /* Fig. 7.21: fig07_21.cpp
2 Copiare una stringa usando la notazione con indici di vettore
3 e la notazione con puntatore. */
4 #include <stdio.h>
5
6 void copy1( char *, char * );
7 void copy2( char *, char * );
8
9 int main()
10 {
11 char string1[ 10 ], *string2 = "Hello",
12 string3[ 10 ], string4[] = "Good Bye";
13
14 copy1( string1, string2 );
15 printf( "string1 = %s\n", string1 );
16
17 copy2( string3, string4 );
18 printf( "string3 = %s\n", string3 );
19
20 return 0;
21 }
22
23 /* copia s2 in s1 usando la notazione con gli indici di vettore */
24 void copy1( char *s1, char *s2 )
25 {
26 int i;
27 for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ );
28  /* il corpo del for è vuoto */
29 }
30
31 /* copia s2 in s1 usando la notazione con i puntatori */
32 void copy2( char *s1, char *s2 )
33 {
34 for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ );
35 /* il corpo del for è vuoto */
36 }
```

Prototipi identici, eseguono lo stesso compito ma hanno implementazioni diverse

L'algoritmo di ordinamento Bubble Sort mediante chiamata per riferimento

- Implementazione bubblesort mediante puntatori
  - Scambio di due elementi
  - La funzione swap che riceve due indirizzi degli elementi dell'array.
- Pseudo-codice

  Initializzazione dell'array
  stampa degli elementi dell'ordine iniziale
  Chiamata della funzione bubblesort
  stampa dell'array ordinato

```
1    /*
2       This program puts values into an array, sorts the values into
3       ascending order, and prints the resulting array. */
4    #include <stdio.h>
5    #define SIZE 10
6    void swap( int *, int * );
7    void bubbleSort( int *array, const int size ); /* prototype */
8
9    int main()
10   {
11      /* initialize array a */
12      int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14      int i;
15
16      printf( "Data items in original order\n" );
17
18      /* loop through array a */
19      for ( i = 0; i < SIZE; i++ ) {
20         printf( "%4d", a[ i ] );
21      }
22
23      bubbleSort( a, SIZE ); /* sort the array */
24
25      printf( "\nData items in ascending order\n" );
26
```


```
27      /* loop through array a */
28      for ( i = 0; i < SIZE; i++ ) {
29         printf( "%d", a[ i ] );
30      }
31
32      printf( "\n" );
33
34      return 0; /* indicates successful termination */
35
36   } /* end main */
37
38   /* sort an array of integers using bubble sort algorithm */

39    void bubbleSort( int *array, const int size )
40   {
41
42      int pass; /* pass counter */
43      int j;    /* comparison counter */
44
45      /* loop to control passes */
46      for ( pass = 0; pass < size - 1; pass++ ) {
47
48         /* loop to control comparisons during each pass */
49         for ( j = 0; j < size - 1; j++ ) {
50
```


```
51            /* swap adjacent elements if they are out of order */
52            if ( array[ j ] > array[ j + 1 ] ) {
53               swap( &array[ j ], &array[ j + 1 ] );
54            } /* end if */
55
56         } /* end inner for */
57
58      } /* end outer for */
59
60   } /* end function bubbleSort */
61
62   /* swap values at memory locations to which element1Ptr and
63      element2Ptr point */
64   void swap( int *element1Ptr, int *element2Ptr )
65   {
66      int hold = *element1Ptr;
67      *element1Ptr = *element2Ptr;
68      *element2Ptr = hold;
69   } /* end function swap */
```


Programmazione I

Allocazione della memoria

I puntatori

- Un puntatore è un indirizzo di memoria
  - Accessibile mediante una variabile, di cui e' noto il tipo
- L'indirizzo è assegnato…
  - Considerando l'indirizzo di una variabile esistente
    - il puntatore punterà a quella variabile
  - Allocando (oppure riservando) della memoria libera
    - il puntatore punterà alla locazione di memoria riservata
    - Si ha l'allocazione *dinamica*

Assegnazione di un valore a un puntatore

- Tre sono i casi in cui è possibile assegnare valori a un puntatore:
  - a un **puntatore** é assegnato il valore **NULL** (non punta a "niente")

```
int *pi = NULL;
```

  - a un **puntatore** è assegnato l'**indirizzo** di una variabile esistente, restituito dall'operatore **&**

```
( Es.  int a;   int* p;   p=&a; )
```

  - é eseguita un'operazione di **allocazione dinamica della memoria**

Allocazione dinamica di memoria

Heap di memoria: spazio dedicato dall'ambiente di esecuzione al programma, all'atto del suo caricamento in memoria

• La allocazione dinamica è l'acquisizione di spazio di memoria dallo heap durante l'esecuzione del programma

• La libreria standard ANSI <stdlib.h> fornisce due funzioni fondamentali per la allocazione dinamica: malloc( ) e calloc( )

```
void *malloc(size_t nBytes);
void *calloc(int n, size_t nBytes);
```


Allocazione

- La keyword `malloc` sta per "memory allocation"

```
void* malloc(unsigned int n);
```

- alloca n bytes di memoria e restituisce un puntatore (cioè l' indirizzo) alla memoria appena allocata
- Il puntatore è generico → void, puntatore senza tipo, cioè, un semplice indirizzo di memoria

Allocazione di << vettori >>

```
(void*) calloc(unsigned int n, unsigned int size);
```

- calloc = contiguous allocation
  - Funziona come **malloc**
  - Alloca ***n*** elementi contigui ognuno grande ***size***.
  - In pratica, alloca un area di memoria grande ***n*** x ***size***
  - La memoria viene inizializzata a 0.

- Esempio:
- `int* p;`
- `p = (int*) calloc(1000,sizeof(int));`
  - Alloca un vettore di 1000 interi.

Allocazione dinamica di memoria

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   int  *a;                    /* da usare come ARRAY */
   int   n;                    /* lunghezza dell'array */
  . . . . .  /* l'utente inserisce da input il valore di n */
  a = (int *) malloc (n * sizeof(int)); /*crea spazio per
   l'array a */
  .....
/* utilizza a come array */
```

- N.B.: **sizeof** determina la dimensione in byte di un vettore, di un tipo di dato, ecc

Allocazione dinamica di memoria

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

int main(void)
{
   int  *a           /* da usare come ARRAY */
   int   n;           /* lunghezza dell'array */
  . . . . ./* l'utente inserisce da input il valore di n */
  a = (int *) calloc (n, sizeof(int)): /*crea spazio per
   l'array a */

  . . . ..    /* utilizza a come array */
```

- N.B.: **sizeof** determina la dimensione in byte di un vettore, di un tipo di dato, ecc

Allocazione dinamica di memoria

- Lo spazio di memoria allocato in maniera automatica viene anche liberato automaticamente, all'uscita dalla funzione
- Lo spazio allocato dinamicamente non viene deallocato all'uscita dalla funzione, ma deve essere liberato manualmente quando non è più necessario (non farlo potrebbe comportare un esaurimento prematuro nella memoria del sistema "memory leak")
- **`void free(void *p);`**

Esempio

```
int* p;
p = malloc( … );
```

E' corretta l'espressione?

- Errore di tipo nell'espressione: un (int*) e un (void*)
- Soluzione: type-casting

```
int* p;
p = (int*) malloc( … );
```


Sizeof

- Solitamente si usa **sizeof** in combinazione con la malloc.

sizeof(int)

- dice quanti byte sono usati per rappresentare convenientemente un `int` (solitamente 4)
- Si può usare con i tipi base e con i tipi "avanzati" int, short, float, double, struct...

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


Stack vs. Heap

- Stack è automatico

```
int main() {
   int x; //dichiarazione di un int sullo stack
   x = 5; //assegna 5 alla locazione di memoria
   return 0;
}
```

- Heap è usato per l'allocazione dinamica dello spazio di memoria

```
int main() {
   int *p = (int*) malloc(sizeof(int)); //allocaz. mem. dall'heap
   *p = 5;       // assegna 5 alla locazione di memoria
   return 0;
}
```


Se la memoria non è sufficiente…

- E' buon abitudine controllare se la memoria a disposizione è sufficiente (supponendo un certo numero di allocazioni già avvenute in precedenza).
- Se infatti non c'è sufficiente memoria disponibile, malloc restituisce **NULL**
  - **NULL** è un "puntatore che non punta a nulla", rappresentato dal valore 0

```
int* p, n=100;
p = (int*) malloc (n * sizeof(int));
if (p == NULL) {
  /* memoria non sufficiente ... */
}
```


Deallocazione

- Come visto per liberare la memoria allocata dopo l'utilizzo

```
void free(void* p);
```

- Esempio:

```
int* p;
p = (int*) malloc(sizeof(int));
... /* Utilizzo di p*/
free(p);
```


Allocazione di array

- Solitamente abbiamo creato un vettore:
  - `int a[10]; // la taglia è fissata inizialmente`
- Con l'allocazione dinamica:
  - `int* p;`
  - `p = (int*) calloc(10,sizeof(int));`
- oppure
  - `int* p;`
  - `p = (int*) malloc(10*sizeof(int));`

Esercizio

- Taglia dell'array determinata a run-time piuttosto che in fase di compilazione

```
#include <stdlib.h>
int main() {
   int n = rand() % 100;  //numero random tra 0 e 99
   int *array = (int *) malloc(n * sizeof(int));
   for(int i = 0; i < n; i++)
       array[i] = 0;

   free(array);
   array = NULL;
   return 0;
}
```


Allocazione/deallocazione: problemi 1/3

```
int nome_funzione ()
{
  int *p;
  /* allocazione di un vettore di 1000 interi */
  p = (int*)calloc( 1000, sizeof(int));
  /* utilizzo del vettore */
  ...
  /* fine della funzione */
  return 0;
}
```

- **Prima dell'uscita dalla funzione non c'è la deallocazione! (free(p);)**
- **Ogni volta che si usa questa funzione, si perdono (*leak*) 1000x4 bytes di memoria**

Allocazione/deallocazione: problemi 2/3

```
...
  int *p, i;
  /* allocazione di un vettore di 1000 interi */
  p = (int*) malloc( 1000 * sizeof(int));
  /* uso il vettore */
  ...
  /* p assume un nuovo indirizzo di memoria */
  p = &i;
```

- Una volta assegnato il puntatore a una nuova locazione, non c'è possibilità di recuperare la locazione di memoria precedentemente allocata → è impossibile deallocare!

Allocazione/deallocazione: problemi 3/3

```
...
  int *p;
  /* allocazione di un vettore di 1000 interi */
  p = (int*) calloc( 1000, sizeof(int));
  /* uso il vettore */
  ...
  /* allocazione di un altro vettore */
  p = (int*) malloc( 1000 * sizeof(int));
```

- Non c'è la deallocazione!

Esercizio

- Dati due array ordinati (si suppongano dati da programma, già ordinati), costruirne uno che è il merge ordinato dei due dati

Programmazione I

Strutture

Strutture 1/2

- Le Strutture
  - Insieme di variabili correlate (aggregati) da un unico nome
  - Possono contenere variabili di tipi di dato differenti
  - Comunemente usate per definire record da memorizzare nei file
  - Combinate con puntatori, possono servire a creare tipi di dati strutturati come liste a puntatori, pile, code ed alberi
- Le strutture sono un tipo di dato derivato.

Strutture 2/2

- Sono utili per organizzare come un unico oggetto un insieme di variabili correlate.
- Esempio:

Etichetta di una struttura

```
struct point{
  int x;
  int y;
};
```

Membri di una struttura

- Una dichiarazione `struct` definisce un tipo.

Esempio 1/3

- La definizione della struttura **impiegato**:

```
struct impiegato{
  char nome[20];
  char cognome[20];
  int eta;
  char sesso;
  float stipendio;
};
```


Esempio 2/3

- La definizione della struttura **libro**:

```
struct libro {
   char ISBN[10];
   char titolo[40];
   char autore[25];
   char editore[25];
   int anno;
   float prezzo;
   };
```


Esempio 3/3

- La definizione della struttura **`card`**:

```
struct card {
   char *face;
   char *suit;
};
```

- `card` è il nome della struttura e contiene due elementi di tipo `char *`

Definizione di struttura e istanze

- La definizione di strutture definisce una strutture ma concretamente NON è creata alcuna istanza della struttura definita.
- In altre parole, le istruzioni precedenti non dichiarano, assegnando spazio di memoria ad alcuna struttura.
- Dopo la definizione abbiamo solo detto "come è fatta" la struttura, ma non abbiamo dichiarata una variabile del tipo della struttura da poter utilizzare.
- *La sola dichiarazione di struttura non riserva alcun area di memoria: descrive unicamente la struttura*

Dichiarazione di una struttura 1/2

- Esistono due modi per dichiarare una struttura:
  - La definizione è seguita da uno o più nomi di variabili:

```
struct point{
  int x;
  int y;
 } a, b, c;
```

  - Dichiarare le variabili istanza della struttura in un punto differente del codice, separato dalla definizione della struttura stessa.

```
struct point a, b, c;
```


Dichiarazione di una struttura 2/2

- Esempi:

```
struct libro {
  char ISBN[10];
  char titolo[40];
  char autore[25];
  char editore[25];
  int anno;
  float prezzo;
  } Libro;
```

```
struct impiegato{
  char nome[20];
  char cognome[20];
  int eta;
  char sesso;
  float stipendio;
}imp, impiegati[20];
```


Inizializzazione di una struttura 1/2

- Una struttura può essere inizializzata facendo seguire la sua definizione da una lista di inizializzatori

```
struct point maxpt ={300, 220};
```

- Oppure potrebbe essere inizializzata mediante il costrutto del tipo: *nome-struttura.membro*:

```
struct point pt;
pt.x=10;
pt.y=10;
```

- L'operatore di membro di una struttura "." connette il nome della struttura con quello del membro della struttura.

```
printf("%d", maxpt.y);
```


Inizializzazione di una struttura 2/2

- Esempio:

```
struct libro Libro ={"8870564436", "Linguaggio C",
  "B.K. Kernighan, D.M. Ritchie", "Jackson", 1990,
  35.4};
```


Accesso ai membri delle strutture

- Operatore membro di struttura (.)
  - Detto anche operatore punto

```
struct card myCard;
printf( "%s", myCard.suit );
```

- Operatore puntatore a struttura (->)
  - Detto anche operatore freccia
  - Accede ai membri della struttura attraverso un puntatore alla stessa

```
struct card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

- Le seguenti espressioni sono equivalenti:

```
myCardPtr->suit
( *myCardPtr ).suit
```


Esempio

```
#include <stdio.h>
 struct card {
   char *face; /* define pointer face */
   char *suit; /* define pointer suit */
 };

int main()
{
 struct card a;      /* define struct a */
 struct card *aPtr; /* define a pointer to card */
 a.face = "Ace";
 a.suit = "Spades";
 aPtr = &a; /* assign address of a to aPtr */
 printf( "%s%s%s\n%s%s%s\n%s%s%s\n",
          a.face, " of ", a.suit,
          aPtr->face, " of ", aPtr->suit,
         ( *aPtr ).face, " of ", ( *aPtr ).suit );
 return 0;
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```


Typedef 1/2

- Fornisce un meccanismo per creare sinonimi per tipi di dati definiti in precedenza.
- typedef consente abbreviazioni di nomi di tipi

- Esempio:

```
typedef int integer;
```

  - crea il sinonimo integer del tipo int

```
typedef struct Card *CardPtr;
```

  - definisce un nuovo `CardPtr` come sinonimo per il tipo `struct Card *`

- typedef non crea un nuovo tipo, piuttosto un alias.

Typedef 1/2

- Spesso si usa typedef per definire un tipo di struttura in modo da evitare l'uso dell'etichetta della struttura.

- Esempio:

```
typedef struct {
   char *face;
   char  *suit;
} Card;
```

- Creerà il tipo di struttura Card, senza la necessità di un'istruzione typedef separata

Definire un tipo di record

- Sintassi della dichiarazione:

typedef struct *nome_tipo* {
Dichiarazione dei campi
} *nome_tipo*;

```
typedef struct point{
   int x;
    int y;
 } point;
```


Dichiarare variabili di un tipo di record 1/2

```
 typedef struct studente {
  int matricola;
  char nome[25];
  char indirizzo[25];
 } studente;

 int main (void)
 {
//inizizializzazione
  studente std1={123456, "Paolo Rossi", "via della
   Repubblica"};
  ........

...
}
```


Dichiarare variabili di un tipo di record 2/2

```
 typedef struct studente {
  int matricola;
  char nome[25];
  char indirizzo[25];
 } studente;

 int main (void)
 {

//oppure inserimento da tastiera:
  studente st;
  printf("nome studente:\n");
  scanf("%s", st.nome);
  printf("matricola:\n");
  scanf("%d", &st.matricola);
  printf("indirizzo:\n");
  scanf("%s", st.indirizzo);
}
```


Selezione dei campi 2/2

```
typedef struct libro {
 char ISBN[10];
 char titolo[40];
 char autore[25];
 char editore[25];
 int anno;
 float prezzo;
 } Libro;
```

```
Libro *l, paper;
l = &paper;
```

Come accedere all'elemento ISBN?

```
(*l).ISBN
```

N.B. Le parentesi intorno al puntatore sono necessarie perché l'operatore punto ha precedenza sull'operatore asterisco

Strutture annidate

- Le strutture possono essere nidificate l'una nell'altra

```
typedef struct studente
 {
  char *nome;
  int matricola;
  Indirizzo ind;
 } Studente;
```

```
typedef struct indirizzo
 {
  char *via;
  char *stato;
  int zipCode;
 } Indirizzo;
```


Come accedere a strutture annidate

```
Studente st;
st.nome= "Paolo Rossi";
…
st.ind.via="Corso Umberto I";
…
st.ind.zipCode=84084;
```


Passaggio di record come parametri: esercizio 1/2

```
#include <stdio.h>
# include <math.h>
typedef struct point {
 int x;
 int y;
} punto;
punto makePoint (int, int);
double computeDistance (punto, punto);

int main (void)
{
 double distanza;
 punto origine, centro;
 origine = makePoint(0,0);
 centro =  makePoint(4, 5);
 distanza = computeDistance (origine, centro);
return 0;
}
```


Passaggio di record come parametri: esercizio 2/2

```
punto makePoint (int x, int y)
{
  punto p;
  p.x = x;
  p.y = y;
  return p;
}

double computeDistance (punto p1, punto p2)
{
double d, dx, dy;
dx = p1.x - p2.x;
dy = p1.y - p2.y;
d = sqrt((double)(dx*dx + dy*dy));
return d;
}
```

Array di strutture

- Se volessi definire una “biblioteca”, cioè un insieme di libri?

```
typedef struct libro {
 char ISBN[10];
 char titolo[40];
 char autore[25];
 char editore[25];
 int anno;
 float prezzo;
 } Libro;
;
```

```
Libro biblioteca[100];
```