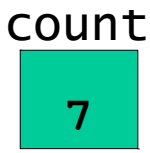
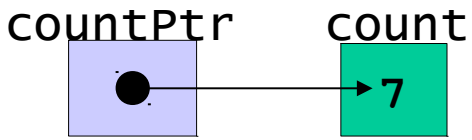
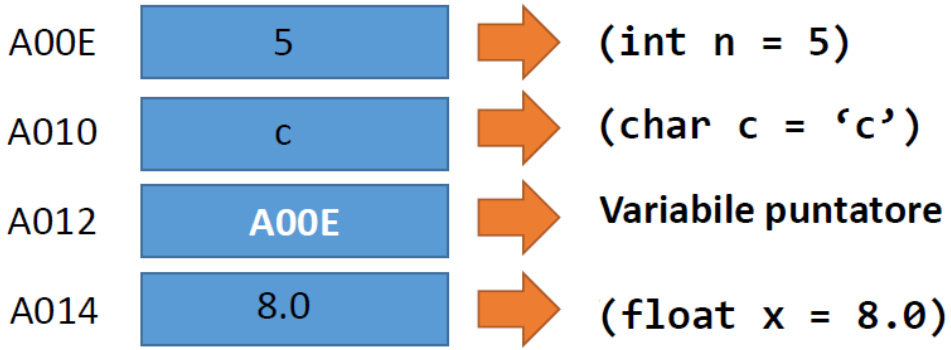
Linguaggio C: Variabili Puntatori

Generalità

- Comunemente ad altre variabili, i puntatori sono caratterizzati da tipo, nome, locazione di memoria.
- Il tipo di dato si riferisce agli indirizzi di memoria usati da un compilatore C
- Memorizzano indirizzi di memoria di variabili aventi valori riferiti ad un tipo di dato specifico







Dichiarazione

- `Type * name_variable`

  ```
  int *myPtr;
  ```

  definisce un puntatore ad un intero.
- La variabile myPtr deve contenere l'indirizzo di una variabile a valore intero, ma questo non è verificato automaticamente, perciò è lo sviluppatore che deve controllare che il valore contenuto sia coerente col tipo. Il compilatore non controlla ed eventuali incongruenze sono evidenti sono in esecuzione.
- Per forzare un puntatore di un tipo A a prendere un valore di un tipo B si usa l'operatore *cast*.
- Possibili valori sono 0, NULL o un indirizzo.

Inizializzazione

- Quando non inizializzati, il compilatore assegna un valore “random”, che punterà ad una locazione di memoria random.
- Questo può portare a deframmentare la memoria, rendendola ingestibile con possibili incongruenze durante la esecuzione.

```
int *myPtr; int my=5;
my= *myPtr;
*myptr= 100;
```

Quale sarà il valore della variabile dato my? Lo sviluppatore è consapevole di questo?
Meglio inizializzare *myPtr a NULL!

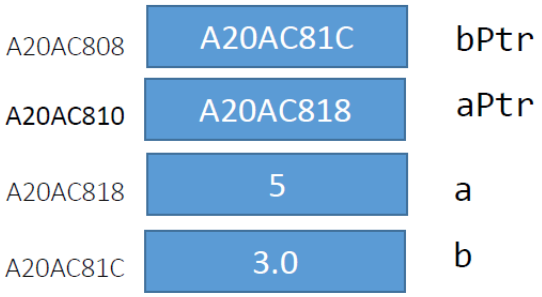
Operatori

- * operatore di dereferenziazione, usato nella dichiarazione per definire un puntatore
- & operatore di referenziazione, usato nella elaborazione
- & restituisce la locazione di memoria di un dato

```c
#include <stdio.h>
int main() {
  int a = 5;
  float b = 3.0;

  int* aPtr; // puntatore a intero
  float* bPtr; // puntatore a float

  aPtr = a; // errore
  aPtr = &a; // ok
  bPtr = &b; // ok

  printf("\n a: %d \t\t &a: %X", a, aPtr);
  printf("\n b: %.2f \t &b: %X", b, bPtr);
}
```

Operatori

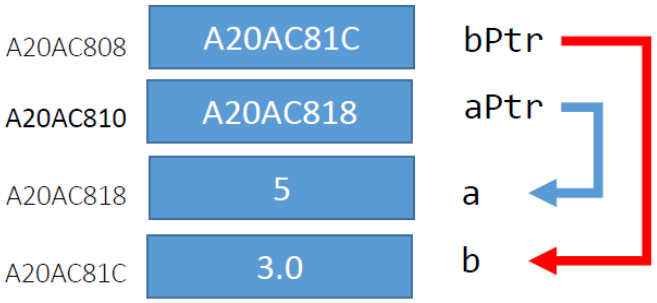
- * usato nella dichiarazione per definire un puntatore
- & usato nella elaborazione
- & restituisce la locazione di memoria di un dato




```c
#include <stdio.h>
int main() {
  int a = 5;
  float b = 3.0;

  int* aPtr; // puntatore a intero
  float* bPtr; // puntatore a float

  aPtr = a; // errore
  aPtr = &a; // ok
  bPtr = &b; // ok

  printf("\n a: %d \t\t &a: %X", a, aPtr);
  printf("\n b: %.2f \t &b: %X", b, bPtr);
}
```

Operatori

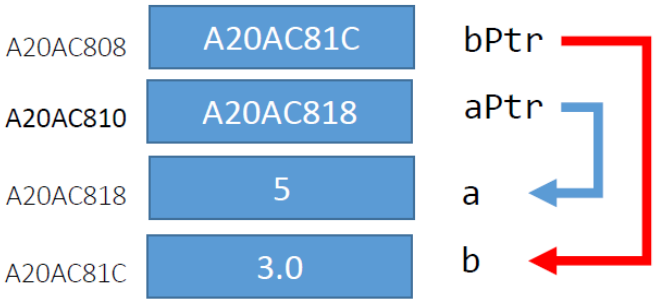
- * usato nella dichiarazione per definire un puntatore
- & usato nella elaborazione
- & restituisce la locazione di memoria di un dato




```c
#include <stdio.h>
int main() {
  int a = 5;
  float b = 3.0;

  int* aPtr; // puntatore a intero
  float* bPtr; // puntatore a float

  aPtr = a; // errore
  aPtr = &a; // ok
  bPtr = &b; // ok

  printf("\n a: %d \t\t &a: %X", a, aPtr);
  printf("\n b: %.2f \t &b: %X", b, bPtr);
}
```

Operatori

- * usato nella dichiarazione per definire un puntatore
- & usato nella elaborazione
- & restituisce la locazione di memoria di un dato

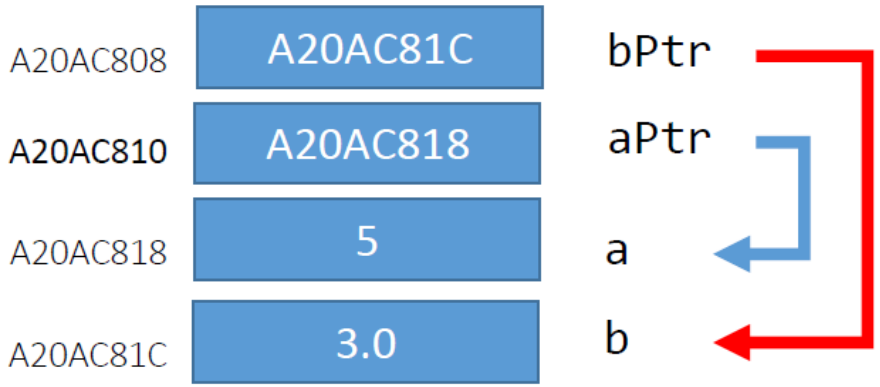



```c
#include <stdio.h>
int main() {
  int a = 5;
  float b = 3.0;

  int* aPtr; // puntatore a intero
  float* bPtr; // puntatore a float

  aPtr = a; // errore
  aPtr = &a; // ok
  bPtr = &b; // ok

  printf("\n a: %d \t\t &a: %X", a, aPtr);
  printf("\n b: %.2f \t &b: %X", b, bPtr);
}
```

```
a: 5            &a: A20AC81C
b: 3.00         &b: A20AC818
```

Operatori

- * usato anche nella elaborazione, restituisce il contenuto della locazione puntata



- *aPtr è un alias/sinonimo della variabile-dato e restituirà valore 5

```
int c=*aPtr;
```

Operatori

- & restituisce la locazione di memoria di un dato, ad esempio per la acquisizione da tastiera con *scanf()*:

  scanf("%d %d, **&n, &m**")

  Nell'esempio, la funzione ottiene l'indirizzo delle due variabili per memorizzarne il valore:

Operatori

Ricapitolando:

- *a* variabile dato
- *&a* indirizzo della variabile
- *type *p* variabile puntatore
- *p* indirizzo (di una variabile dato) a cui punta la variabile puntatore p
- **p* valore della variabile dato cui punta la variabile puntatore p
- *& *p* indirizzo della variabile dato cui punta la variabile puntatore p
- **&a* valore della variabile dato cui riferisce l'indirizzo &a

Operatori

```c
/* Fig. 7.4: fig07_04.c
   Using the & and * operators */
#include <stdio.h>

int main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

Indirizzo di a è il valore di aPtr.

* e & agiscono in maniera complementare rispetto ad una locazione di memoria

Operatori

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Usi dei puntatori: passaggio di parametri

- Nell'uso di funzioni, spesso potremmo aver bisogno di applicare le azioni delle funzioni direttamente sugli argomenti, senza la necessità di doverli copiare.
- Diversamente, usare il passaggio per copia, richiederebbe maggior uso di **memoria** e **istruzioni** suppletive per propagare le azioni. Questo in sistemi complessi richiede costi maggiori.

```
main() {
  int x = 33, y = 5;
  scambia(&x, &y);
  printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a, int* b) {
  int t; // variabile locale di appoggio

  t = *a; // scambio dei valori
  *a = *b;
  *b = t;
```

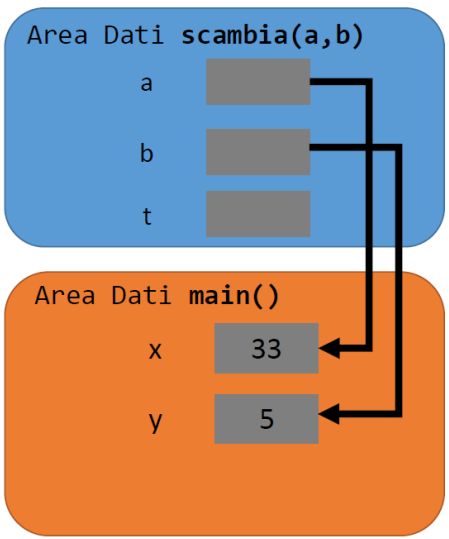
Usi dei puntatori: passaggio di parametri

- E' lo scenario tipico in casi di modularizzazione in cui dati e funzioni sono definiti in differenti moduli.
- La invocazione delle funzioni prevederà la specifica dei **riferimenti** alle variabili (sintassi degli indirizzi), mentre il prototipo avrà puntatori come parametri formali (sintassi dei puntatori):

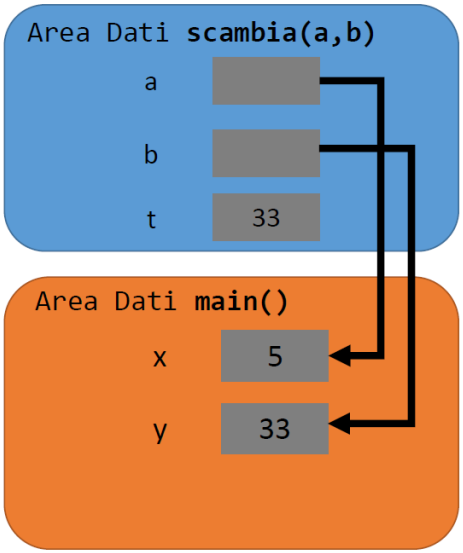
```
main() {
  int x = 33, y = 5;
  scambia(&x, &y);
  printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a, int* b) {
  int t; // variabile locale di appoggio

  t = *a; // scambio dei valori
  *a = *b;
  *b = t;
```

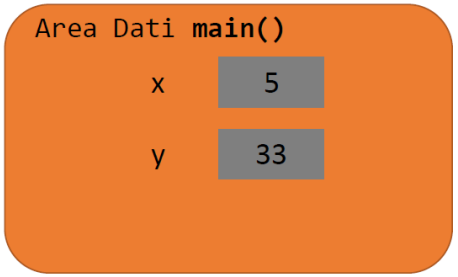
Usi dei puntatori: passaggio di parametri



1



2



3

```
main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio

   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
```

Usi dei puntatori: passaggio di parametri

Ancora un esempio:

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* prototype */

int main()
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass address of number to cubeByReference */
   cubeByReference( &number );

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

   return 0; /* indicates successful termination */

} /* end main */

/* calculate cube of *nPtr; modifies variable number in main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
} /* end function cubeByReference */
```

Usi dei puntatori: passaggio di parametri

Ancora un esempio:

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* prototype */

int main()
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass address of number to cubeByReference */
   cubeByReference( &number );

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

   return 0; /* indicates successful termination */

} /* end main */

/* calculate cube of *nPtr; modifies variable number in main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
} /* end function cubeByReference */
```

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

Usi dei puntatori: passaggio di parametri

- Per definire un puntatore possono essere usati tipi primitivi (built-in) e tipi strutturati (user-defined)
- Il vantaggio di usare puntatori di *struct* è quello di evitare la copia di una valore strutturato di elevate dimensioni nel passaggio di parametri per valore.
- Ad esempio:

```
5   #include <stdio.h>
6
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

La dimensione di una variable *data* sarebbe 24 byte (4+15+4+1 padding)

Usi dei puntatori: passaggio di parametri

```
5   #include <stdio.h>
6
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

```
// Prototipi di funzione
void function_data (data d);
void function_data_ptr (data* dPtr);

void function_data (data d) {
. . .
}
void function_data_ptr (data* dPtr) {
. . .
}

// main
int main() {
        function_data(d1);
        function_data_ptr(&d2);
}
```

- Le funzioni *function_data()* e *function_data_prt()* svolgono lo stesso compito, ma acquisiscono gli argomenti con differenti meccanismi.
- La differenza è che *function_data_prt()* non ha bisogno di copiare la data. Questo vantaggio deve essere controbilanciato con la possibilità di modificare i campi della struct.

Usi dei puntatori: passaggio di parametri

```
5   #include <stdio.h>
6
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

```
// Prototipi di funzione
void function_data (data d);
void function_data_ptr (data* dPtr);

void function_data (data d) {
. . .
}
void function_data_ptr (data* dPtr) {
. . .
}

// main
int main() {
        function_data(d1);
        function_data_ptr(&d2);
}
```

- Di fatti, passando un puntatore, se ne può modificare il contenuto.
- Se si volesse rendere non modificabile la data, si può ricorrere al qualificatore *const*.
- Perciò, il prototipo della funzione cambierebbe in

```
void function_data_ptr (const data* dPtr)
```

- Qualsiasi tentativo di modifica sarebbe rilevato dal compilatore

Usi dei puntatori: accesso ad un array

- Il nome di un array è il puntatore al primo elemento

```
int array[10]
int * p_array

p_array=&array[0];
p_array=array;                  // equivalenti
```

Usi dei puntatori: accesso ad un array

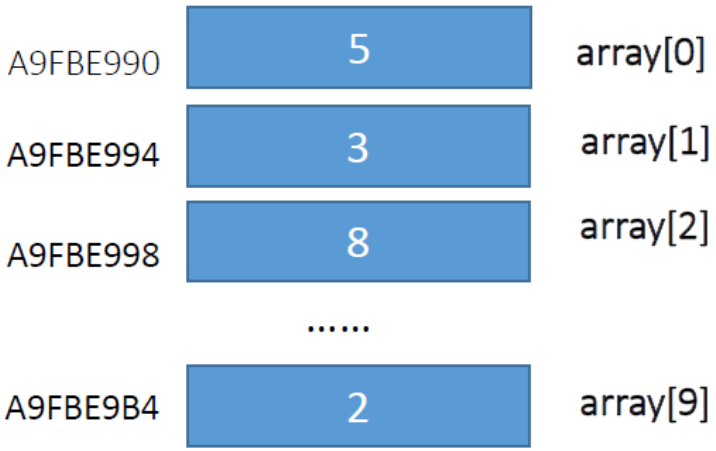
```c
#include <stdio.h>

int main() {
  int array[10]; // dichiaro un array di interi
  int* p_array; // dichiaro un puntatore a un intero

  p_array = &array[0];

  printf("Nome dell'array: \t\t%X \n", array);
  printf("Indirizzo del primo elemento: \t%X \n", &array[0]);
  printf("Puntatore all'array: \t\t%X \n", p_array);
}
```

il cui output potrebbe essere:

```
Nome dell'array:                A9FBE990
Indirizzo del primo elemento:   A9FBE990
Puntatore all'array:            A9FBE990
```

Usi dei puntatori: accesso ad un array

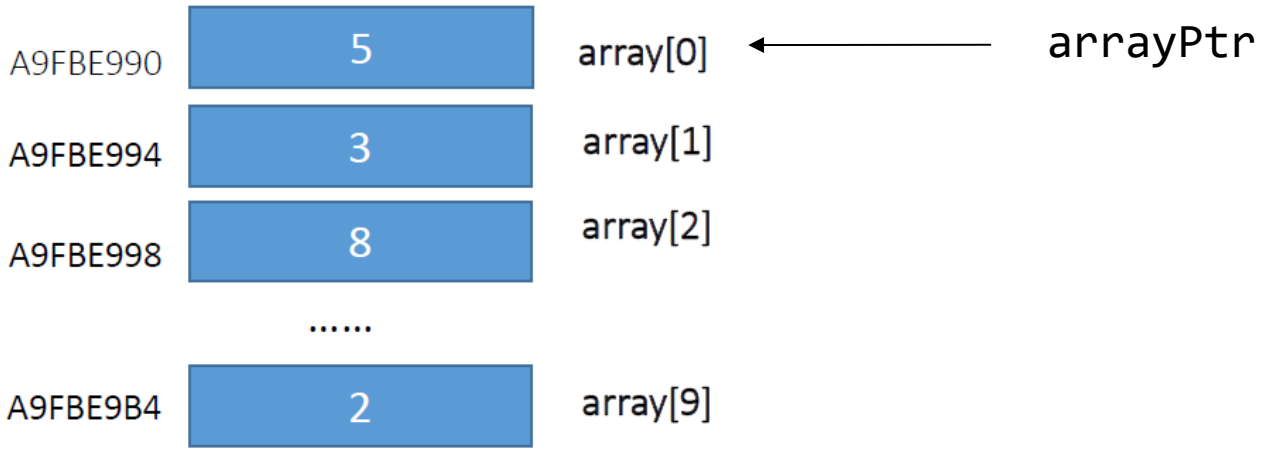
- Ai successivi elementi di un array si accede agendo sul puntatore definito applicandovi **operatori aritmetici.**
- Questo meccanismo prende il nome di Aritmetica dei puntatori.
- Tradizionalmente, l'accesso prevedrebbe espressioni aritmetiche sull'indice, mentre, con i puntatori, le espressioni aritmetiche sono definite con le variabili puntatori.



Usi dei puntatori: accesso ad un array

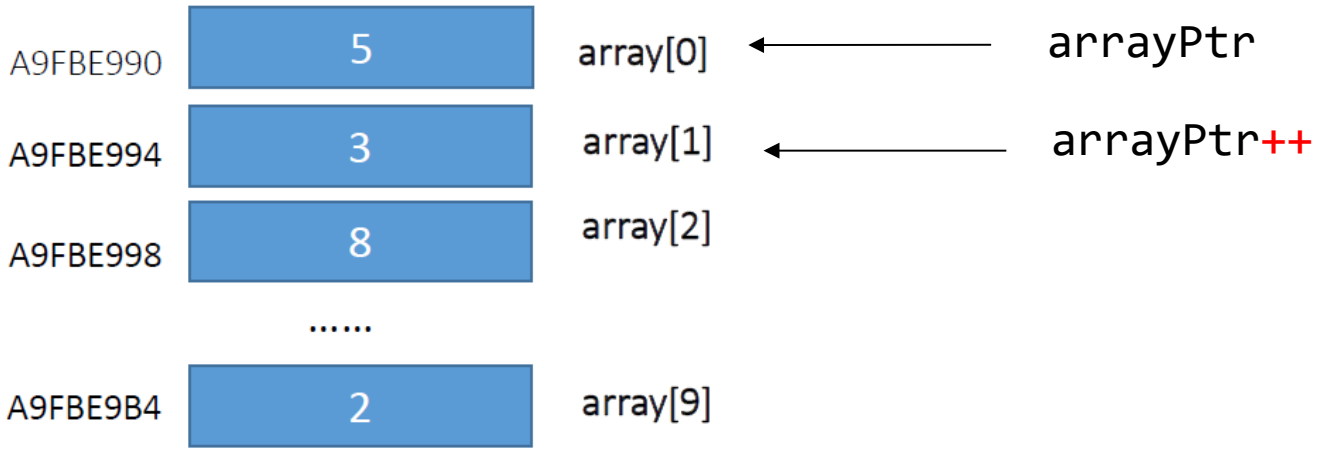
- Ai successivi elementi di un array si accede agendo sul puntatore definito applicandovi **operatori aritmetici.**
- Tradizionalmente, l'accesso prevedrebbe espressioni aritmetiche sull'indice, mentre, con i puntatori, le espressioni aritmetiche sono definite con le variabili puntatori.

```
int* arrayPtr= &array[0];
```



Usi dei puntatori: accesso ad un array

- Quindi, possiamo usare gli operatori aritmetici di incremento:
  - `arrayPtr++` punterà all'elemento successivo
  - `arrayPtr+=2` punterà all'elemento prossimo al successivo
  - `arrayPtr+i` punterà all'i-esimo elemento

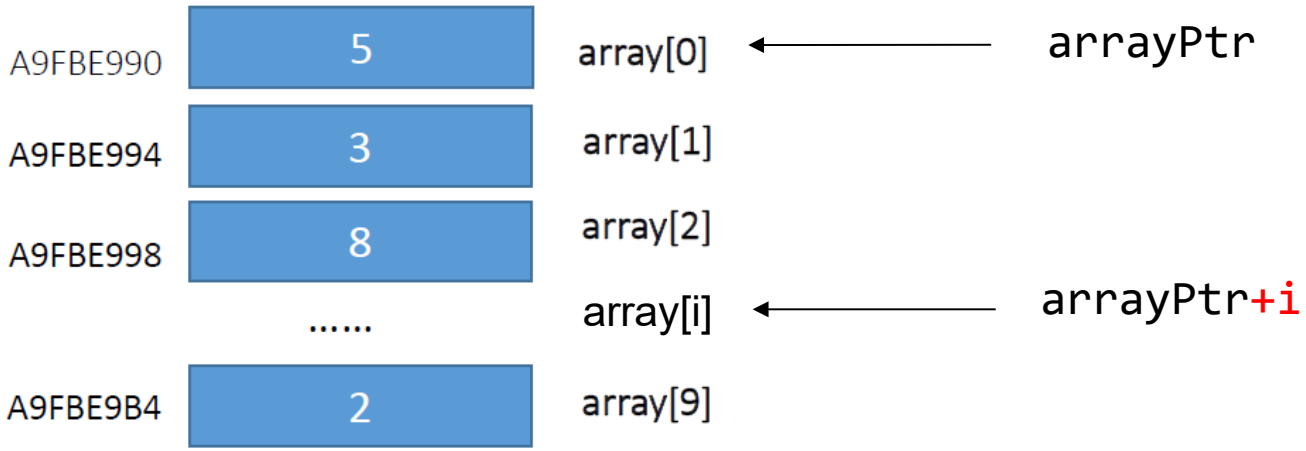


Usi dei puntatori: accesso ad un array

- Una volta calcolato l'indirizzo potremmo **accedere** al valore memorizzato con l'operatore *

```
*(arrayPtr+i)
```

che restituisce lo stesso dato di array[i]



Usi dei puntatori: accesso ad un array

- Esempio

```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b;                /* set bPtr to point to array b */
10    int i;                        /* counter */
11    int offset;                   /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
24
```

Usi dei puntatori: accesso ad un array

- Esempio

```
25    /* loop through array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* end for */
29
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termination */
47
48 } /* end main */
```

Usi dei puntatori: accesso ad un array

- Esempio

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Usi dei puntatori: accesso ad un array

- Quando gli array sono usati come parametri nelle funzioni, vi passiamo il nome dell'array, cioè l'indirizzo al primo elemento, quindi gli array sono passati per riferimento.
- Sintatticamente, abbiamo due modalità

```
int sum(int v[], int n)
```

esplicita che si tratta di un array. Più leggibile

```
int sum(int *v, int n)
```

non è esplicito che si tratta di un array. Indicato con aritmetica dei puntatori.

Puntatori a funzioni

- Puntatori a funzioni denotano l'area di memoria in cui è conservata la funzione.
- Come per gli array, il nome della funzione è il primo indirizzo dell'area.
- Puntatori a funzioni usati per passaggio di parametri, memorizzati in un array ed assegnate ad altri puntatori.

Puntatori a funzioni

Esempio:

```
1  /* Fig. 7.26: fig07_26.c
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* prototypes */
7  void bubble( int work[], const int size, int (*compare)( int a, int b ) );
8  int ascending( int a, int b );
9  int descending( int a, int b );
10
11 int main()
12 {
13    int order;   /* 1 for ascending order or 2 for descending order */
14    int counter; /* counter */
15
16    /* initialize array a */
17    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    printf( "Enter 1 to sort in ascending order,\n"
20            "Enter 2 to sort in descending order: " );
21    scanf( "%d",  &order );
22
23    printf( "\nData items in original order\n" );
24
```

Puntatori a funzioni

Esempio:

```
25    /* output original array */
26    for ( counter = 0; counter < SIZE; counter++ ) {
27       printf( "%5d", a[ counter ] );
28    } /* end for */
29
30    /* sort array in ascending order; pass function ascending as an
31      argument to specify ascending sorting order */
32    if ( order == 1 ) {
33       bubble( a, SIZE, ascending );
34       printf( "\nData items in ascending order\n" );
35    } /* end if */
36    else { /* pass function descending */
37       bubble( a, SIZE, descending );
38       printf( "\nData items in descending order\n" );
39    } /* end else */
40
41    /* output sorted array */
42    for ( counter = 0; counter < SIZE; counter++ ) {
43       printf( "%5d", a[ counter ] );
44    } /* end for */
45
46    printf( "\n" );
47
48    return 0; /* indicates successful termination */
49
50 } /* end main */
51
```

Puntatori a funzioni

Esempio:

```
52 /* multipurpose bubble sort; parameter compare is a pointer to
53    the comparison function that determines sorting order */
54 void bubble( int work[], const int size, int (*compare)( int a, int b ) )
55 {
56    int pass;  /* pass counter */
57    int count; /* comparison counter */
58
59    void swap( int *element1Ptr, int *element2ptr ); /* prototype */
60
61    /* loop to control passes */
62    for ( pass = 1; pass < size; pass++ ) {
63
64       /* loop to control number of comparisons per pass */
65       for ( count = 0; count < size - 1; count++ ) {
66
67          /* if adjacent elements are out of order, swap them */
68          if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
69             swap( &work[ count ], &work[ count + 1 ] );
70          } /* end if */
71
72       } /* end for */
73
74    } /* end for */
75
76 } /* end function bubble */
77
```

Puntatori a funzioni

Esempio:

```
78 /* swap values at memory locations to which element1Ptr and
79    element2Ptr point */
80 void swap( int *element1Ptr, int *element2Ptr )
81 {
82    int hold; /* temporary holding variable */
83
84    hold = *element1Ptr;
85    *element1Ptr = *element2Ptr;
86    *element2Ptr = hold;
87 } /* end function swap */
88
89 /* determine whether elements are out of order for an ascending
90    order sort */
91 int ascending( int a, int b )
92 {
93    return b < a; /* swap if b is less than a */
94
95 } /* end function ascending */
96
97 /* determine whether elements are out of order for a descending
98    order sort */
99 int descending( int a, int b )
100 {
101    return b > a; /* swap if b is greater than a */
102
103 } /* end function descending */
```

Puntatori a funzioni

Esempio:

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Puntatori a funzioni

- Si noti che la sintassi

```
int ( *compare )( int a, int b )
```

non deve essere confusa con

```
int *compare( int a, int b )
```

che definisce una funzione che restituisce un puntatore ad int.

Cittadini di prima classe

Ricordiamo: In C, ed in tutti i linguaggi, possiamo avere, **cittadini** di

- **terza classe**, se possono essere solo **chiamate** (invocazione)
- **seconda classe**, se possono essere passate come **argomenti**
- **prima classe**, se, oltre agli usi di terza e seconda classe, possono essere anche restituite come **risultato della chiamata** di altre funzioni o possono essere assegnate come **valore** a una variabile

Quindi, in C le funzioni sono considerate cittadini di **seconda classe** quando usate nel passaggio di parametri con puntatori.

Esercizi

Riscrivere la libreria Matematica che include gli operatori per il calcolo del 1-minimo tra due interi, 2-massimo tra due interi, 3-media tra due interi, 4-valore assoluto di un numero, in modo che

- gli operatori lavorino su argomenti ottenuti tramite passaggio di riferimento.
- il risultato di ogni calcolo sia il valore restituito dell'operatore

Esercizi

Riscrivere la libreria Matematica che include gli operatori per il calcolo del 1-minimo tra due interi, 2-massimo tra due interi, 3-media tra due interi, 4-valore assoluto di un numero, in modo che

- gli operatori lavorino su argomenti ottenuti tramite passaggio di riferimento.
- il risultato di ogni calcolo sia consegnato usando passaggio di parametri per riferimento.

Esercizi

Scrivere un programma modularizzato che, usando puntatori a funzioni, calcoli il massimo o minimo tra due numeri. La scelta della operazione è affidata all'utente.