Programmazione I

Corso di Laurea in Informatica
a.a. 2018-2019

Dr. Gabriele Tolomei
Dipartimento di Matematica
Università degli Studi di Padova
gtolomei@math.unipd.it

UNIVERSITÀ DEGLI STUDI DI PADOVA

13 Marzo 2019

Facciamo un passo indietro…

- Una locazione di memoria (RAM) possiede un indirizzo (univoco) a cui corrisponde un contenuto
- L'indirizzo di memoria è un numero (spesso espresso in notazione esadecimale)
- Solitamente, ciascun indirizzo si riferisce ad una locazione che contiene **8 bit** (**1 byte**)
- È compito del programmatore "interpretare" sequenze binarie in corrispondenti valori (ad es., interi, reali, caratteri)

Facciamo un passo indietro…

- Scrivere programmi che utilizzino esplicitamente gli indirizzi di memoria numerici sarebbe complicato!
- I linguaggi di programmazione ad alto livello (come il C) hanno introdotto il concetto di **variabile**
- Una variabile non è altro che un **identificatore** (nome) associato ad una certa locazione di memoria in cui può essere memorizzato un **valore** di un certo **tipo**
- In questo modo, è più facile per il programmatore riferirsi ad indirizzi di memoria

Facciamo un passo indietro…

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00			
90000001	00	sum	int (4 bytes)	000000FF(255_{10})
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

fonte: https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html

Puntatori

- Un puntatore è una variabile che, come altre, è associata ad un tipo e contiene un valore
- A differenza di altre variabili (**`int`**, **`double`**, **`char`**) il valore contenuto da un puntatore è un indirizzo di memoria
- Come ogni altra variabile, anche i puntatori devono essere dichiarati prima di poter essere usati

Puntatori: Dichiarazione

In generale

```
type *ptr;   // Declare a pointer variable called ptr as a pointer of type
// or
type* ptr;
// or
type * ptr;  // I shall adopt this convention
```

Due esempi concreti

```
int * iPtr;     // Declare a pointer variable called iPtr pointing to an int (an int pointer)
                // It contains an address. That address holds an int value.
double * dPtr;  // Declare a double pointer
```

* si applica *solo* all'identificatore che segue

```
int *p1, *p2, i;    // p1 and p2 are int pointers. i is an int
int* p1, p2, i;     // p1 is a int pointer, p2 and i are int
int * p1, * p2, i;  // p1 and p2 are int pointers, i is an int
```

Puntatori: Inizializzazione

- In seguito alla dichiarazione di una variabile puntatore il suo contenuto **non** è inizializzato
- Più precisamente, il puntatore contiene l'indirizzo di una locazione di memoria non meglio specificato e **non** valido
- **ATTENZIONE!** Dovete sempre inizializzare un puntatore assegnandogli un indirizzo di memoria valido

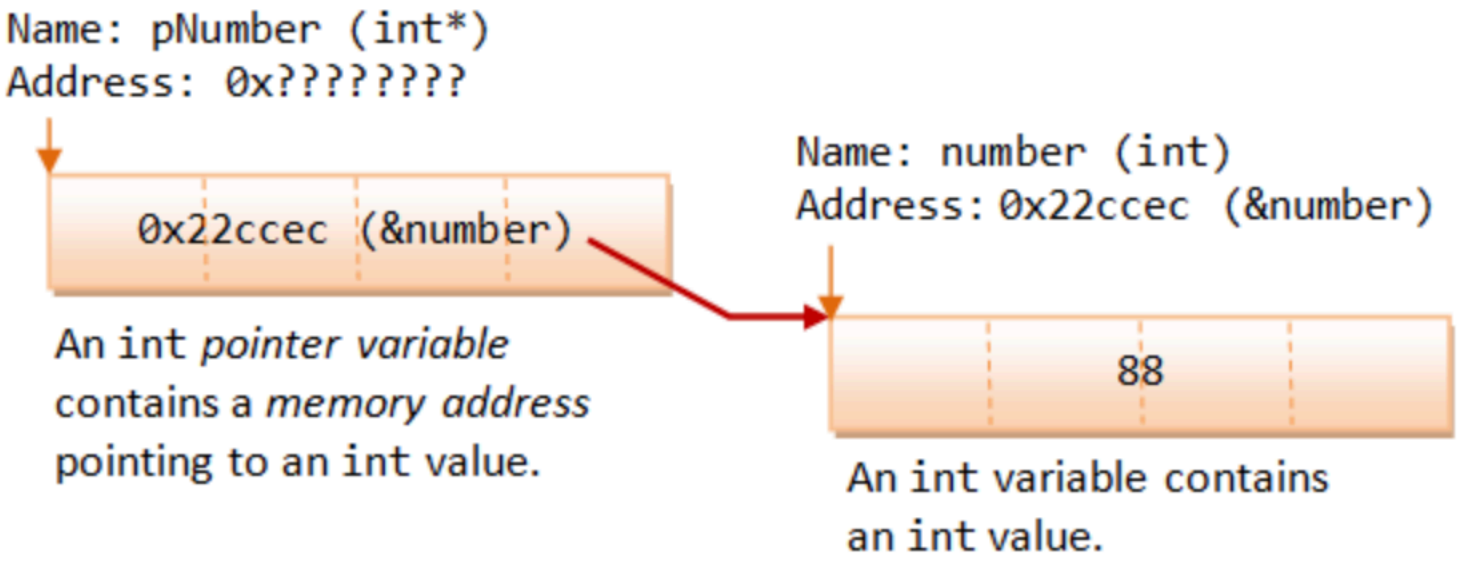
Puntatori: Inizializzazione

- L'assegnamento di un puntatore viene effettuato con l'operatore "address-of"(`&`).
- Questo operatore si applica ad una variabile e ne ritorna l'indirizzo
- Esempio: se `num` è una variabile `int`, `&num` restituisce l'indirizzo del valore identificato da `num`
- È possibile usare l'operatore `&` per assegnare l'indirizzo ad una variabile puntatore

Puntatori: Inizializzazione


```
int number = 88;      // An int variable with a value
int * pNumber;        // Declare a pointer variable called pNumber pointing to an int (or int pointer)
pNumber = &number;    // Assign the address of the variable number to pointer pNumber

int * pAnother = &number; // Declare another int pointer and init to address of the variable number
```

In memoria principale:



Puntatori: Dereferenziazione

- L'operatore "dereferencing" (**`*`**) si applica a un puntatore e restituisce il valore contenuto nell'indirizzo memorizzato dal puntatore
- Esempio: se **`pNum`** è un puntatore a **`int`**, **`*pNum`** restituisce il valore **`int`** "puntato da" **`pNum`**

```
int number = 88;
int * pNumber = &number;  // Declare and assign the address of variable number to pointer pNumber (0x22ccec)
cout << pNumber<< endl;   // Print the content of the pointer variable, which contain an address (0x22ccec)
cout << *pNumber << endl; // Print the value "pointed to" by the pointer, which is an int (88)
*pNumber = 99;            // Assign a value to where the pointer is pointed to, NOT to the pointer variable
cout << *pNumber << endl; // Print the new value "pointed to" by the pointer (99)
cout << number << endl;   // The value of variable number changes as well (99)
```

NOTA:

Se **`*`** è usato in una dichiarazione (e.g., **`int * pNumber`**), indica che il nome che segue è una variabile puntatore

Se **`*`** è usato in un'espressione (e.g., **`*pNumber = 99; temp << *pNumber;`**), indica il valore puntato dalla variabile puntatore

Il Tipo dei Puntatori

- Un puntatore è associato a un tipo (del valore a cui punta)
- Pertanto, un puntatore può contenere soltanto indirizzi di un certo tipo
- Esempio

```
int i = 88;
double d = 55.66;
int * iPtr = &i;     // int pointer pointing to an int value
double * dPtr = &d; // double pointer pointing to a double value

iPtr = &d;    // ERROR, cannot hold address of different type
dPtr = &i;    // ERROR
iPtr = i;     // ERROR, pointer holds address of an int, NOT int value

int j = 99;
iPtr = &j;  // You can change the address stored in a pointer
```

NULL Pointer

- È possibile inizializzare un puntatore a **NULL** o **0**
- Dereferenziare un puntatore nullo causa un'eccezione **STATUS_ACCESS_VIOLATION**
- Inizializzare un puntatore a **NULL** è considerata una buona pratica

```
int * iPtr = 0;          // Declare an int pointer, and initialize the pointer to point to nothing
cout << *iPtr << endl;   // ERROR! STATUS_ACCESS_VIOLATION exception

int * p = NULL;          // Also declare a NULL pointer points to nothing
```

Riferimenti o Alias

- C++ ha introdotto variabili “reference” (riferimento), simili ai puntatori
- Un riferimento è un *alias*, ossia un nome alternativo associato ad una variabile esistente
- I riferimenti si usano principalmente per specificare i parametri formali delle funzioni (passaggio per riferimento)
- Quando una variabile è passata per riferimento come parametro di una funzione, questa lavora sulla copia originale della variabile (anziché su un clone, come nel caso del passaggio per valore)
- Le modifiche all’interno della funzione si ripercuotono anche all’esterno

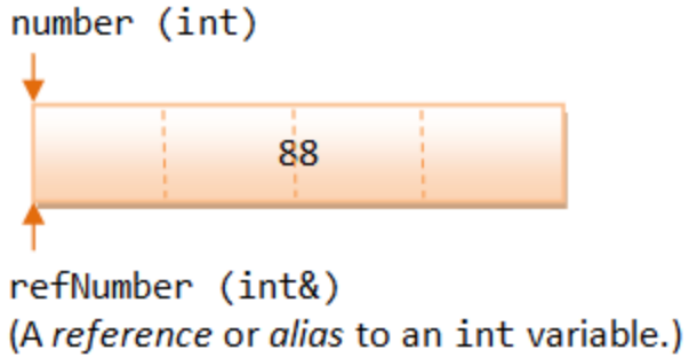
Riferimenti: Operatore &

- Il significato di **&** è diverso se usato in un'espressione o in una dichiarazione
- Quando usato in un'espressione denota l'operatore "address-of" e ritorna l'indirizzo di una variabile
- Quando è usato in una dichiarazione (incluso la firma di una funzione) è parte dell'identificatore di tipo

Riferimenti: Operatore &

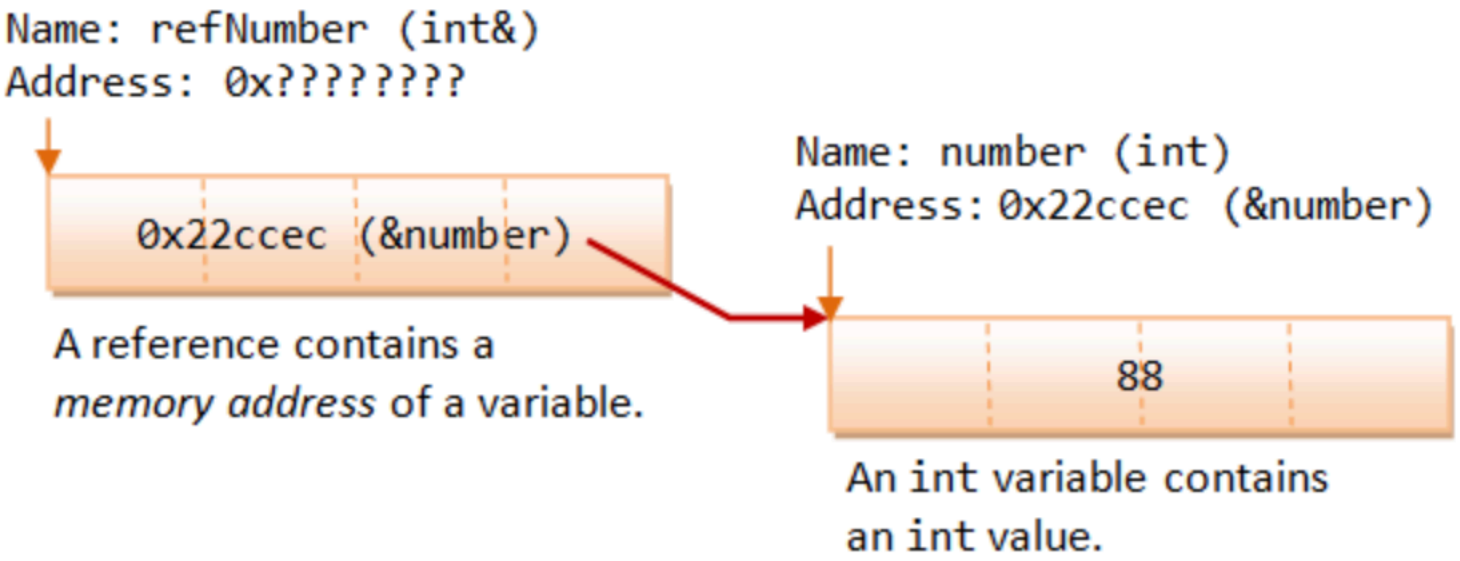

```
type &newName = existingName;
// or
type& newName = existingName;
// or
type & newName = existingName;  // I shall adopt this convention
```

```
/* Test reference declaration and initialization (TestReferenceDeclaration.cpp) */
#include <iostream>
using namespace std;

int main() {
   int number = 88;           // Declare an int variable called number
   int & refNumber = number; // Declare a reference (alias) to the variable number
                              // Both refNumber and number refer to the same value

   cout << number << endl;    // Print value of variable number (88)
   cout << refNumber << endl; // Print value of reference (88)

   refNumber = 99;            // Re-assign a new value to refNumber
   cout << refNumber << endl;
   cout << number << endl;    // Value of number also changes (99)

   number = 55;               // Re-assign a new value to number
   cout << number << endl;
   cout << refNumber << endl; // Value of refNumber also changes (55)
}
```



Riferimenti: Come funzionano?

Come i puntatori! Memorizzano l'indirizzo della variabile di cui sono alias



Riferimenti vs. Puntatori

- Sono equivalenti eccetto:
  - Un riferimento è un nome costante per un indirizzo (occorre inizializzarlo al momento della dichiarazione)

```
int & iRef;    // Error: 'iRef' declared as reference but not initialized
```

  - Per ottenere il valore puntato da un puntatore occorre usare l'operatore *****
  - Per assegnare un indirizzo di una variabile a un puntatore occorre usare l'operatore **&**
  - Le due operazioni sono implicite con i riferimenti
  - Di fatto, i riferimenti sono assimilabili a puntatori costanti (ma al contrario di questi ultimi non possono essere **NULL**)

Riferimenti vs. Puntatori


```
/* References vs. Pointers (TestReferenceVsPointer.cpp) */
#include <iostream>
using namespace std;

int main() {
   int number1 = 88, number2 = 22;

   // Create a pointer pointing to number1
   int * pNumber1 = &number1;  // Explicit referencing
   *pNumber1 = 99;             // Explicit dereferencing
   cout << *pNumber1 << endl;  // 99
   cout << &number1 << endl;   // 0x22ff18
   cout << pNumber1 << endl;   // 0x22ff18 (content of the pointer variable - same as above)
   cout << &pNumber1 << endl;  // 0x22ff10 (address of the pointer variable)
   pNumber1 = &number2;        // Pointer can be reassigned to store another address

   // Create a reference (alias) to number1
   int & refNumber1 = number1;  // Implicit referencing (NOT &number1)
   refNumber1 = 11;             // Implicit dereferencing (NOT *refNumber1)
   cout << refNumber1 << endl;  // 11
   cout << &number1 << endl;    // 0x22ff18
   cout << &refNumber1 << endl; // 0x22ff18
   //refNumber1 = &number2;     // Error! Reference cannot be re-assigned
                                // error: invalid conversion from 'int*' to 'int'
   refNumber1 = number2;        // refNumber1 is still an alias to number1.
                                // Assign value of number2 (22) to refNumber1 (and number1).
   number2++;
   cout << refNumber1 << endl;  // 22
   cout << number1 << endl;     // 22
   cout << number2 << endl;     // 23
}
```

Array e Puntatori

- In C/C++, il nome di un array è un puntatore che punta al primo elemento della sequenza (indice 0)
- Esempio:

```
/* Pointer and Array (TestPointerArray.cpp) */
#include <iostream>
using namespace std;

int main() {
   const int SIZE = 5;
   int numbers[SIZE] = {11, 22, 44, 21, 41};  // An int array

   // The array name numbers is an int pointer, pointing at the
   // first item of the array, i.e., numbers = &numbers[0]
   cout << &numbers[0] << endl; // Print address of first element (0x22fef8)
   cout << numbers << endl;     // Same as above (0x22fef8)
   cout << *numbers << endl;          // Same as numbers[0] (11)
   cout << *(numbers + 1) << endl;    // Same as numbers[1] (22)
   cout << *(numbers + 4) << endl;    // Same as numbers[4] (41)
}
```

Aritmetica dei Puntatori


```
int numbers[] = {11, 22, 33};
int * iPtr = numbers;
cout << iPtr << endl;          // 0x22cd30
cout << iPtr + 1 << endl;      // 0x22cd34 (increase by 4 - sizeof int)
cout << *iPtr << endl;         // 11
cout << *(iPtr + 1) << endl; // 22
cout << *iPtr + 1 << endl;    // 12
```

```
int numbers[100];
cout << sizeof(numbers) << endl;      // Size of entire array in bytes (400)
cout << sizeof(numbers[0]) << endl;  // Size of first element of the array in bytes (4)
cout << "Array size is " << sizeof(numbers) / sizeof(numbers[0]) << endl;  // (100)
```

Funzioni e Passaggio dei Parametri

- Passaggio per Valore
- Passaggio per Riferimento (con puntatori)
- Passaggio per Riferimento (con alias)

Passaggio per Valore

- In C/C++, i parametri “attuali”* sono passati alle funzioni per valore (eccetto gli array che sono trattati come puntatori)
- Viene creato un clone del parametro e passato all’interno della funzione
- Eventuali modifiche alla copia/clone all’interno della funzione **non** si ripercuotono sull’argomento originale del chiamante

* Il termine parametri “attuali”, sebbene entrato nell’uso comune, è frutto di una traduzione poco felice del corrispettivo inglese *actual parameters*, con cui si intende specificare gli argomenti effettivamente passati in input ad una funzione, distinti dai cosiddetti *formal parameters* che invece sono utilizzati per la sua dichiarazione

Passaggio per Valore


```
/* Pass-by-value into function (TestPassByValue.cpp) */
#include <iostream>
using namespace std;

int square(int);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;                     // 8
   cout << square(number) << endl; // 64
   cout << number << endl;                     // 8 - no change
}

int square(int n) {  // non-const
   cout <<  "In square(): " << &n << endl;  // 0x22ff00
   n *= n;             // clone modified inside the function
   return n;
}
```

Passaggio per Riferimento (Puntatori)

- Talvolta si vuole ottenere una modifica dell'argomento all'interno della funzione senza creare un clone
- È possibile realizzare questa funzionalità passando un puntatore alla funzione
- **NOTA:** Il puntatore è comunque passato per valore ma potendo usare l'operatore ***** è possibile creare *side effects*

Passaggio per Riferimento (Puntatori)


```
/* Pass-by-reference using pointer (TestPassByPointer.cpp) */
#include <iostream>
using namespace std;

void square(int *);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;    // 8
   square(&number);           // Explicit referencing to pass an address
   cout << number << endl;    // 64
}

void square(int * pNumber) {  // Function takes an int pointer (non-const)
   cout <<  "In square(): " << pNumber << endl;  // 0x22ff1c
   *pNumber *= *pNumber;      // Explicit de-referencing to get the value pointed-to
}
```

Passaggio per Riferimento (Alias)

- Un comportamento analogo si può ottenere passando direttamente un alias alla funzione
- In questo modo però l'operazione di "address-of" (**&**) e quella di "dereferencing" (*****) sono implicite!
- Quello che cambia è solamente la dichiarazione dei parametri della funzione
- NOTA: Gli alias devono essere sempre inizializzati **contestualmente** alla dichiarazione
- Nel caso vengano usati come parametri formali, questi vengono inizializzati al momento della chiamata (con i parametri "attuali" specificati dal chiamante)

Passaggio per Riferimento (Alias)


```
/* Pass-by-reference using reference (TestPassByReference.cpp) */
#include <iostream>
using namespace std;

void square(int &);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;  // 8
   square(number);          // Implicit referencing (without '&')
   cout << number << endl;  // 64
}

void square(int & rNumber) {  // Function takes an int reference (non-const)
   cout <<  "In square(): " << &rNumber << endl;  // 0x22ff1c
   rNumber *= rNumber;         // Implicit de-referencing (without '*')
}
```

Ritorno da Funzione

- Il valore di ritorno di una funzione (non **`void`**) può essere restituito come puntatore o alias

```
/* Passing back return value using reference (TestPassByReferenceReturn.cpp) */
#include <iostream>
using namespace std;

int & squareRef(int &);
int * squarePtr(int *);

int main() {
   int number1 = 8;
   cout <<  "In main() &number1: " << &number1 << endl;  // 0x22ff14
   int & result = squareRef(number1);
   cout <<  "In main() &result: " << &result << endl;  // 0x22ff14
   cout << result << endl;   // 64
   cout << number1 << endl;  // 64

   int number2 = 9;
   cout <<  "In main() &number2: " << &number2 << endl;  // 0x22ff10
   int * pResult = squarePtr(&number2);
   cout <<  "In main() pResult: " << pResult << endl;  // 0x22ff10
   cout << *pResult << endl;   // 81
   cout << number2 << endl;    // 81
}

int & squareRef(int & rNumber) {
   cout <<  "In squareRef(): " << &rNumber << endl;  // 0x22ff14
   rNumber *= rNumber;
   return rNumber;
}

int * squarePtr(int * pNumber) {
   cout <<  "In squarePtr(): " << pNumber << endl;  // 0x22ff10
   *pNumber *= *pNumber;
   return pNumber;
}
```

Ritorno da Funzione

- Non si deve **mai** ritornare il valore di una variabile locale allo scope della funzione come puntatore o alias
- Le variabili locali (alla funzione) esistono **solo** all'interno della funzione e vengono deallocate una volta terminata la sua esecuzione
- Il compilatore (GCC) solleva un warning (ma non un errore)

```
/* Test passing the result (TestPassResultLocal.cpp) */
#include <iostream>
using namespace std;

int * squarePtr(int);
int & squareRef(int);

int main() {
   int number = 8;
   cout << number << endl;  // 8
   cout << *squarePtr(number) << endl;  // ??
   cout << squareRef(number) << endl;   // ??
}

int * squarePtr(int number) {
   int localResult = number * number;
   return &localResult;
      // warning: address of local variable 'localResult' returned
}

int & squareRef(int number) {
   int localResult = number * number;
   return localResult;
      // warning: reference of local variable 'localResult' returned
}
```

Passaggio di Array a Funzione

- Un array viene passato ad una funzione come un puntatore al primo elemento
- Nella dichiarazione di funzione si può usare sia la notazione tipica degli array (**`int []`**) che quella dei puntatori (**`int *`**)
- Il compilatore tratterà l'argomento sempre come un puntatore (al primo elemento)
- Esempio di dichiarazioni equivalenti:

```
int max(int numbers[], int size);
int max(int *numbers, int size);
int max(int number[50], int size);
```

Passaggio di Array a Funzione

- Poiché trattati come puntatori, il passaggio degli array avviene per riferimento
- In altre parole, il chiamante ha la facoltà di modificare il contenuto dell'array attraverso l'uso dell'operatore `*`
- È possibile dichiarare il parametro costante (`const`) in modo da evitare side effect all'interno della funzione
- **NOTA:** La dimensione dell'array deve essere passata come parametro addizionale; il compilatore non è in grado di inferire la dimensione dell'array né di eseguire controlli sui "bound"

Passaggio di Array a Funzione


```
/* Passing array in/out function (TestArrayPassing.cpp) */
#include <iostream>
using namespace std;

// Function prototypes
int max(const int arr[], int size);
void replaceByMax(int arr[], int size);
void print(const int arr[], int size);

int main() {
   const int SIZE = 4;
   int numbers[SIZE] = {11, 22, 33, 22};
   print(numbers, SIZE);
   cout << max(numbers, SIZE) << endl;
   replaceByMax(numbers, SIZE);
   print(numbers, SIZE);
}

// Return the maximum value of the given array.
// The array is declared const, and cannot be modified inside the function.
int max(const int arr[], int size) {
   int max = arr[0];
   for (int i = 1; i < size; ++i) {
      if (max < arr[i]) max = arr[i];
   }
   return max;
}

// Replace all elements of the given array by its maximum value
// Array is passed by reference. Modify the caller's copy.
void replaceByMax(int arr[], int size) {
   int maxValue = max(arr, size);
   for (int i = 0; i < size; ++i) {
      arr[i] = maxValue;
   }
}

// Print the array's content
void print(const int arr[], int size) {
   cout << "{";
   for (int i = 0; i < size; ++i) {
      cout << arr[i];
      if (i < size - 1) cout << ",";
   }
   cout << "}" << endl;
}
```

notazione array

Passaggio di Array a Funzione


```
/* Passing array in/out function using pointer (TestArrayPassingPointer.cpp) */
#include <iostream>
using namespace std;

// Function prototype
int max(const int *arr, int size);

int main() {
   const int SIZE = 5;
   int numbers[SIZE] = {10, 20, 90, 76, 22};
   cout << max(numbers, SIZE) << endl;
}

// Return the maximum value of the given array
int max(const int *arr, int size) {
   int max = *arr;
   for (int i = 1; i < size; ++i) {
      if (max < *(arr+i)) max = *(arr+i);
   }
   return max;
}
```

notazione puntatori