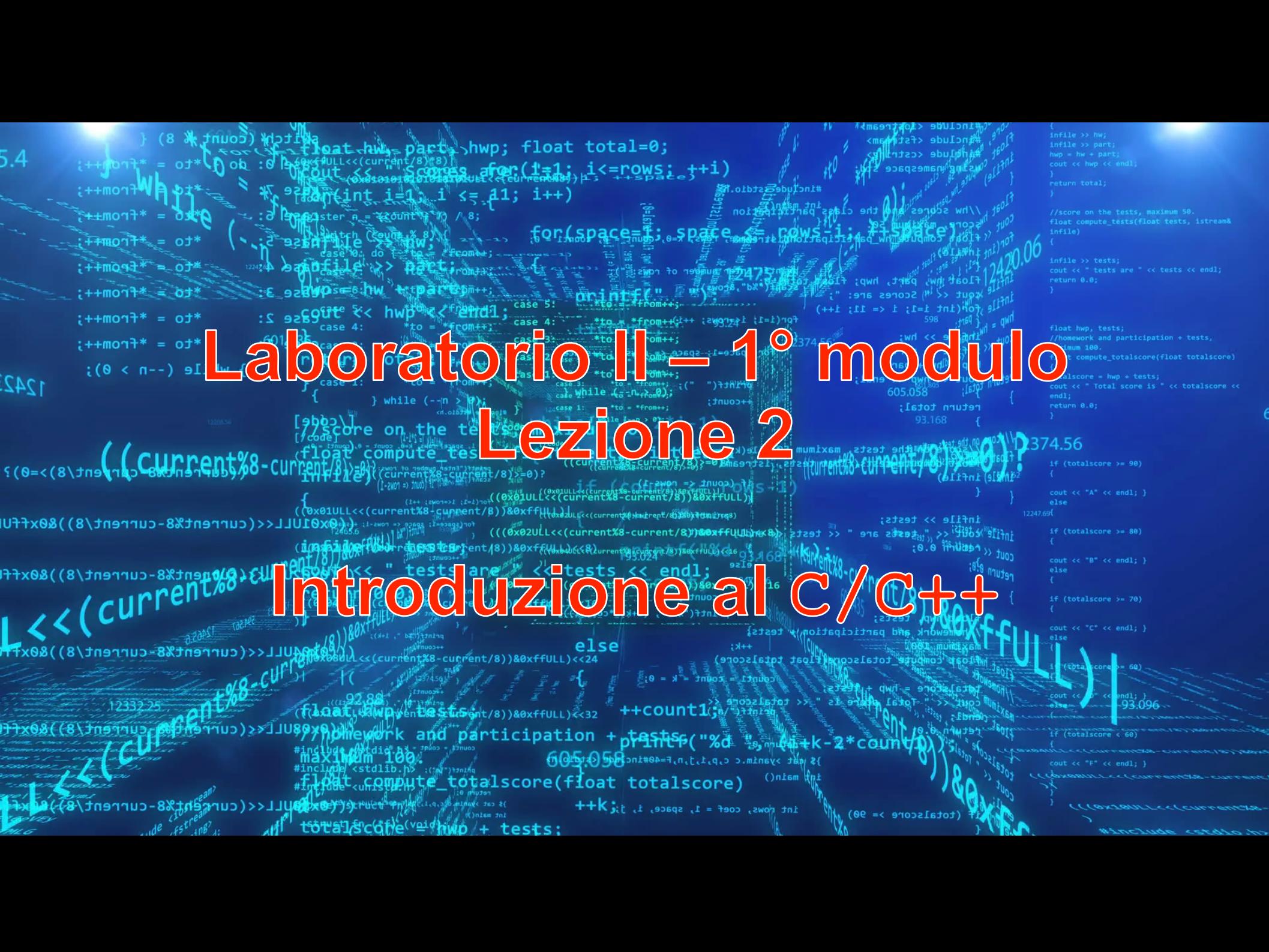


# Laboratorio II – 1° modulo

## Lezione 2

### Introduzione al C/C++



# Indice

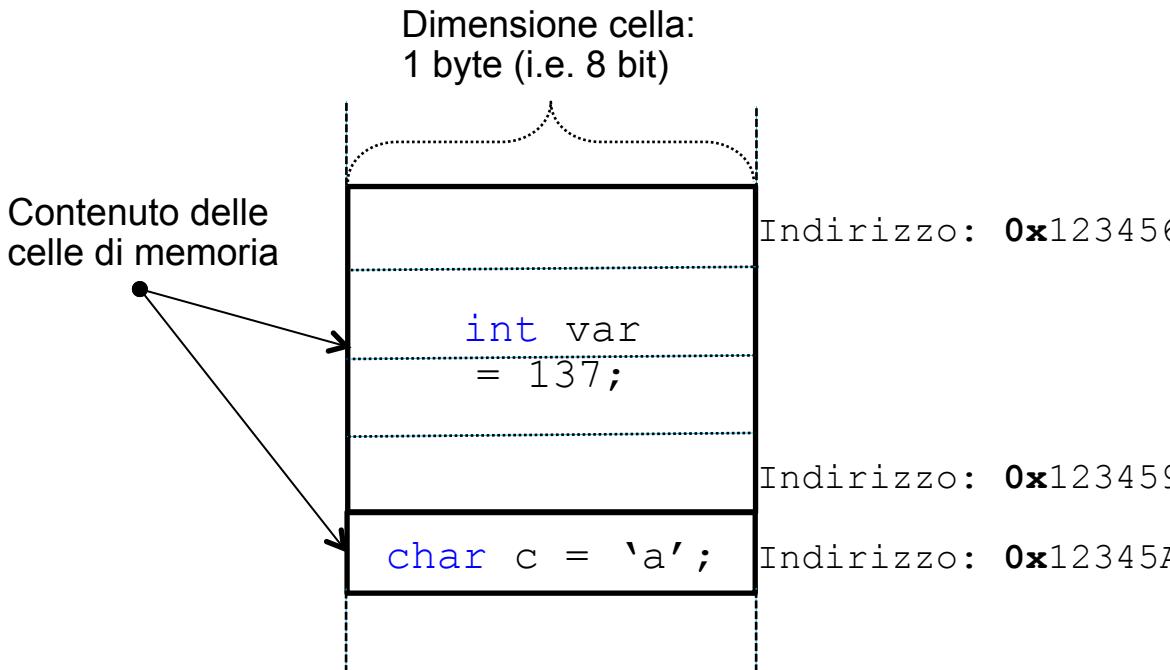
---

- Gestione della memoria
  - puntatori e referenze
  - passaggio dei parametri alle funzioni: valore / referenza
  - gestione dinamica della memoria
- Passaggio dei parametri al `main` program
- Suddivisione di un programma in file (`.cpp`, `.cc`, `.h`)
- Esercizi
- Come passare per referenza un array ad una funzione
- *Exception handling*

# I puntatori e la gestione della memoria

# Rappresentazioni delle variabili nella memoria del calcolatore

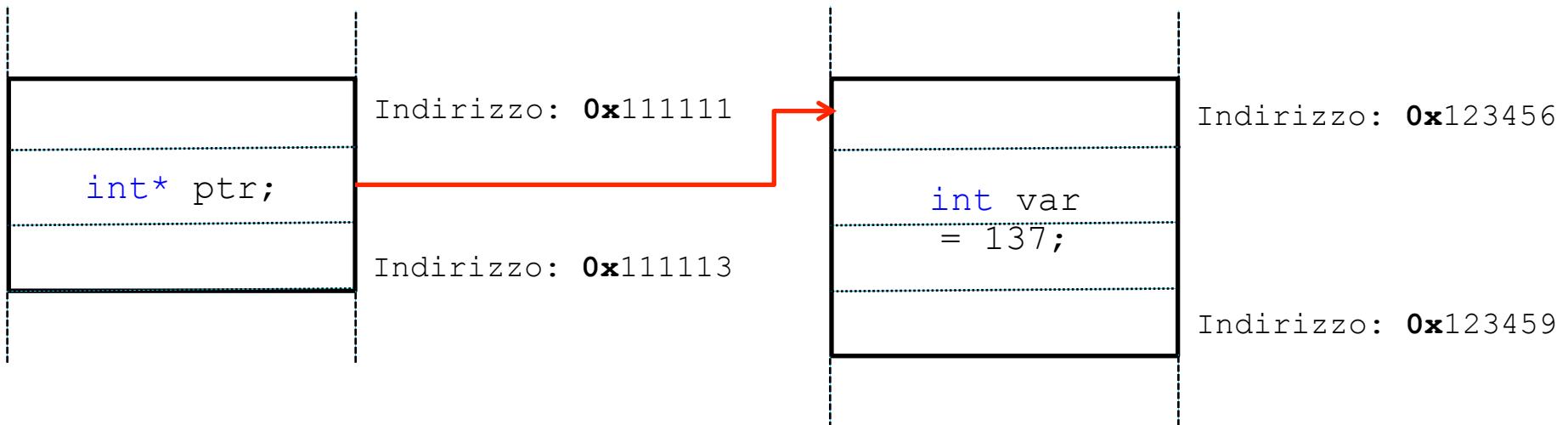
- In un computer tutto è rappresentato mediante numeri
- I numeri sono salvati in celle di memoria (Random Access Memory = RAM)
- Le celle della memoria sono individuate mediante indirizzi



| Type               | Typical Bit Width | Typical Range                   |
|--------------------|-------------------|---------------------------------|
| char               | 1byte             | -127 to 127 or 0 to 255         |
| unsigned char      | 1byte             | 0 to 255                        |
| signed char        | 1byte             | -127 to 127                     |
| int                | 4bytes            | -2147483648 to 2147483647       |
| unsigned int       | 4bytes            | 0 to 4294967295                 |
| signed int         | 4bytes            | -2147483648 to 2147483647       |
| short int          | 2bytes            | -32768 to 32767                 |
| unsigned short int | Range             | 0 to 65,535                     |
| signed short int   | Range             | -32768 to 32767                 |
| long int           | 4bytes            | -2,147,483,648 to 2,147,483,647 |
| signed long int    | 4bytes            | same as long int                |
| unsigned long int  | 4bytes            | 0 to 4,294,967,295              |
| float              | 4bytes            | +/- 3.4e +/- 38 (~7 digits)     |
| double             | 8bytes            | +/- 1.7e +/- 308 (~15 digits)   |
| long double        | 8bytes            | +/- 1.7e +/- 308 (~15 digits)   |
| wchar_t            | 2 or 4 bytes      | 1 wide character                |

# I puntatori

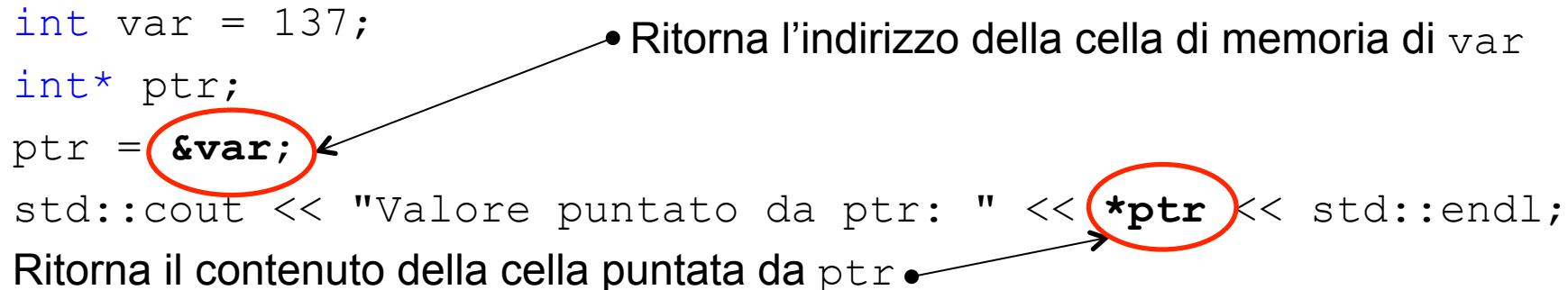
- In C/C++ è possibile definire anche delle variabili che contengono indirizzi di memoria: **puntatori**
- Sintassi: `type* ptr;`
  - Indica che la variabile `ptr` è di tipo puntatore
  - La variabile `ptr` punta ad una cella di memoria di tipo `type` (e.g. `int`) e quindi il numero a cui fa riferimento è composto da 4 byte ed è da interpretarsi come un intero



# Come manipolare i puntatori

- Data una variabile (e.g. `var`) come faccio a sapere il suo indirizzo di memoria?
- Data una variabile puntatore (e.g. `ptr`) come faccio a farmi dare il contenuto della cella di memoria a cui punta?
- **Risposte:** si utilizzano due operatori speciali, `*` e `&`
  - L'operatore `&` ritorna l'indirizzo della variabile (i.e. della cella di memoria)
  - L'operatore `*` ritorna il contenuto della variabile (i.e. della cella di memoria)

```
int var = 137;
int* ptr;
ptr = &var;          • Ritorna l'indirizzo della cella di memoria di var
std::cout << "Valore puntato da ptr: " << *ptr << std::endl;
Ritorna il contenuto della cella puntata da ptr
```



# Esempio sull'uso dei puntatori

```
#include <iostream>
```

```
int main()
{
    int var = 137;
    std::cout << "var vale: " << var
        << " ed il suo indirizzo e': " << &var << std::endl;

    int* ptr;
    std::cout << "Indirizzo a cui punta ptr e': " << ptr << std::endl;

    ptr = &var;
    std::cout << "Ora ptr punta a var: " << ptr
        << " ed il valore a cui punta e': " << *ptr << std::endl;

    *ptr = 100;
    std::cout << "var ora vale: " << var << std::endl;

    int pippo = *ptr;
    std::cout << "La variabile pippo vale: " << pippo << std::endl;

    (*ptr)++;
    std::cout << "var ora vale: " << var << " e pippo vale: " << pippo << std::endl;

    int vec[] = {2, 20};
    std::cout << "Vec[0] = " << *vec << "; Vec[1] = " << *(vec+1) << std::endl;
}
```

**Attenzione:** le variabili non sono mai implicitamente inizializzate

```
var vale: 137 ed il suo indirizzo e': 0x7fff50c8a5b8
Indirizzo a cui punta ptr 0x7fff50c8a670
Ora ptr punta a var: 0x7fff50c8a5b8 ed il valore a cui punta e' 137
var ora vale: 100
La variabile pippo vale: 100
var ora vale: 101 e pippo vale: 100
Vec[0] = 2; Vec[1] = 20
```

# Puntatori e array

- Quando si definisce un array, ad esempio

```
int vec[] = {2, 20};
```

vec contiene l'indirizzo di memoria del primo elemento dell'array:

```
std::cout << vec << std::endl;
```

- Quindi il nome usato per definire l'array (vec) è un **puntatore**
- Attraverso `*vec` leggo il valore del primo elemento dell'array, con `* (vec+1)` leggo il secondo elemento dell'array, etc...

```
int main()
{
    int vec[] = {2, 20};
    std::cout << "Vec[0] = " << *vec << "; Vec[1] = " << *(vec+1) << std::endl;
    return 0;
}
```

Se stampo vec a terminale ottengo un numero esadecimale, corrispondente all'indirizzo di memoria da cui parte l'array vec

Questa sintassi è del tutto equivalente alle parentesi [] (molto più intuitive, e consigliate):

**\*vec** equivale a **vec[0]**

**\* (vec+i)** equivale a **vec[i]**

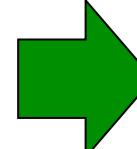
# Ancora puntatori e riepilogo

- Anche i puntatori sono variabili e possono cambiare valore
  - Un puntatore si può creare senza assegnargli un valore
  - Il valore del puntatore è l'indirizzo di memoria della variabile alla quale punta

```
double pi_greco = 3.1415;  
double* ptr;  
ptr = &pi_greco;  
std::cout << "Valore puntato: " << *ptr << std::endl;  
double nepero = 2.7183;  
ptr = &nepero;  
std::cout << "Valore puntato: " << *ptr << std::endl;
```

Riepiloghiamo la sintassi ed il significato:

```
int var = 137;  
int* ptr;  
ptr = &var;  
int new_var = *ptr;
```



Creazione variabile **int**  
Creazione variabile puntatore a **int**  
Assegnazione a **ptr** dell'indirizzo di **var**  
Contenuto della cella puntata da **ptr**

# Le referenze (i.e. alias)

- Le **referenze** sono degli **alias** per i nomi delle variabili. La variabile o la sua referenza sono la stessa cosa

```
double pi_greco = 3.1415;
```

```
double& ref = pi_greco;
std::cout << "ref e` un alias di pi_greco: "
    << ref << std::endl;
```

```
pi_greco = 3.141592;
std::cout << "ref e` un alias di pi_greco: "
    << ref << std::endl;
```

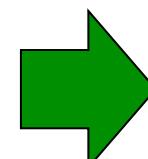
- Una referenza si crea a partire da una variabile esistente
- Particolarmente utile per passare variabili a/da funzioni

*Cosa ottengo se eseguo questo codice?*

```
std::cout << pi_greco << std::endl;
std::cout << ref << std::endl;
```

Riepiloghiamo la sintassi ed il significato:

```
int var = 137;
int* ptr;
ptr = &var;
int new_var = *ptr;
int& ref = var;
```



Creazione variabile `int`  
 Creazione variabile puntatore a `int`  
 Indirizzo di memoria di `var`  
 Contenuto della cella puntata da `ptr`  
 Creazione di un alias di `var`

# Il passaggio di argomenti alle funzioni

- Quello che abbiamo visto fino ad ora può essere applicato anche al passaggio di argomenti alle funzioni

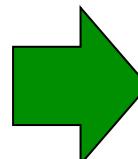
```
int raddoppia(int input)
{
    return input * 2;
}
```

```
void raddoppiaPointer(int* input)
{
    *input = *input * 2;
}
```

```
void raddoppiaReference(int& input)
{
    input = input * 2;
}
```

```
int var = 4;

raddoppia(var);
raddoppiaPointer(&var);
raddoppiaReference(var);
```



Come varia var?

var rimane = 4, la funzione restituisce 8  
var diventa = 8  
var diventa = 8

# Passaggio per valore

## Main program

Indirizzo: 0x123456

```
int var  
= 137;
```

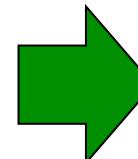
Indirizzo: 0x123459

```
int raddoppia(int input)  
{  
    return input * 2;  
}  
  
int main ()  
{  
    int var = 137;  
    std::cout << raddoppia(var);  
    return 0;  
}
```

## Funzione raddoppia

Indirizzo: 0x111111

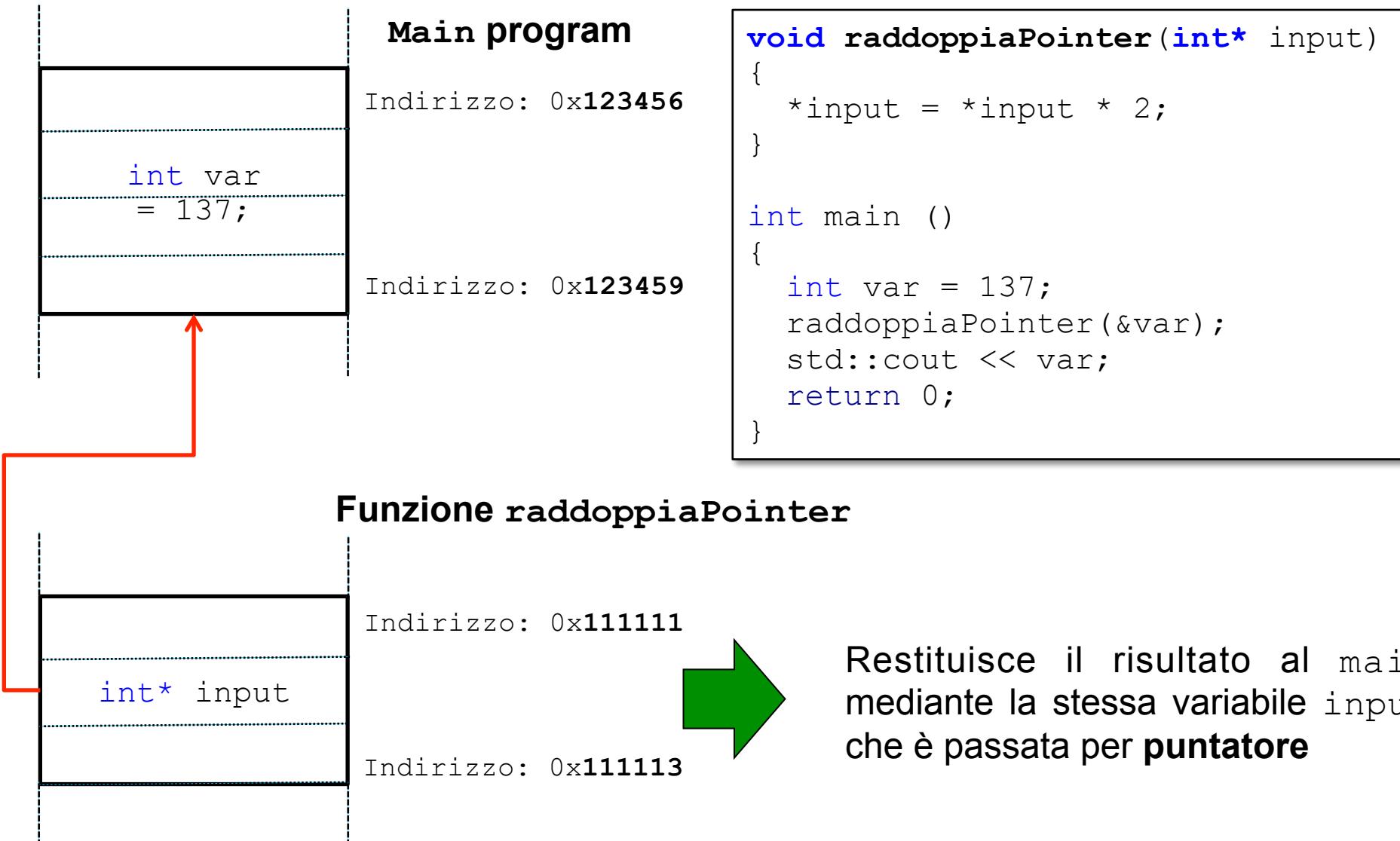
```
int input  
(137)
```



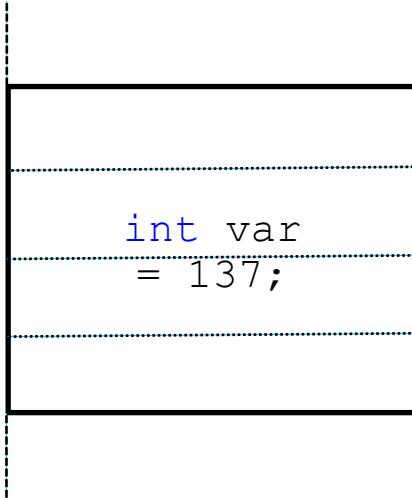
Indirizzo: 0x111114

Restituisce il **valore** del risultato al main mediante un tipo `int` con l'istruzione `return input * 2;`

# Passaggio per puntatore



# Passaggio per referenza



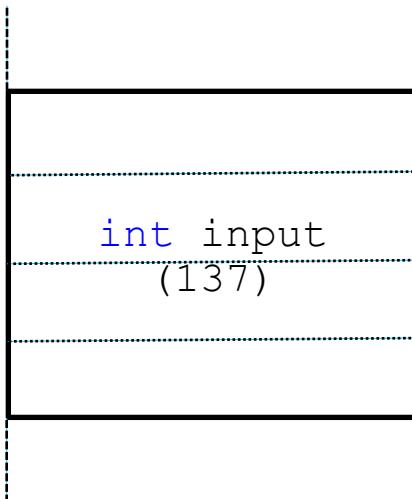
## Main program

Indirizzo: `0x123456`

```
int var  
= 137;
```

Indirizzo: `0x123459`

```
void raddoppiaReference(int& input)  
{  
    input = input * 2;  
}  
  
int main ()  
{  
    double var = 137;  
    raddoppiaReference(var);  
    std::cout << var;  
    return 0;  
}
```

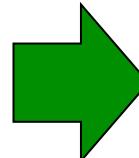


## Funzione `raddoppiaReference`

Indirizzo: `0x123456`

```
int input  
(137)
```

Indirizzo: `0x123459`



Restituisce il risultato al `main` mediante la stessa variabile `input` che è passata per **referenza**

# Gestione dinamica della memoria

**Problema:** voglio poter creare un vettore delegando la creazione ad una funzione. Come faccio a creare il vettore nella funzione e passare la struttura all'esterno?

**Risposta:** devo gestire la memoria in maniera dinamica, mi devo preoccupare di riservare le celle di memoria (istruzione **new**) e anche della loro cancellazione/liberazione (istruzione **delete**)

```
int* creaVettore(int dim)
{
    int array[dim];
    return array;
}
```

Modo errato

```
int* creaVettore(int dim)
{
    int* array = new int[dim];
    return array;
}

...
int* vec = creaVettore(4);
...
delete[] vec;
...
```

Modo corretto

# Gestione dinamica della memoria

```
int* creaVettore(int dim)
{
    int array[dim];
    return array;
}
```

## Modo errato

- Tutte le variabili “statiche” create nella funzione vivono solo nella funzione. Si dice che vanno *out of scope* dopo che la funzione è terminata
- Non c’è modo, facendo uso della memoria “statica”, di creare un vettore in una funzione e di passarlo all’esterno

```
int* creaVettore(int dim)
{
    int* array = new int[dim];
    return array;
}

...
int* vec = creaVettore(4);

...
delete[] vec;

...
```

## Modo corretto

- L’istruzione `new` alloca una struttura nell’area di memoria dinamica e ne **restituisce il puntatore**
- Il tipo di struttura è specificato alla destra di `new`
- Nell’esempio: `new` crea `dim` celle di tipo `int` (4 byte x `dim`) e **restituisce il puntatore alla prima cella**

# Gestione dinamica della memoria

```
#include <iostream>

int* creaVettore (int dim)
{
    int* array = new int[dim];
    return array;
}

int main()
{
    int dim = 0;
    std::cout << "Inserisci la dimensione del vettore: ";
    std::cin >> dim;

    if (dim < 0)
    {
        std::cout << "Numero negativo" << std::endl;
        return -1;
    }

    // Alloca dinamicamente un vettore di dim celle
    int* vec = creaVettore(dim);

    // Riempì il vettore
    for (int i = 0; i < dim; ++i)
    {
        vec[i] = i+1;
    }

    // Stampa gli elementi del vettore
    for (int i = 0; i < dim; ++i)
    {
        std::cout << "Contenuto della cella " << i << " : " << vec[i] << std::endl;
    }

    delete[] vec;

    return 0;
}
```

**N.B.: l'operatore `new` restituisce sempre il puntatore alla struttura dati creata**

Dopo `new` è buona norma controllare che l'operazione sia andata a buon fine, e.g.:

```
int* array = new (std::nothrow) int[dim];
if (array == nullptr)
    "l'allocazione non ha funzionato"
else
    "tutto ok, prosegui"
...
```

Oppure mediante le istruzioni per individuare eccezioni:

```
int* array;
try
{
    array = new int[dim];
}
catch (const std::bad_alloc& e)
{
    "l'allocazione non ha funzionato"
    std::cout << e.what() << std::endl;
    ...
}
```

**N.B.: `nullptr` introdotto dal C++11 in poi, altrimenti usare `NULL` tramite `#include <stddef.h>`**

# Gestione dinamica della memoria

- Gli operatori `new` e `delete` si possono usare non solo per definire degli array allocati dinamicamente, ma anche semplici variabili
- In questo caso la sintassi è la seguente:

```
double* pointer = new double(1.5);
```

```
std::cout << *pointer << std::endl;
```

```
delete pointer;
```

Lo spazio di memoria occupato dalla variabile `double` viene liberato (in questo caso non si usano le parentesi `[]` perché `pointer` non punta ad un array)

Valore a cui viene inizializzata la cella di memoria puntata da `pointer`

E` capitato, o capiterà, a tutti di aver problemi di *memory leak* nei propri programmi. La ragione è che il C++ non ha un modo di de-allocare la memoria in maniera automatica (altri linguaggi, come per esempio Python, invece sì). Il meccanismo di de-allocazione automatica della memoria si chiama **Garbage Collector**: appena una zona di memoria riservata dal programma non è più referenziata viene automaticamente resa libera. Perché il C++ non implementa un meccanismo di *garbage collection*? Beh, perché il meccanismo ha un prezzo sia in termini di memoria sia in termini di tempo

# Esempi di uso scorretto della memoria dinamica

Cosa c'è di sbagliato in queste linee di codice?

```
int anArray[10];
int* num;
int* vec = new int[10];
num = anArray;
vec = num;
delete[] vec;
```

```
int nElem;
...
double* myArray;
if (nElem > 0)
    myArray = new double[nElem];
...
delete[] myArray;
```

**Messaggio di errore in fase di esecuzione:**

```
malloc: *** error for object 0x7fffea6445f0: pointer being
freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

E in queste linee di codice?

```
int nElem;
double* myArray = new double[10];
...
if (nElem > 0)
    myArray = new double[nElem];
...
delete[] myArray;
```

# Esempi di uso scorretto della memoria dinamica

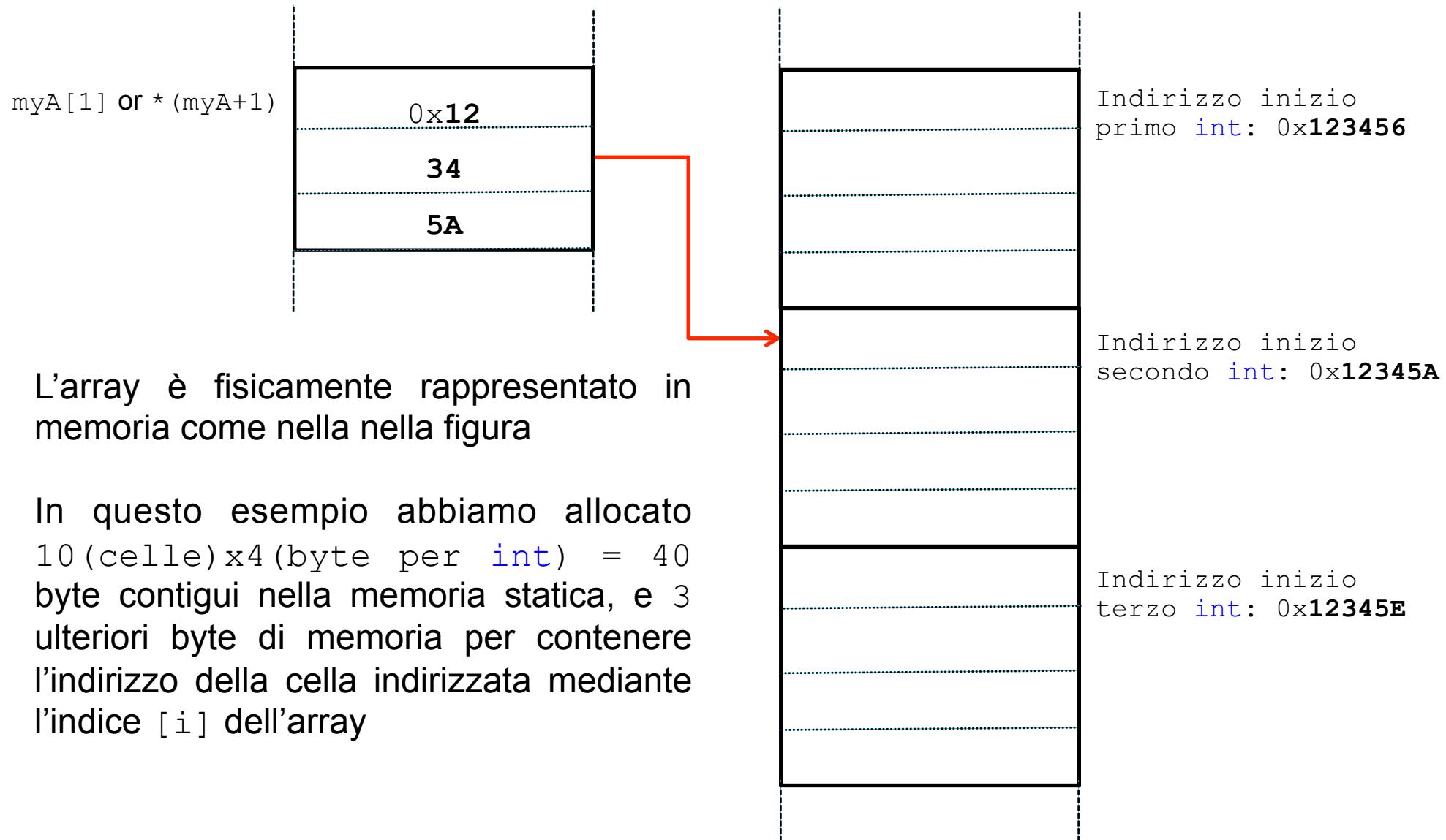
```
int myFunction (int* inArray, int dim) (A)
{
    int sumEven = 0;
    int myArray[dim];
    unsigned int indx = 0;
    for (unsigned int i = 0; i < 10; i++)
    {
        if (inArray[i] % 2 == 0)
        {
            myArray[indx] = inArray[i];
            indx++;
        }
    }
    for (unsigned int i = 0; i < indx; i++)
        sumEven += myArray[i];
    return sumEven;
}
```

```
int myFunction (int* inArray, int dim) (B)
{
    int sumEven = 0;
    int* myArray = new int[dim];
    unsigned int indx = 0;
    for (unsigned int i = 0; i < 10; i++)
    {
        if (inArray[i] % 2 == 0)
        {
            myArray[indx] = inArray[i];
            indx++;
        }
    }
    for (unsigned int i = 0; i < indx; i++)
        sumEven += myArray[i];
    return sumEven;
}
```

Cosa succede alla fine dell'esecuzione delle funzioni?

# Rappresentazione in memoria degli array

```
int* myA[10];
```

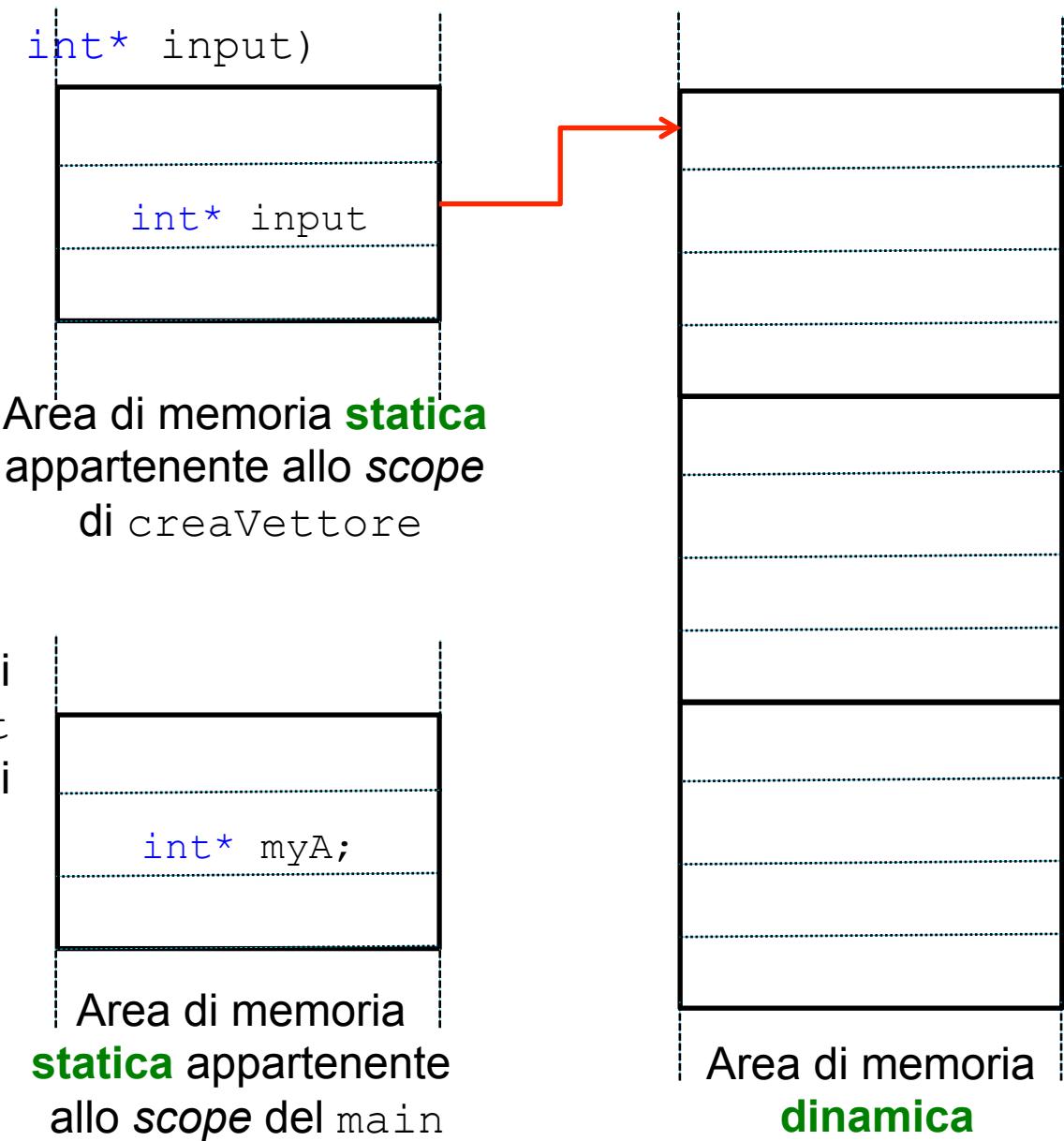


# Rappresentazione in memoria degli array

```
void creaVettore(int dim; int* input)
{
    input= new int[dim];
}

int main()
{
    ...
    int* myA;
    creaVettore(10,myA);
    ...
}
```

Appena si chiude lo *scope* di creaVettore la variabile input va *out of scope*, perdendo così l'indirizzo dell'array



# Rappresentazione in memoria degli array

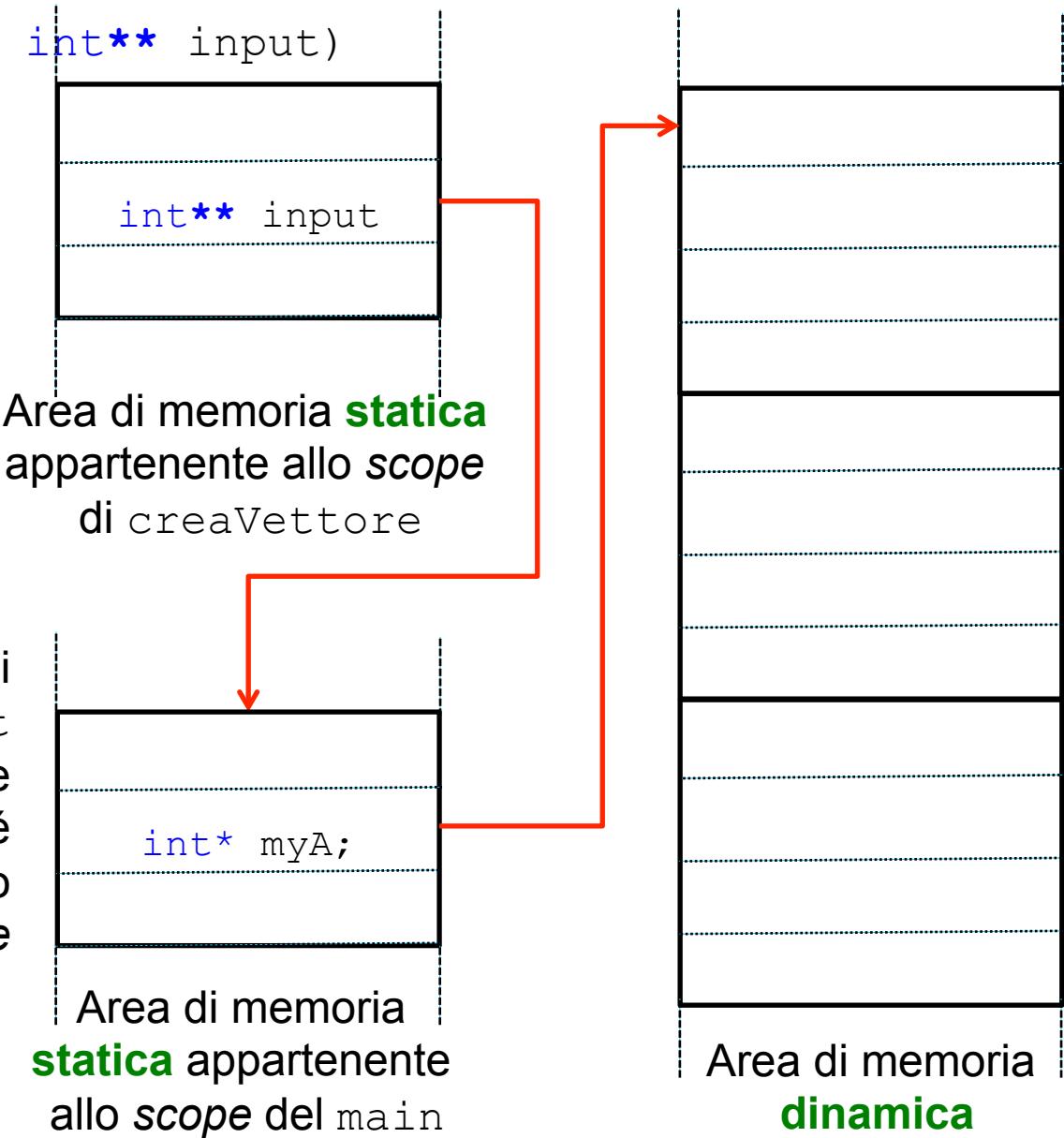
```

void creaVettore(int dim; int** input)
{
    *input= new int[dim];
}

int main()
{
    ...
    int* myA;
    creaVettore(10, &myA);
    ...
}

```

Appena si chiude lo *scope* di `creaVettore` la variabile `input` va *out of scope*, ma non viene perso l'indirizzo dell'array perché in realtà abbiamo operato nell'area di memoria dello *scope* del `main`



# **Passaggio dei parametri al main program**

# Passaggio di argomenti da terminale (argc e argv)

**Obiettivo:** vogliamo che il programma riceva e utilizzi degli argomenti (numeri, stringhe di caratteri...) che vengono passati da terminale in fase di esecuzione

Come funziona?

- Definisco il `main` con due parametri:
  - `int argc`: “argument counter” (numero degli argomenti)
  - `char** argv`: “argument vector” (array contenente gli argomenti)
- I nomi sono convenzioni, possono anche essere diversi
- Gli argomenti sono poi passati da terminale in fase di esecuzione:  
`./test 10 20 3000 ciao`

# Passaggio di argomenti da terminale (**argc** e **argv**)

```
#include <iostream>

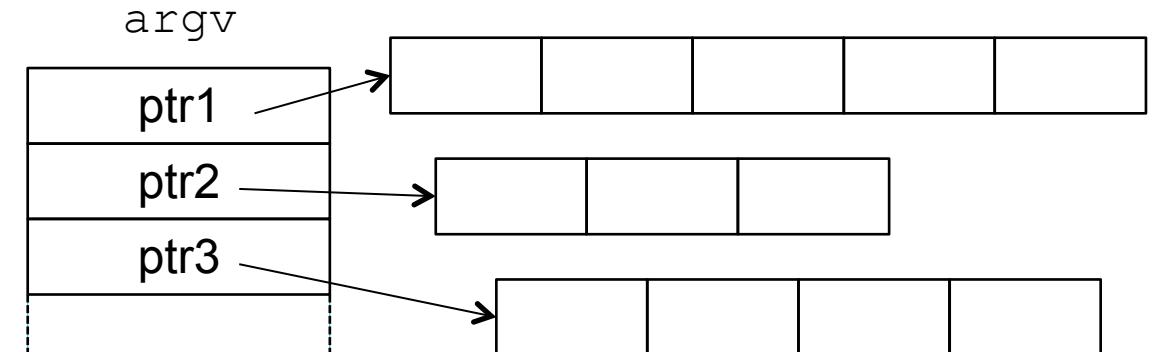
int main (int argc, char** argv)
{
    std::cout << "Ho " << argc << " argomenti" << std::endl;

    for (int i = 0; i < argc; i++)
    {
        std::cout << "argv[" << i << "]": " << argv[i] << std::endl;
    }

    return 0;
}
```

```
./test 10 20 3000 ciao
Ho 5 argomenti
argv[0]: ./test
argv[1]: 10
argv[2]: 20
argv[3]: 3000
argv[4]: ciao
```

Ciascun elemento di **argv** e` un vettore di caratteri



# **Suddivisione di un programma in file (.cpp, .cc, .h)**

# Divisione in diversi file

Per evitare di avere programmi troppo lunghi e per suddividere il codice in unità logiche separate (facilitando così lo sviluppo condiviso del codice), le funzioni sono impacchettate in **librerie**

```
#ifndef myLib_h  
#define myLib_h
```

```
double raddoppia (double input);
```

```
#endif
```

**myLib.h**

Contiene le definizioni di variabili e funzioni (i.e. i prototipi)

Le istruzioni `#ifndef`, `#define`, `#endif` sono direttive al preprocessore e servono per evitare di duplicare le definizioni se il file viene incluso più volte

```
#include "myLib.h"
```

**myLib.cc**

```
double raddoppia (double input)  
{  
    return input * 2.;  
}
```

Contiene l'implementazione delle funzioni

Conosce il prototipo da `myLib.h` attraverso l'istruzione `#include`

# Uso nel programma principale

- I prototipi delle librerie vengono inclusi mediante la direttiva al preprocessore `#include "nomeLib.h"`

```
#include <iostream>
#include "myLib.h"
int main ()
{
    double valore_iniziale = 4;
    double valore_finale = raddoppia(valore_iniziale);

    std::cout << "Raddoppia: " << valore_iniziale << " x 2 = "
                  << valore_finale << std::endl;

    return 0;
}
```

- La sintassi è nota dall'inclusione di `myLib.h`
- Le librerie ed il programma principale vengono compilati e "linkati" insieme formando un unico codice eseguibile:

```
c++ -o testLib testLib.cpp myLib.cc
```

# Esercizi

---

- **Esercizio 1:** Scrivere un programma che assegna il valore di una variabile ad un'altra utilizzando un puntatore. Stampare inoltre a terminale i valori e gli indirizzi di ogni variabile prima e dopo l'assegnazione
- **Esercizio 2:** Dichiare un puntatore e poi cercare di assegnargli direttamente un valore numerico. Cosa succede? Perché?
- **Esercizio 3:** Utilizzare `new` e `delete` per creare e distruggere una variabile `double` ed un array di `double`

# Esercizi

---

- **Esercizio 4:** Realizzare una funzione che risolve un'equazione di secondo grado:  $ax^2 + bx + c = 0$ . La funzione deve rendere disponibile il risultato al programma che la chiama. Il prototipo della funzione deve essere:

```
bool solve2ndDegree(double* par, double* x);
```

- La funzione deve restituire una variabile `bool` (`true/false`) a seconda che esistano o meno soluzioni reali dell'equazione
- La funzione riceve in input due puntatori a `double`:

`double* par` serve per passare l'array dei coefficienti

`double* x` è l'indirizzo di un array in cui salvare le soluzioni dell'equazione

# Esercizi

---

- **Esercizio 5:** Rifare l'esercizio su media/varianza realizzando un'unica funzione che le calcoli entrambe (fare uso di puntatori/referenze)
- **Esercizio 6:** Scrivere una funzione che, dato un array di  $N$  interi casuali, lo ordini dal più grande al più piccolo (suggerimento: creare un array specificando voi alcuni numeri a caso per testare che l'ordinamento funzioni). Il prototipo della funzione deve essere:

```
void SortArray(double* myArray, int dim);
```

- **Esercizio 7:** Riscrivere la `creaVettore(...)` senza fare uso dell'istruzione `return` (il passaggio degli argomenti deve essere leggermente modificato)

# Come passare per referenza un array ad una funzione

```
int arr[] = {1, 2, 3}; // Definisce un array di 3 elementi di tipo int
```

```
int* ptr = arr; // Definisce un puntatore ad int
```

```
int (&arr_ref)[3] = arr; // Definisce una referenza all'array
```

- Quindi per passare `arr` per referenza ad una funzione si può scrivere:

```
void funzione(int (&var)[3], ...)
```

```
{
```

```
    ...
```

```
    var[0] = 100;
```

```
    ...
```

```
}
```

# *Exception handling (“it’s easier to ask for forgiveness than permission”)*

```
#include <iostream>

bool testThrow (bool var)
{
    if (var != false)
        throw "Houston, we have a problem ... ";
    return var;
}

int main (int argc, char** argv)
{
    bool output;
    bool input = true;
    try
    {
        output = testThrow(input);
    }
    catch (const char* msg)
    {
        std::cout << "Exception : " << msg
                << std::endl;
        return 1;
    }
    std::cout << "Houston, we are all " << output << "k" << std::endl;
    return 0;
}
```

Il linguaggio di programmazione C++ mette a disposizione del programmatore un sistema di messaggistica per la gestione delle eccezioni, cioè per la gestione delle situazioni inaspettate  
Ecco qui un esempio ...

**Esercizio:** Riscrivere il programma per la risoluzione dell’equazione di secondo grado usando il seguente prototipo:  
**void solve2ndDegree(double\* par, double\* x);**  
e usando **throw / try&catch** per gestire il caso di determinante negativo