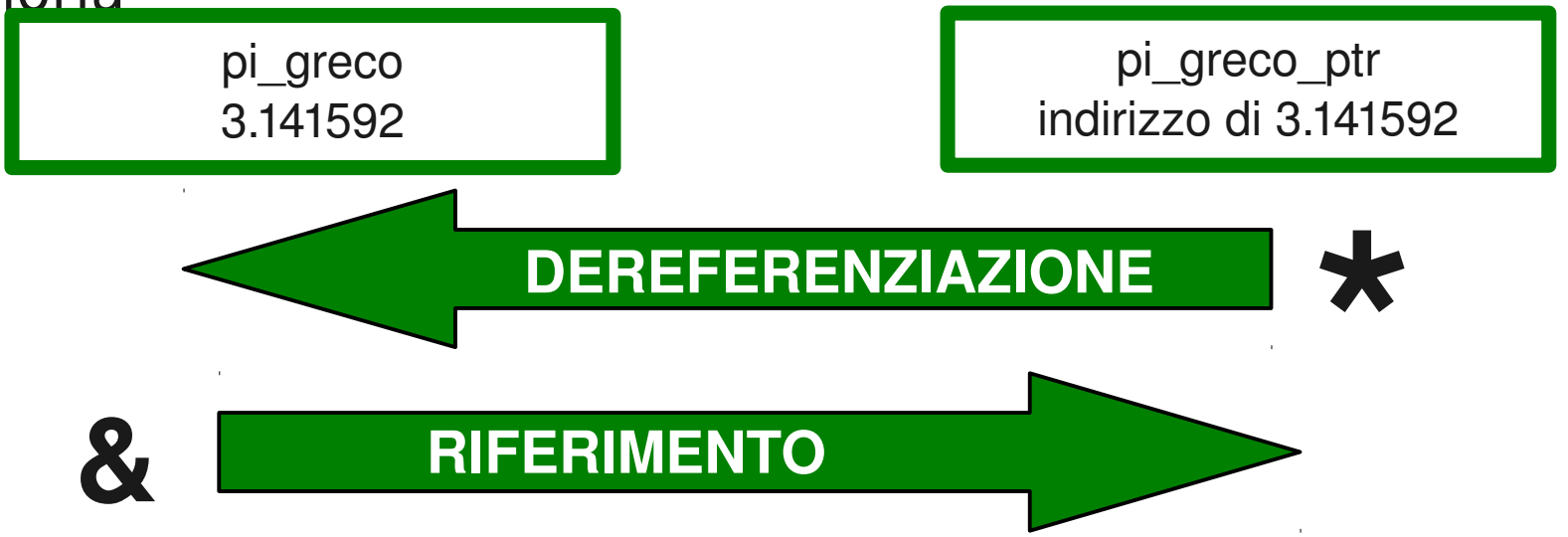
Introduzione al C++ (continua)

I puntatori

- Un **puntatore** è una variabile che contiene un indirizzo di memoria




```
double pi_greco = 3.1415 ;

pi_greco_ptr = &pi_greco ;
```

```
double * pi_greco_ptr ;

std::cout << *pi_greco_ptr << "\n" ;
```

- se **x** contiene l'indirizzo di **y**, si dice che x *punta* ad y

Gli operatori * e &

- Con i puntatori si utilizzano due operatori speciali, ***** e **&**.
- L'**operatore &** ritorna l'indirizzo della variabile che precede. Tradotto verbalmente, suonerebbe come *"l'indirizzo di"*.

```
int value;
int *p;

p = &value;
```

  - Scrivo nella variabile puntatore *p* il valore dell'indirizzo della variabile *value*
  - Verbalmente *"p riceve l'indirizzo di value"*

- L'**operatore** * ritorna il valore della variabile che si trova all'indirizzo specificato dal suo operando. Verbalmente *"all'indirizzo"*.

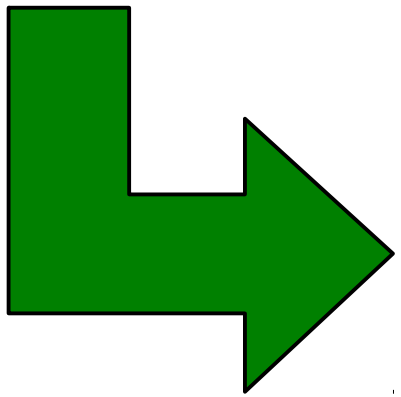
```
int value;
int num;
int *p;

value = 1000;
p = &value;
num = *p;
```

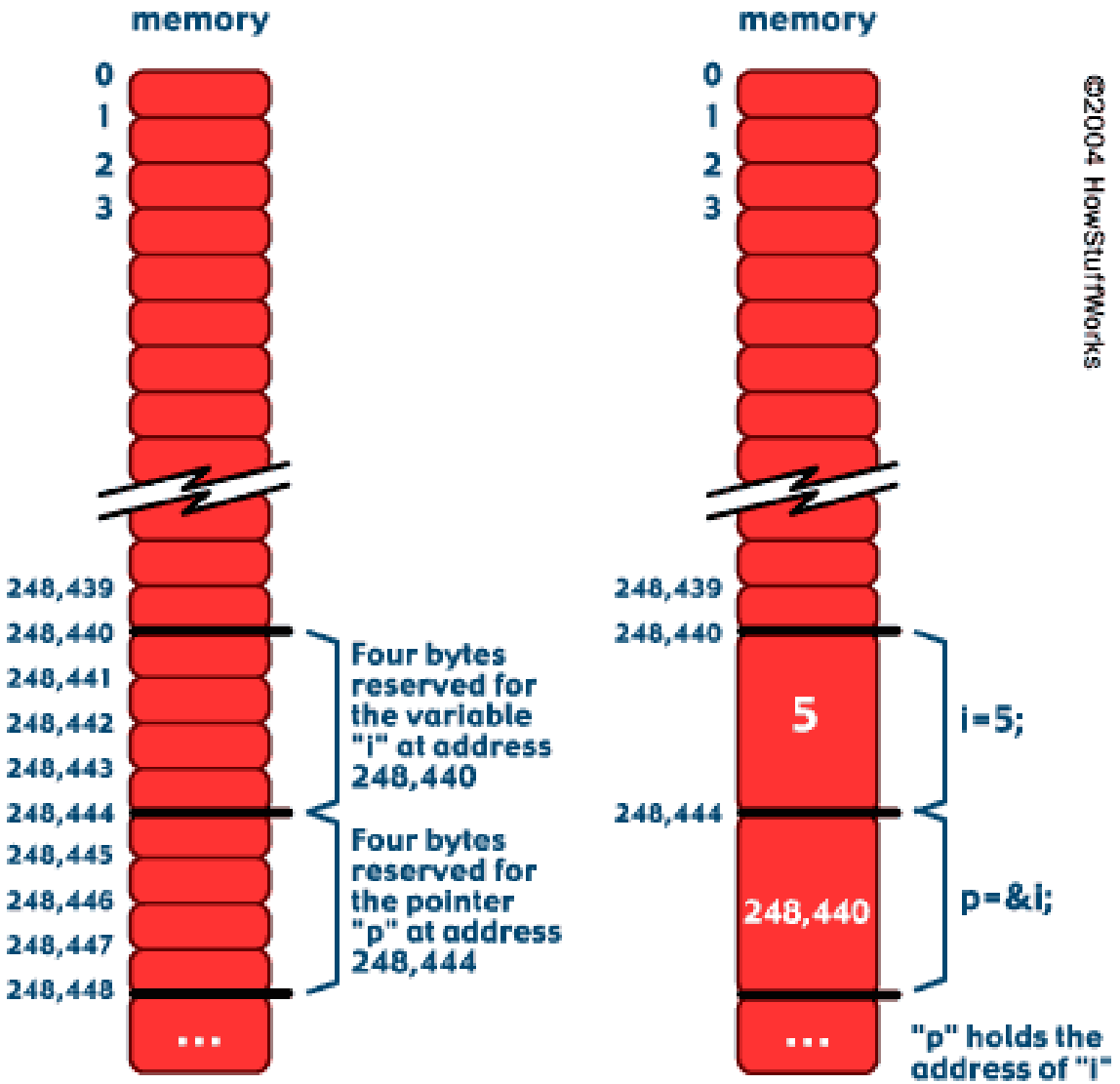
  - *"num riceve il valore che si trova all'indirizzo puntato da p"*
  - Ora il valore di num è 1000

Puntatori e RAM

Vediamo che cosa accade nella RAM del calcolatore

```
int i ;
i = 5 ;
int * p ;
p = &i ;
```





Puntatori e referenze

- NB: Anche i puntatori sono variabili e possono cambiare valore

```
double pi_greco = 3.1415 ;
double * altro_pointer ;
altro_pointer = &pi_greco ;
double nepero = 2.7183 ;
altro_pointer = &nepero ;
```

- Un puntatore si può creare senza assegnargli un valore
- Il valore del puntatore è l'indirizzo di memoria della variabile alla quale punta

- Le referenze invece si comportano come puntatori, ma sono costruite su una variabile specifica e rimangono vincolate ad essa

```
double pi_greco = 3.1415 ;
double & pi_greco_ref = pi_greco ;
std::cout << pi_greco_ref << "\n" ;
```

- Una referenza si crea a partire da una variabile esistente
- Si utilizza come una variabile e si comporta come un puntatore

Gestione della memoria

- Solitamente il C++ decide come gestire la memoria
- Se vogliamo farlo noi possiamo utilizzare l'operatore **new**

```
double * pointer = new double (1.5) ;
std::cout << *pointer << std::endl ;
delete pointer ;
```

Il valore 1.5 viene assegnato alla variabile a cui punta il puntatore

Dopo averla utilizzata, dobbiamo liberare la memoria con **delete**

Il puntatore e' l'unico legame con la varia

- Si possono creare anche vettori con **new**
- Questo ci permette di scegliere **runtime** la dimensione del vettor

```
double * array = new double [10] ;
for (int i=0 ; i<10 ; ++i)
  {
    array[i] = 0.31 * i ;
  }
delete [] array ;
```

Il valore degli elementi va assegnato dopo la definizione

Anche in questo caso bisogna chiamare il **delete[]**

UN ARRAY E' UN PUNTATORE!

I puntatori - esercizi

- **Esercizio 1**: Scrivete un programma che assegni il valore di una variabile ad un'altra utilizzando un puntatore. Fatevi inoltre stampare a terminale i valori e gli indirizzi di ogni variabile prima e dopo l'assegnazione.

- **Esercizio 2**: Dichiarate un puntatore e poi cercate di assegnargli direttamente un valore numerico. Cosa succede? Perchè?

- **Esercizio 3**: Utilizzate **new** e **delete** per creare e distruggere una variabile double ed un array di double

Le funzioni

- Serie circoscritte di istruzioni possono essere isolate in funzioni

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

- La funzione si definisce prima del programma principale (main)
- Può avere diversi oggetti come input ed (al più) un solo oggetto output
- All'interno del programma principale viene chiamata secondo il prototipo dichiarato

```
double valore_iniziale = 4 ;
double valore_finale = raddoppia (valore_iniziale) ;
std::cout << valore_iniziale
          << " x 2 = "
          << valore_finale << "\n" ;
```

Il passaggio di argomenti

- La stessa funzione può essere implementata in diversi modi e le variabili possono esserle passate in diversi modi

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

```
double raddoppiaReference (double & input)
{
  input = input * 2 ;
  return input ;
}
```

```
double raddoppia2 (double input)
{
  input = input * 2 ;
  return input ;
}
```

```
double raddoppiaPointer (double * input)
{
  *input = *input * 2 ;
  return *input ;
}
```

```
raddoppia:          4 x 2 = 8
raddoppia2:         4 x 2 = 8
raddoppiaPointer:   8 x 2 = 8
raddoppiaReference: 8 x 2 = 8
```

- Non tutti gli output sono uguali, come mai?

Divisione in diversi file

- Per evitare di avere programmi troppo lunghi e per semplificare la vita al compilatore, di solito le funzioni sono impacchettate in librerie

```
#ifndef esempio04_h
#define esempio04_h

double raddobbia (double input) ;

#endif
```

esempio04.h

Contiene le definizioni di variabili e funzioni (i prototipi)

Le istruzioni **#ifndef, #define, #endif** servono per evitare di duplicare le definizioni se il file viene usato più volte

```
#include "esempio04.h"

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

esempio04.cc

Contiene l'implementazione delle funzioni

Conosce il prototipo da esempio04.h attraverso l'istruzione **#include** (che incolla il contenuto di esempio04.h dove è chiamata)

Uso nel programma principale

- Nel programma principale, le funzioni definite nelle librerie sono utilizzate senza bisogno di implementarle

```
#include <iostream>
#include "esempio04.h"

int main (int numArg, char *listArg[])
{
        double valore_iniziale = 4;
        double valore_finale = raddoppia(valore_iniziale);
        std::cout << "Raddoppia: "
                  << valore_iniziale
                  << " x 2 = "
                  << valore_finale << "\n";
        return 0;
}
```

La grammatica è nota dall'inclusione di esempio06.h

L'implementazione viene linkata automaticamente dal c++, che viene chiamato così:

```
$ c++ -o esempio04 esempio04.cc esempio04.cpp
```

Le stringhe in C e C++

- Le parole sono gestite da vettori di lettere in C

```
char parola[12] = "pippo" ;
std::cout << parola << std::endl ;
int num3 = 5 ;
sprintf (parola,"pippo_%d", num3) ;
std::cout << parola << std::endl ;
```

La dimensione della stringa deve essere scelta con cautela

La funzione **sprintf** permette di cambiare il contenuto della stringa (comprese altre variabili)

- In C++ esiste un tipo dedicato, std::string
- Per questo è necessario includere la libreria: `#include<string>`

```
std::string parolaBis ;
parolaBis = "pippoBis" ;
std::cout << parolaBis << std::endl ;
parolaBis += "_aggiungo_" ;
std::cout << parolaBis << std::endl ;
```

Non è necessario scegliere la dimensione della stringa

Si possono aggiungere contenuti alla stringa con l'operatore +

Esercizi

- **Esercizio 4**: Scrivere una funzione che calcoli il fattoriale di un numero intero non negativo
- **Esercizio 5**: Scrivere la funzione fattoriale in modo ricorsivo, cioè facendo in modo che la funzione che calcola il fattoriale chiami se stessa

Logica Booleana
ed operatori bit a bit
(BITWISE OPERATORS)

Operatori Logici

Logical Operators	
Operator	**Meaning**
&&	AND
||	OR
!	NOT

- Una variabile **bool** può assumere due valori: true o false.
- Il C++ converte automaticamente *true* in 1 e *false* in 0.

```
int main ()
{
  bool p, q;
  std::cout<<"Inserisci P (0 o 1):";
  std::cin>>p;
  std::cout<<"Inserisci Q (0 o 1):";
  std::cin>>q;

  std::cout<<"P AND Q: "<<(p&&q)<<std::endl;
  std::cout<<"P OR Q: "<<(p||q)<<std::endl;
  std::cout<<"P XOR Q: "<<XOR(p,q)<<std::endl;

  return 0;
}
```

```
bool XOR(bool a, bool b)
{
  return (a||b) && !(a&&b);
}
```

Operatori bit a bit in C++

- Possibilità di operare direttamente sui singoli bit all'interno di un *byte* o un *word*.
- È possibile modificare i valori dei bit usando le operazioni dell'algebra booleana.

Operator	Action
&	AND
|	OR
^	exclusive OR (XOR)
~	one's complement (NOT)
>>	shift right
<<	shift left

AND

```
  1101 0011
& 1010 1010
-----------
  1000 0010
```

OR

```
  1101 0011
| 1010 1010
-----------
  1111 1011
```

XOR

```
  0 1 1 1 1 1 1 1
^ 1 0 1 1 1 0 0 1
---------------
  1 1 0 0 0 1 1 0
```

Rappresentazione binaria

```
int n, m;
int nb[32], mb[32];

std::cout << " Inserisci due numeri: ";
std::cin >> n >> m;
std::cout << " n: " << n << " m: " << m << std::endl;
for (int i=0; i<32; i++){
        if((n & (int) pow(2,i))) {
                nb[i] = 1;
        } else {
                nb[i] = 0;
        }
        if((m & (int) pow(2,i))) {
                mb[i] = 1;
        } else {
                mb[i] = 0;
        }
}
```

```
std::cout << " Bitwise operators " << std::endl;
std::cout << " Rappresentazione binaria di n: " << std::endl;
for (int i=31; i>=0; i--) {
        std::cout << nb[i];
}
std::cout << std::endl;
std::cout << " Rappresentazione binaria di m: " << std::endl;
for (int i=31; i>=0; i--) {
        std::cout << mb[i];
}
```

- *Gli operatori booleani **non** devono essere utilizzati con variabili di tipo float o double.*
- *In binario, le potenze di 2 si rappresentano con tutte le cifre uguali a 0 eccetto una cifra che è 1, con posizione dipendente dall'esponente.*

Bitwise operations

BITWISE OPERATION: AND

```
t = n & m;
std::cout << " n & m " << t << std::endl;
for (int i=0; i<32; i++){
        if((t & (int) pow(2,i))) {
                tb[i] = 1;
        } else {
                tb[i] = 0;
        }
}
```

OUTPUT

```
 Inserisci due numeri: 10 45
 n: 10 m: 45
 Bitwise operators
 Rappresentazione binaria di n:
00000000000000000000000000001010
 Rappresentazione binaria di m:
00000000000000000000000000101101

 n & m 8
 Rappresentazione binaria di (n AND m):
00000000000000000000000000001000

 n | m 47
 Rappresentazione binaria di (n OR m):
00000000000000000000000000101111

 n ^ m 39
 Rappresentazione binaria di (n XOR m):
00000000000000000000000000100111
```

Complemento a 1 (NOT)

- L'operatore bitwise NOT (~) inverte il valore di tutti i bit.
- Quest'operazione viene chiamata **complemento a 1**.
- I numeri negativi vengono rappresentati eseguendo la seguente operazione:

  -n = (~n) + 1

Esercizi

Esercizi

- **Esercizio 6:** Dati due numeri interi in input, stampare a schermo la loro rappresentazione binaria
- **Esercizio 7:** Eseguire le operazioni bit a bit &, |, ^ e stampare i risultati in binario.
- **Esercizio 8:** Applicare l'operatore bitwise NOT ad un numero di tipo int n e verificare che:

  (~n) +1 = -n