ROMA TRE
UNIVERSITÀ DEGLI STUDI

Tutorato di Laboratorio di Calcolo

A.A. 2013/2014

Ilaria Carlomagno

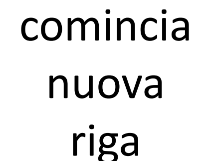
ROMA TRE
UNIVERSITÀ DEGLI STUDI

Input e Output

- Gestire la comunicazione tra utente e computer
- In C++ si usano due comandi: `cin` e `cout`.
- All'inizio del programma va incluso il pacchetto <iostream>
- Sintassi:

```
cout << "Questo testo verrà visto dall'utente" <<endl;
cin >> variabile;
```

Esempio




```
#include <iostream>
int main () {
cout<< "inserire il valore relativo a x" << endl;
cin >> x;
(…)
}
```

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Confronto tra variabili

- Condizioni sulle variabili: possono essere imposte se esiste un modo di verificare confronti tra più valori.

Condizione di confronto	Simbolo
uguaglianza, disuguaglianza	== , !=
maggiore, minore	> , <
Maggiore uguale, minore uguale	>= , <=

Attenzione alla differenza tra == e =

Il simbolo " = " assegna alla variabile di sinistra il valore che si trova a destra.

Il simbolo " == " confronta i due valori.

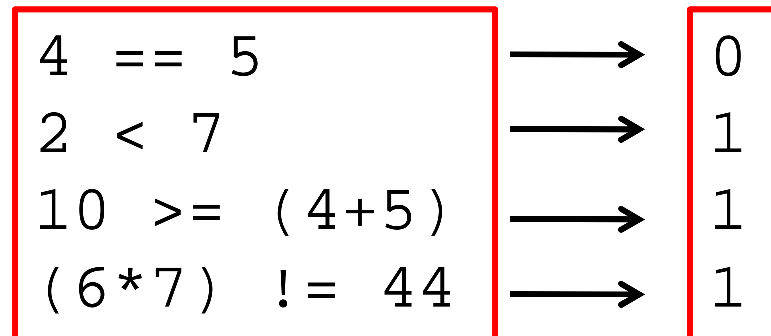
ROMA TRE
UNIVERSITÀ DEGLI STUDI

Logica Booleana

- Il computer conosce solo due valori: 0 e 1.

Per convenzione, si usa 0 per indicare FALSO e 1 per indicare VERO.

Condizione di confronto	Simbolo
uguaglianza, disuguaglianza	== , !=
maggiore, minore	> , <
Maggiore uguale, minore uguale	>= , <=

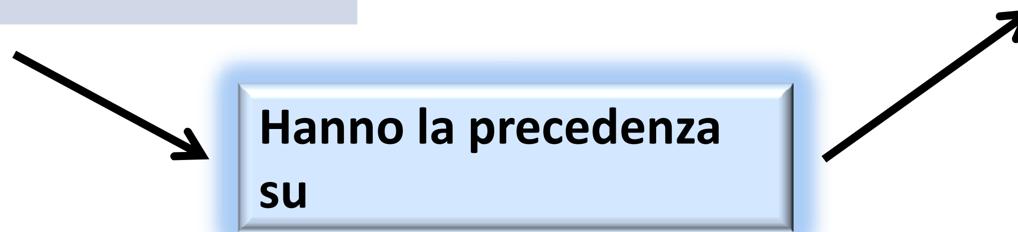


Con questi comandi, il computer è in grado di riconoscere le relazioni tra due variabili.

Imposizione di più condizioni

- Condizioni sulle variabili: possono essere imposte se esiste un modo di verificare confronti tra più valori.

Condizione di confronto	Simbolo
uguaglianza, disuguaglianza	== , !=
maggiore, minore	> , <
Maggiore uguale, minore uguale	>= , <=



Nome	Simbolo	Input
AND	&&	2
OR	||	2
NOT	!	1

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Esempio

- Voglio che il mio programma riconosca i casi in cui la variabile x è compresa tra 0 e 10.

```
x>=0 && x <= 10
```

Voglio che le due condizioni siano verificate CONTEMPORANEAMENTE

- Oppure...

```
x<0 || x > 10
```

Mi basta che anche una sola delle due sia verificata affinché il programma si “accorga” dell’errore.

ROMA TRE UNIVERSITÀ DEGLI STUDI

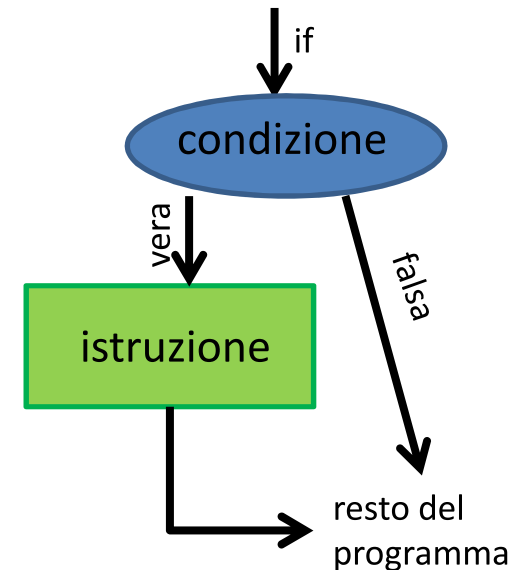
Istruzioni condizionate: **if**

Esegue un'istruzione se e solo se è verificata una condizione.

- Sintassi: if (condizione) { istruzione }

Esempio:

```
if( a<0 ) {
        cout << "a è negativo" << endl ;
}
```

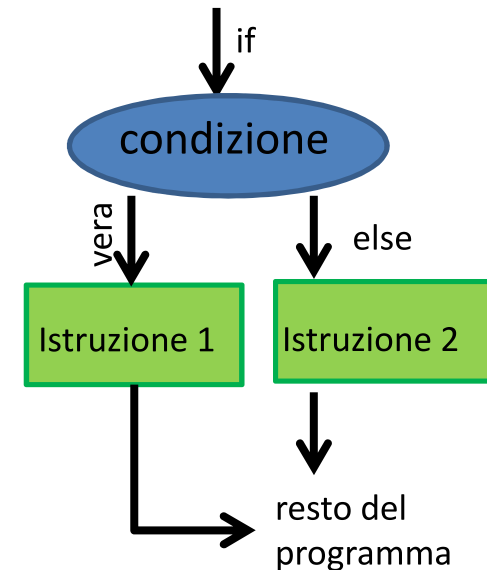


Alternative: **if…else if…else**

- L'**if** può essere usato anche quando dobbiamo eseguire diverse istruzioni in diversi casi.
- Sintassi: else { istruzione }

ESEMPIO: due alternative

```
if( a<0 ) {
        cout << "a è negativo" << endl ;
} else {
        cout << " a è positivo o nullo " << endl;
        }
```



L'**else** non ha bisogno di condizioni perché l'istruzione nelle sue **{ }** viene eseguita in tutti gli altri casi che non rientrano nella condizione dell'**if**. **else** può esser usato solo dopo **if**, MAI da solo!

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Alternative: **if…else if…else**

- **else if** ha la stessa sintassi di **if** e, come **else**, non può essere usato da solo.
- Sintassi: if else (condizione) { istruzione }

ESEMPIO: 3 (o più) alternative

```
if( a<0 ) {
            cout << "a è negativo" << endl ;
      } else if ( a==0 ) {
            cout << " a è nullo " << endl;
      } else {
            cout << " a è negativo " << endl;
            }
```

else esegue l'istruzione in tutti i casi non previsti dagli altri **if** o **else if**.

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Cicli di funzioni: **for**

Esegue un'istruzione un certo numero di volte, specificato da una condizione.

- Sintassi: for (partenza; condizione; passo) { istruzione }

Esempio:

```
cout << "Contiamo fino a 10!" << endl ;
for( i=1; i<= 10; i++ ) {
                    cout << i << endl ;
                    }
```

Attenzione: **for** è spesso usato per le sommatorie ma sul simbolo Σ indichiamo il punto in cui fermarci (traguardo), nel **for** dobbiamo specificare fino a quando continuare (percorso).

$$\sum_{I=1}^{10}$$

ROMA TRE
UNIVERSITÀ DEGLI STUDI

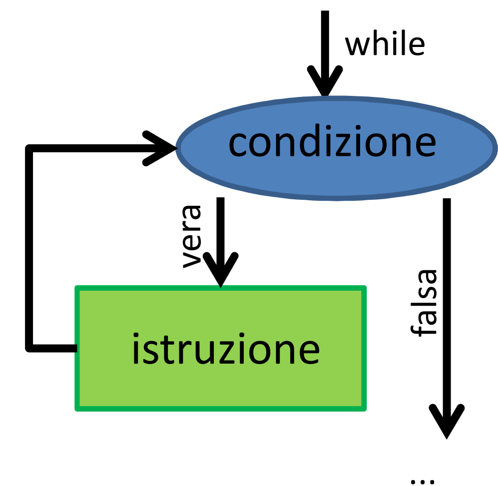
Cicli di funzioni: **while**

Esegue un'istruzione **finché** rimane verificata una certa condizione

- Sintassi: while (condizione) { istruzione }

Esempio:

```
while ( a == 0 ) {
                cout << "Inserito valore nullo. Riprova.";
                cin>>a;
                }
```



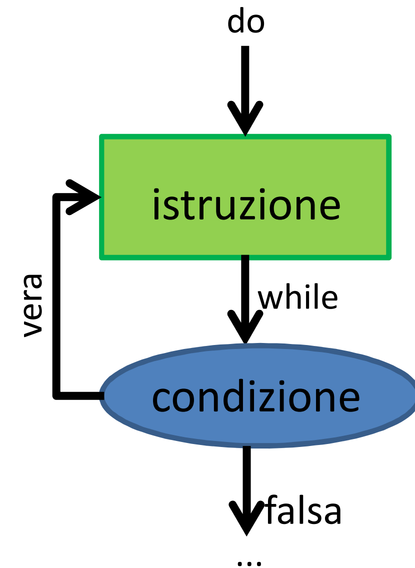
Attenzione: nell'istruzione del **while** dobbiamo ricordarci di includere un comando che agisca sulla variabile usata nella condizione, altrimenti non usciremo mai dal ciclo!!!

Cicli di funzioni: **do…while**

Esegue un'istruzione, poi verifica una condizione e continua ad eseguire l'istruzione **finché** rimane verificata la condizione

- Sintassi: do { istruzione } while (condizione)
  Esempio:

```
do {      cout << "a è nullo!Inserire nuovo valore!";
          cin>>a;
          } while ( a == 0 )
```



Equivalente al **while** MA l'istruzione viene <u>sempre</u> eseguita almeno una volta.
(Ricordatevi di includere la condizione di uscita, altrimenti LOOP!)

ROMA TRE
UNIVERSITÀ DEGLI STUDI

I vettori

Sono una serie di elementi, tutti dello stesso tipo, memorizzati in posizioni di memoria contigue.

Si dichiarano specificando il tipo e la dimensione:

```
tipo nome [dimensione]
int vettore_di_interi [5];
```

Si utilizzano i singoli elementi, specificando il vettore di appartenenza e la posizione.

Il primo elemento è il numero 0, quindi l'ultimo elemento di un vettore con n componenti, sarà l'(n-1)esimo elemento. Se voglio che il 3° elemento del vettore sia nullo, scriverò:

```
vettore_di_interi [2] = 0;
```

Le operazioni vengono fatte sui singoli elementi, spesso ricorrendo a cicli.

Dimensione e numero degli elementi sono INTERI!

I vettori (2)

L'inizializzazione di un vettore può esser fatta elemento per elemento

```
int vettore_di_interi [5];
vettore_di_interi = { 1, 2, 3, 4, 5 }
```

Oppure tramite un ciclo for:

```
for (i=0; i<5; i++){
                    vettore_di_interi [i] = i+1;
                    }
```

L'uso dei singoli elementi è analogo a una semplice variabile. Per stampare l'elemento 2 del vettore, basterà utilizzare la sintassi

```
cout<<vettore[1];
```

ROMA TRE UNIVERSITÀ DEGLI STUDI

Matrici (vettori multi-dim.)

Le matrici non sono altro che vettori a più dimensioni (vettori di vettori). Si inizializzano specificando tipo e dimensione:

```
double matrice [2] [3];
```

Analogamente ai vettori, il primo elemento della matrice (prima riga, prima colonna) si indica con:

```
Matrice [0] [0];
```

Come per i vettori, dimensione e numero degli elementi di una matrice sono INTERI!

ROMA TRE UNIVERSITÀ DEGLI STUDI

I puntatori

- Sono degli oggetti che rimandano ad altri. Fanno riferimento a posizioni di memoria, "indirizzi" in cui è scritta una certa informazione.
- Si inizializzano indicando il tipo, specificando il nome e aggiungendo un asterisco.

```
int *puntatore1;
int* puntatore2;
```

Posso usare i puntatori per riferirmi all'indirizzo di memoria di altre variabili.

```
int numero;
int* puntatore = &numero;
```

oppure

```
*puntatore = numero;
```

oppure

```
puntatore = &numero;
```

Vantaggi:
- Accesso ai dati velocizzato
- Allocazione dinamica della memoria durante l'esecuzione
- Condivisione dei dati tra diversi blocchi

I puntatori (2)

- Posso cambiare il valore della variabile “numero” in due modi:

```
numero = 0;
*puntatore = 0;
```

Senza * è un indirizzo (puntatore)

Con * è l’oggetto puntato (valore numerico)

Attenzione agli asterischi!

```
cout<<puntatore;
cout<<*puntatore
```

Stampano risultati diversi!!!

ROMA TRE UNIVERSITÀ DEGLI STUDI

Puntatori a vettori

E se volessi costruire un puntatore ad un vettore?
Vale la sintassi

```
int vett [5];
int *p;
p = vett;
```

Quando uso un puntatore a un vettore, il puntatore contiene l'indirizzo del primo elemento (elemento [0]) del vettore a cui punta.

Il puntatore è UN indirizzo, nei vettori, ogni dato ha il proprio (indirizzo). Possiamo però sfruttare il fatto che gli indirizzi degli elementi vettoriali sono contigui: se il primo elemento si trova all'indirizzo «p», il successivo si troverà all'indirizzo «p+1».

```
for (i=0; i<5; i++){
        cout<<vett[i]<< " corrisponde a "<<…;
```

Qualcosa che rimandi a *p

Scrivete un programma che contenga un vettore e un puntatore al vettore.
Stampate su schermo sia il puntatore sia il vettore.

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Puntatori a vettori

E se volessi costruire un puntatore ad un vettore?
Vale la sintassi

```
int vett [5];
int *p;
p = vett;
```

Quando uso un puntatore a un vettore, il puntatore contiene l'indirizzo del primo elemento (elemento [0]) del vettore a cui punta.

Il puntatore è UN indirizzo, nei vettori, ogni dato ha il proprio (indirizzo). Possiamo però sfruttare il fatto che gli indirizzi degli elementi vettoriali sono contigui: se il primo elemento si trova all'indirizzo «p», il successivo si troverà all'indirizzo «p+1».

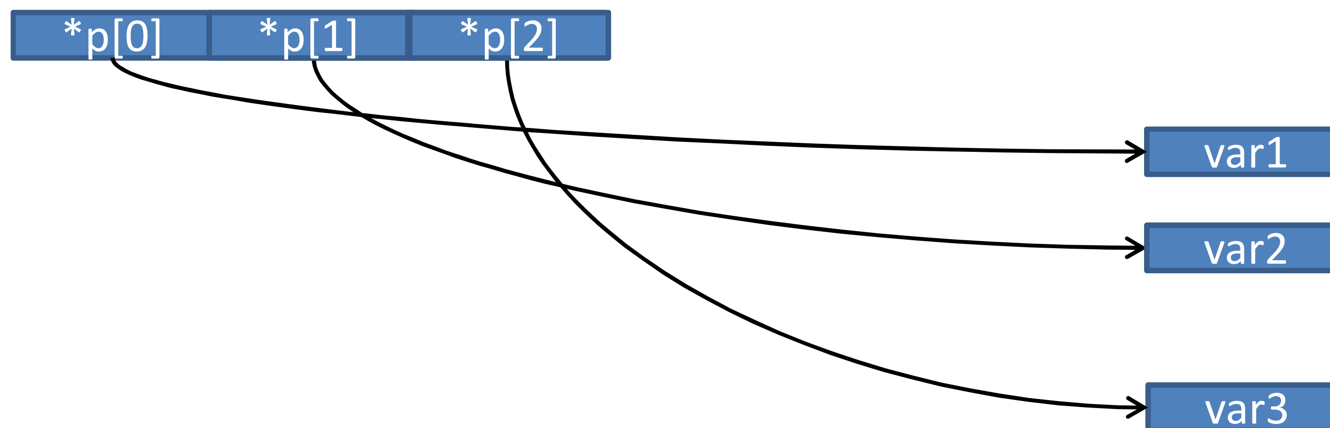
```
for (i=0; i<5; i++){
                    cout<<vett[i]<< " corrisponde a "<<*(p+i);
                    }
```

Vettori di puntatori

Posso definire un vettore di puntatori:

```
int var1, var2, var3;
int *p[3];
*p[0] = var1;
*p[1] = var2;
*p[2] = var3;
```

e utilizzarlo per scorrere i valori…

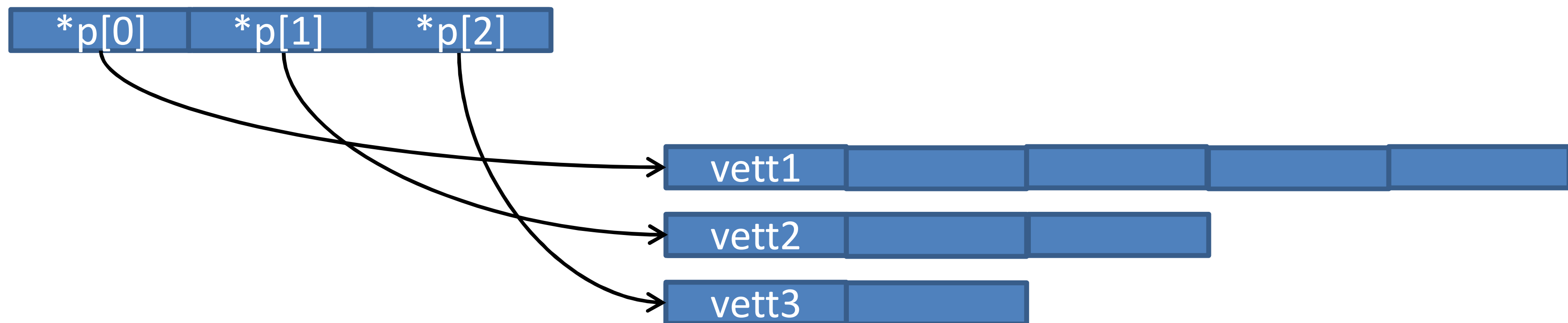


Vettori di puntatori

Posso definire un vettore di puntatori:

```
int vett1[5], vett2 [3], vett3 [2];
int *p[3];
*p[0] = vett1;
*p[1] = vett2;
*p[2] = vett3;
```

e utilizzarlo per scorrere i valori…comodo nel caso di vettori!
Scorrendo gli elementi di p, passo da un vettore all'altro. Stampando gli oggetti agli indirizzi (p[0]+i), scorro il primo vettore, (p[1]+i) il secondo e così via…



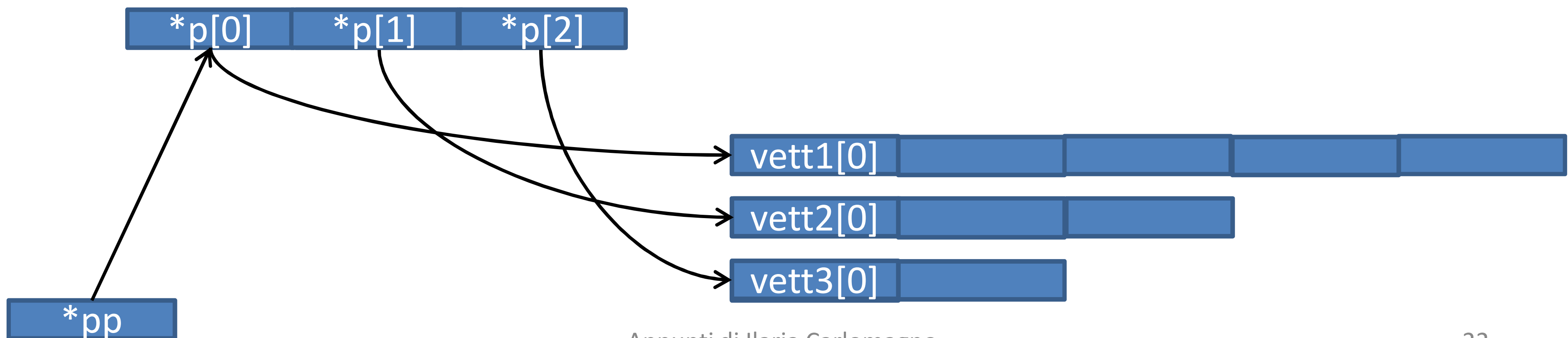
ROMA TRE UNIVERSITÀ DEGLI STUDI

Puntatori a puntatori

Contengono l'indirizzo dei puntatori. Es:

```
int vett1[5], vett2[3], vett3[2];
int *p[3];
p[0] = vett1;
p[1] = vett2;
p[2] = vett3;
int **pp;
pp = p;
```

p contiene gli indirizzi dei primi elementi di ciascun vettore.
pp contiene l'indirizzo del primo elemento di p.



Funzioni

Le funzioni permettono di strutturare i programmi in segmenti di codice che eseguono singole istruzioni

```
#include <iostream>
using namespace std;

int somma(int a, int b) {
        int r;
        r = a + b;
        return r;
        }

int main () {
        int z;
        z = somma(5,3);
        cout << "Il risultato
è" << z;
}
```

Se ho due o più funzioni, l'ordine con cui le dichiaro non conta: la funzione principale è il «main» ed è l'unica che viene cercata ed eseguita automaticamente. Le altre funzioni non vengono eseguite se non sono richiamate nel main.

L'importante è dichiarare le funzioni prima del main, cioè prima che queste vengano usate: proprio come si fa per le variabili!

Funzioni (2)

Le funzioni permettono di strutturare i programmi in segmenti di codice che eseguono singole istruzioni

```
#include <iostream>
using namespace std;

int somma(int a, int b) {
        int r;
        r = a + b;
        return r;
}

int main () {
        int z;
        z = somma(5,3);
        cout << "Il risultato
è" << z;
}
```

Questo vuol dire che il programma inizia dal main, una volta arrivato alla funzione sconosciuta «somma», il compilatore va a cercarla.
Una volta trovata, la esegue e poi prosegue nel main.

La funzione richiede due parametri di input: `a` e `b`.
Sono i valori che le vengono passati dal main.

Il `return` ferma l'esecuzione della funzione «somma» e fa tornare all'esecuzione del main.

Il valore relativo al return, viene assegnato alla variabile legata alla funzione «somma» .

È come se la funzione fosse rimpiazzata dal valore che essa stessa dà in uscita.

Funzioni (3)

Le funzioni possono essere chiamate ripetutamente durante un programma.

```
int somma(int a, int b) {
        int r;
        r = a + b;
        return r;
}

int main () {
        int z, x, y;
        z = somma(5,3);
        cout << "Il risultato è" << z;
        cout << "2 + 3 fa "<< somma (2,3);
        cout << " La somma di x e y inseriti fa " << somma
(x,y);
}
```

Posso usarle con valori numerici, con variabili e con combinazioni dei due (2+x ecc.).

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Funzioni di tipo void

Quando la funzione non fornisce alcun risultato come output, dobbiamo considerarla come funzione void.

```
void stampa () {
        cout << "Sono una funzione!";
                }

int main () {
        stampa();
                }
```

Quando la funzione non richiede parametri di input, le parentesi che seguono il nome della funzione vanno lasciate vuote…ma ricordatevi di metterle SEMPRE dopo il nome della funzione, altrimenti quest'ultima non verrà chiamata!

ROMA TRE UNIVERSITÀ DEGLI STUDI

Argomenti di default

Possiamo definire dei valori di default per alcune variabili in modo da poter scegliere se specificarne il valore o no, quando la funzione viene chiamata.

```
int dividi (int a, int b=2) {
        int r;
        r = a/b;
        return (r);
        }

int main () {
        cout << dividi (12) << endl;
        cout << dividi (20,4) << endl;
        return 0;
        }
```

Nella prima chiamata alla funzione specifico solo un valore, il compilatore assegna dunque alla seconda variabile, il valore di default specificato nella dichiarazione della funzione.

Nella seconda chiamata, specifico entrambi i valori e la funzione lavora con a = 20 e b = 4.

Senza i simboli «&» nell'argomento della funzione, x, y e z non verrebbero modificati!!!

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Argomenti: variabili o indirizzi

Quando passiamo ad una funzione delle variabili come argomento, inizializziamo gli argomenti della funzione con il valore della variabile. Quando inizia l'esecuzione della funzione, lavoriamo con i suoi argomenti e non andiamo a modificare le variabili usate nel main.

```
void duplica(int& a, int& b, int& c){
        a = a*2;
        b = b*2;
        c = c*2;
        }

int main () {
        int x=1, y=3, z=7;
        duplica (x, y, z);
        cout<<"x="<<x<<",y="<<y<<",z="<<z;
        return 0;
        }
```

Se invece uso i puntatori, mentre eseguo la funzione,
Posso andare a modificare il valore delle variabili esterne alla funzione.
Nell'esempio, quando eseguo il cout, vedo l'ultimo valore assegnato a x e y, cioè quello della funzione «duplica».

Usando gli indirizzi, faccio lavorare la funzione direttamente sulle variabili, anziché su loro copie.

Senza i simboli «&» nell'argomento della funzione, x, y e z non verrebbero modificati!!!

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Prototipi di funzioni

Abbiamo visto come sia necessario dichiarare la funzione prima di chiamarla. In realtà, basta dichiarare un **prototipo**, prima della chiamata; il resto della funzione può esser definito altrove.

```
double funzione (int, double);
```

```
double funzione (int a, double b);
```

La dichiarazione di un prototipo richiede di specificare il tipo di argomenti e di return e il nome della funzione.
Il nome delle variabili è opzionale.

Scrivere un programma che usi due funzioni void che stampino su schermo «pari» o «dispari» a seconda del numero inserito dall'utente.

Specificare il nome non è necessario ma aumenta la leggibilità del programma: l'ordine in cui dichiariamo i parametri è importante!!!

Funzioni ricorsive

Mentre eseguo una funzione, posso richiamarne un'altra…anche la stessa!

```
long factorial (long a) {
        if (a > 1) return (a * factorial (a-1));
        else  return 1;
        }

 int main () {
        long number = 9;
        cout << number << "! = " << factorial (number);
        return 0;
        }
```

Overload di funzioni

Posso definire due funzioni con lo stesso nome, a patto che il numero di parametri usati o il loro tipo non siano coincidenti.

```
int operazione (int a, int b) {
        return (a*b);
}

double operazione (double a, double b){
        return (a/b);
        }

int main () {
        int x=5,y=2;
        double n=5.0,m=2.0;
        cout << operazione (x,y);
        cout<<endl;
        cout << operazione (n,m)
        cout<<endl;
        return 0;
        }
```

Le funzioni «operazione» moltiplicano due numeri interi e dividono due numeri razionali.

La prima chiamata è con argomenti interi, quindi avrò una moltiplicazione;
la seconda chiamata è con numeri double, quindi avrò una divisione.

Per utilizzare l'overload il tipo di ALMENO uno dei parametri deve essere diverso dalla precedente dichiarazione della funzione.

Template di funzioni

Quando uso l'overload di una funzione per farla operare su argomenti di tipo diverso ma con lo stesso blocco di istruzioni (corpo), posso usare il template:

```
int sum (int a, int b) {
          return a+b;
          }
double sum(double a,double b){
          return a+b;
          }

 int main () {
          cout<<sum(10,20)<<'\n';
          cout<<sum(1.0,1.5)<<'\n';
          return 0;
          }
```

```
template <class T>
T sum (T a, T b) {
   T result;
   result = a + b;
   return result;
          }
int main () {
 int i=5,j=6,k;
 double f=2.0;
 double g=0.5, h;
 k=sum<int>(i,j);
 h=sum<double>(f,g);
 cout << k << '\n';
 cout << h << '\n';
  return 0;
  }
```

```
template <class T>
T sum (T a, T b) {
   T result;
   result = a + b;
   return result;
          }

int main () {
  cout << sum (1,2);
  cout << sum (1,5,1);
  return 0;
          }
```

Classi

Sono delle strutture di dati che comprendono sia variabili, sia operazioni su di esse. I loro costituenti sono gli «oggetti». Se gli oggetti fossero delle variabili, potremmo pensare alla classe come al tipo.

```
class nome_classe {
 private:
         oggetto1;
 public:
         oggetto2;
}
```

Definizione di una classe

```
class Rettangolo {
 private:
         int base, altezza;
 public:
         void set_values (int, int);
         int area (void);
};
```

Definizione della classe «Rettangolo» e di un suo oggetto «rett». Contiene due variabili intere e due metodi che fissano i valori di base e altezza e li usano per calcolare l'area.

`private`: è accessibile solo dai membri della stessa classe;
`protected`: accessibile anche dalle classi figlie;
`public`: accessibile da qualunque parte del programma.

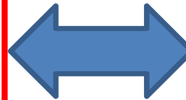
} (attributi)
(metodi)

ROMA TRE UNIVERSITÀ DEGLI STUDI

Classi (2)

Se ci dimentichiamo di specificare il tipo di accesso ai membri, il compilatore considererà `private` come default.

```
class Rettangolo {
 private:
         int base, altezza;
 public:
         void set_values (int, int);
         int area (void);
};
```



```
class Rettangolo {
         int base, altezza;
 public:
         void set_values (int, int);
         int area (void);
};
```

Dopo aver dichiarato la classe e creato almeno un suo oggetto, possiamo accedere e usare i membri pubblici della classe come normali funzioni/variabili, usando la sintassi:
`oggetto.metodo(parametri);`

```
rett.set_values (5,2);
area_calcolata = rett.area();
```

Le uniche parti a cui non posso accedere sono quelle private, in questo caso, base e altezza.

Al metodo `set_values` passiamo, come richiede la sua definizione, 2 parametri interi.
Il metodo area, non richiede parametri, ma richiede comunque di aprire e chiudere le parentesi!

Classi (3)

Esempio di uso della classe rettangolo:

```
#include <iostream>
using namespace std;

class Rettangolo{
        int base, altezza;
public:
        void set_values (int,int);
        int area() {return base*altezza;}
                };

void Rettangolo::set_values (int x, int y) {
        base = x;
        altezza = y;
        }

int main () {
        Rettangolo rett;
        rett.set_values (3,4);
        cout << "area: " <<
        rett.area();
        return 0;
            }
```

1. Definizione della classe: contiene due variabili private e due metodi pubblici. Il primo è solo dichiarato, il secondo è anche definito.
2. Definizione del metodo `set_values`: quando invoco questa parte di programma, il computer associa alle variabili private `base` e `altezza` il valore dei parametri `x` e `y` che passo al metodo. **È indispensabile, se ho variabili private!**
3. Uso dell'oggetto `rett` nel `main`

Classi (3)

Esempio di uso della classe rettangolo:

```
#include <iostream>
using namespace std;

class Rettangolo{
        int base, altezza;
public:
        void set_values (int,int);
        int area() {return base*altezza;}
                };

void Rettangolo::set_values (int x, int y) {
        base = x;
        altezza = y;
        }

int main () {
        Rettangolo rett;
        rett.set_values (3,4);
        cout << "area: " <<
        rett.area();
        return 0;
            }
```

1. Creazione di un oggetto `rett` della classe `Rettangolo`
2. Uso del metodo `set_values` per assegnare una corrispondenza alle variabili private di base e altezza
3. Uso del metodo `area` per calcolare l'area.

Classi (4)

Esempio di uso della classe rettangolo:

```
#include <iostream>
using namespace std;

class Rettangolo{
        int base, altezza;
public:
        void set_values (int,int);
        int area() {return base*altezza;}
                };

void Rettangolo::set_values (int x, int y) {
        base = x;
        altezza = y;
        }

int main () {
        Rettangolo rett;
        rett.set_values (3,4);
        cout << "area: " <<
        rett.area();
        return 0;
            }
```

Attenzione alla punteggiatura!
L'operatore di «scope» :: si usa quando si vuole definire il membro di una classe al di fuori della classe stessa. Indica al compilatore che la funzione che si sta definendo appartiene alla classe `Rettangolo`, non è una funzione indipendente.

I metodi possono dunque essere direttamente definiti (come `area`) o possono essere dichiarati (come prototipi) nella classe di appartenenza per essere definiti più tardi.
E' utile definire immediatamente i metodi solo se sono abbastanza sintetici da non pregiudicare la leggibilità del programma.

Classi (5)

Sono delle strutture di dati che comprendono sia variabili, sia operazioni su di esse. I loro costituenti sono gli «oggetti». Se gli oggetti fossero delle variabili, potremmo pensare alla classe come al tipo.

```
class Rettangolo{
        int base, altezza;
public:
        void set_values (int,int);
        int area() {return base*altezza;}
                };

void Rettangolo::set_values (int x, int y) {
        base = x;
        altezza = y;
        }

int main () {
        Rettangolo rett;
        Rettangolo rett2;
        rett.set_values (3,4);
        rett2.set_values (10, 25);
        cout<<"area1: "<<rett.area()<<endl;
        cout<<"area2: "<<rett2.area()<<endl;
        return 0;
            }
```

Il punto di forza delle classi è la possibilità di definire e gestire più oggetti dello stesso tipo.
…Per esempio diversi rettangoli!
Creando un secondo rettangolo come oggetto della classe, posso far agire area anche su di esso…

Classi (6)

Sono delle strutture di dati che comprendono sia variabili, sia operazioni su di esse. I loro costituenti sono gli «oggetti». Se gli oggetti fossero delle variabili, potremmo pensare alla classe come al tipo.

```
class Rettangolo{
        int base, altezza;
public:
        void set_values (int,int);
        int area() {return base*altezza;}
                };

void Rettangolo::set_values (int x, int y) {
        base = x;
        altezza = y;
        }

int main () {
        Rettangolo rett;
        Rettangolo rett2;
        rett.set_values (3,4);
        rett2.set_values (10, 25);
        cout<<"area1: "<<rett.area()<<endl;
        cout<<"area2: "<<rett2.area()<<endl;
        return 0;
            }
```

Il punto di forza delle classi è la possibilità di definire e gestire più oggetti dello stesso tipo.
…Per esempio diversi rettangoli! Creando un secondo rettangolo come oggetto della classe, posso far agire area anche su di esso…

Vediamo ora come alleggerire il codice del `main`…

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Metodi utili: stampa

```
class Rettangolo {
          int base, altezza;
public:
          void set_values (int,int);
          int area() {return base*altezza;}
          void stampa(){
          cout<<'B= '<<base<<' H= ' <<
altezza<<endl;
          }
                    };

void Rettangolo::set_values (int x, int y) {
          base = x;
          altezza = y;
          }

int main () {
          Rettangolo rett;
          Rettangolo rett2;
          rett.set_values (3,4);
          rett2.set_values (10, 25);
          rett.stampa();
          rett2.stampa();
          return 0;
              }
```

Ho aggiunto alla classe `Rettangolo` un metodo che stampa base e altezza dell'oggetto definito.
In questo modo, nel `main`, posso stampare questi dati semplicemente invocando il metodo di stampa, senza inserire il `cout`!

Quando chiamate un metodo che non richiede parametri, non dimenticate le parentesi!

Metodi utili: stampa

```
class Rettangolo {
	int base, altezza;
public:
	void set_values (int,int);
	int area() {return base*altezza;}
	void stampa(){
	cout<<‘B= ‘<<base<<‘ H= ‘ <<
altezza<<endl;
	}
		};

void Rettangolo::set_values (int x, int y) {
	base = x;
	altezza = y;
	}

int main () {
	Rettangolo rett;
	Rettangolo rett2;
	rett.set_values (3,4);
	rett2.set_values (10, 25);
	rett.stampa();
	rett2.stampa();
	return 0;
	    }
```

```
class Rettangolo{
	int base, altezza;
public:
	void set_values (int,int);
	int area() {return base*altezza;}
	void stampa();
		};

void Rettangolo::set_values (int x, int y) {
	base = x;
	altezza = y;
	}
void Rettangolo::stampa() {
	 cout<<‘B= ‘<<base<<‘ H= ‘ <<
altezza<<endl;}

int main () {
	Rettangolo rett;
	Rettangolo rett2;
	rett.set_values (3,4);
	rett2.set_values (10, 25);
	rett.stampa();
	rett2.stampa();
	return 0;
	    }
```

I due codici sono equivalenti dal punto di vista dell'esecuzione, ciò che cambia è la leggerezza del codice: le parti «pesanti» sono state portate fuori della classe, nel riquadro a destra.

Metodi utili: get

```
class Rettangolo {
          int base, altezza;
public:
          void set_values (int,int);
          int area() {return base*altezza;};
          int get_base () {return base;};

void Rettangolo::set_values (int x, int y) {
          base = x;
          altezza = y;
          }

int main () {
          Rettangolo rett;
          Rettangolo rett2;
          rett.set_values (3,4);
          cout<<"Base = "<<rett.get_base();
          cout<<'Area rett='<<rett.area();
          return 0;
              }
```

Il metodo stampa, accessibile dal `main` perché pubblico, può accedere agli attributi privati perché appartiene alla loro stessa classe, dunque mi consente di stampare gli attributi di `Rettangolo`.

Un'alternativa è l'uso di un metodo di tipo `get`.
Nel codice qui a fianco, `get_base()` permette a parti di programma esterne alla classe di accedere a valori interni e non chiede parametri in input.

Il `main` «vede» le funzioni non void come variabili.
(es.: `get_base()` è trattato come un `int`)

I metodi che restituiscono un valore, sono visti dal `main` come variabili del tipo restituito. La sintassi del `cout` è dunque la stessa usata per le variabili.

Costruttori

Sono dei tipi particolari di funzioni che creano un oggetto appartenente alla classe. Vengono chiamati automaticamente ogni volta che viene creato un oggetto della classe. Servono ad evitare risultati indeterminati nel caso in cui si invochi una funzione della classe senza aver prima definito l'oggetto su cui lavorare.

```
class Rettangolo{
        int base, altezza;
public:
        Rettangolo (int, int);
        void set_values (int,int);
        int area() {return base*altezza;};
                };

Rettangolo::Rettangolo (int x, int y) {
        base = x;
        altezza = y;
        }

int main () { ...}
```

I creatori si dichiarano specificando il nome e il tipo di ogni caratteristica dell'oggetto che creeranno. Nell'esempio, **rendono inutile il metodo `set_values`**.

I costruttori devono avere lo stesso nome della classe in cui sono definiti e non hanno un tipo: nemmeno void! Questo perché i costruttori non danno output, ma hanno la sola funzione di inizializzare gli oggetti.

Costruttori

Possiamo dunque usare il costruttore dal `main`, come un qualsiasi altro metodo della classe. Per definizione, il costruttore richiede due parametri che assegnerà a `base` e `altezza`, dunque nel `main` scriveremo direttamente questi due valori…

```
class Rettangolo{
        int base, altezza;
public:
        Rettangolo (int, int);
        int area() {return base*altezza;}
                };

Rettangolo::Rettangolo (int x, int y) {
        base = x;
        altezza = y;
        }

int main () {
        Rettangolo rett (3,4);
        Rettangolo rett2 (10,20);
...}
```

In una sola riga abbiamo creato l'oggetto e abbiamo assegnato un valore ai suoi attributi.
Prima avevamo:

```
int main () {
        Rettangolo rett;
        Rettangolo rett2;
        rett.set_values (3,4);
        rett2.set_values (10, 25);
        ... }
```

A differenza delle funzioni, i costruttori possono essere chiamati **solo una volta** per ogni oggetto: contestualmente alla loro creazione.

Overload di Costruttori

L'overload è la definizione multipla di funzioni e vale anche per i costruttori. Ad esempio, oltre a quello appena visto, possiamo definire un costruttore di default che viene usato quando non si specificano base e altezza.

```
class Rettangolo{
        int base, altezza;
public:
        Rettangolo ();
        Rettangolo (int, int);
        int area() {return base*altezza;}
                };

Rettangolo::Rettangolo (int x, int y) {
        base = x;
        altezza = y;
        }

Rettangolo::Rettangolo () {
        base = 1;
        altezza = 1;
        }

int main () { ...}
```

In questo modo, quando non specifico base e altezza, l'oggetto creato sarà un rettangolo con base = 1 e altezza = 1….un quadrato!

```
int main () {
        Rettangolo rett (10,6);
        Rettangolo rett2;
                ... }
```

Quando il programma viene eseguito, viene chiamato il costruttore che corrisponde alla lista di parametri inseriti.

A differenza delle funzioni, i costruttori di default vengono chiamati solo quando nessun parametro gli viene passato…ma non vogliono le parentesi tonde vuote!

Ereditarietà

E' possibile creare delle sottoclassi a partire dalle classi già definite. Per esempio, potremmo creare la sotto-classe `Rettangolo`, dalla classe dei `Poligoni`. Le classi figlie mantengono gli attributi delle classi da cui provengono e possono averne degli altri.

```
class Poligoni {
        protected:
                int base, altezza;
        public:
                void set_values (int a, int b) {base=a;
altezza=b;}
             };

class Rettangolo: public Poligoni {
                        public:
                             int area () {
                        return base*altezza;}
                             };

int main () {
        Rettangolo rett;
        rett.set_values (4,5);
        cout<<rett.area()<<'\n';
        return 0;
        }
```

La relazione di ereditarietà delle due classi viene dichiarata nella classe derivata usando la sintassi:

```
class classe_figlia: public
classe_madre {... };
```

Lo specificatore d'accesso (come al solito: `private/public/protected`) stabilisce il livello di accesso della classe figlia.

Membri `protected` sono visibili alle classi figlie, membri `privati` no.

I membri ereditati mantengono la stessa visibilità per gli oggetti esterni alla classe.

Ereditarietà (2)

```
class classe_figlia: public classe_madre {...};
```

Lo specificatore denota il livello più alto accessibile che i membri ereditano dalla classe che lo segue (nell'esempio Poligono) avranno dalla classe ereditata (in questo caso, Rettangolo). Poiché `public` è il livello più alto accessibile, la classe derivata definita con questo specificatore erediterà tutti i membri con gli stessi livelli della classe da cui deriva.

```
class classe_figlia : protected classe_madre {...};
```

Con `protected`, i membri pubblici sono ereditati come protetti, nella classe derivata.

```
class classe_figlia : private classe_madre {...};
```

Con `private`, tutti i membri della classe madre sono ereditati come privati.

Membri Virtuali (Polimorfismo)

Un membro virtuale è una funzione membro che può essere ridefinita in una classe derivata, preservando le sue proprietà di chiamata tramite **referenza**. La sintassi per una funzione derivata prevede di farne preseguire la dichiarazione con `virtual`:

```
class Poligoni {
        protected: int base, altezza;
        public: void set_values (int a, int b) {
 base = a; altezza = b; }
        virtual int area () { return 0; }
        };
class Rettangolo: public Poligoni {
        public: int area () { return base * altezza; }
        };
class Triangolo: public Poligoni {
        public: int area () { return (base* altezza /
2); }
};
```

```
int main () {
Rectangolo rett;
Triangolo triang;
Poligoni poly;
Poligoni * ppoly1 = &rett;
Poligoni * ppoly2 = &triang;
Poligoni * ppoly3 = &poly;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly3->set_values (4,5);
cout << ppoly1->area() << '\n';
cout << ppoly2->area() << '\n';
cout << ppoly3->area() << '\n';
return 0;
}
```

Per applicare un metodo ad un oggetto uso: `oggetto.metodo(parametri)`.
Per applicare un metodo ad un puntatore devo usare: `puntatore->metodo(parametri)`.

Ricordate la sintassi dei puntatori : `&variabile` indica l'indirizzo fisico in cui è salvata l'informazione. Il dato è accessibile con la sintassi `*puntatore`.

Membri Virtuali (Polimorfismo)-(2)

```
class Poligoni {
          protected: int base, altezza;
          public: void set_values (int a, int b) {
base = a; altezza = b; }
          virtual int area () { return 0; }
          };
class Rettangolo: public Poligoni {
          public: int area () { return base * altezza; }
          };
class Triangolo: public Poligoni {
          public: int area () { return (base* altezza/2);
}
};
```

```
int main () {
Rectangolo rect;
Triangolo trgl;
Poligoni poly;
Poligoni * ppoly1 = &rect;
Poligoni * ppoly2 = &trgl;
Poligoni * ppoly3 = &poly;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly3->set_values (4,5);
cout << ppoly1->area() << '\n';
cout << ppoly2->area() << '\n';
cout << ppoly3->area() << '\n';
return 0;
}
```

Il metodo `area()` è stato dichiarato `virtual` nella classe madre perché è ridefinito nelle classi figlie. I membri non-virtuali possono essere anch'essi ridefiniti nelle classi derivate ma l'accesso a tali membri non può avvenire tramite puntatori della classe madre: se `virtual` fosse rimosso dalla dichiarazione, tutte le tre chiamate restituirebbero zero perché verrebbe chiamata al loro posto la versione della classe madre.

In poche parole, `virtual` permette ai membri di una classe figlia aventi lo stesso nome di uno della classe madre di essere opportunamente chiamati da <u>puntatori appartenenti alla classe madre che puntano a un oggetto della classe figlia.</u>

Una classe che contiene o eredita membri virtuali è detta polimorfa.

Esempio

```
Class A {
                 int a;
        public:
                A() {a = 1;}
                 virtual void show(){cout <<a;}
         };

Class B: public A {
                 int b;
         public:
                 B(){b = 2;}
                 virtual void show(){cout <<b;}
                 };

int main() {
           A *pA;
           B oB;
           pA = &oB;
           pA->show();
           return 0;
            }
```

Cosa succede se eseguo questo codice?
Quale numero vedrò sul terminale?

Esempio

Vediamo, riga per riga, cosa sto facendo…

```
Class A {
                 int a;
        public:
                 A() {a = 1;}
        virtual void show(){cout <<a;}
        };

Class B: public A {
                 int b;
         public:
                 B(){b = 2;}
         void show(){cout <<b;}
                 };

int main() {
           A *pA; //dichiaro puntatore di A
           B oB;  //dichiaro oggetto di B
           pA = &oB; //faccio puntare pA a oB
           pA->show(); //applico metodo a pA
           return 0;
        }
```

Sto applicando un metodo ad un puntatore della classe madre (pA) che punta ad un oggetto della classe figlia (oB). Il metodo usato sarà dunque quello della classe figlia e sul terminale vedrò «2».

*.h ; *.c e *.cpp

```
class Rettangolo {
          int base, altezza;
public:
          void set_values (int,int);
          int area() {return base*altezza;}
          void stampa(){
          cout<<"B= " <<base<< "H= " <<
          altezza<<endl;
          }
                    };

void Rettangolo::set_values (int x, int y) {
          base = x;
          altezza = y;
          }

int main () {
          Rettangolo rett;
          Rettangolo rett2;
          rett.set_values (3,4);
          rett2.set_values (10, 25);
          rett.stampa();
          rett2.stampa();
          return 0;
              }
```

Nell'header (*.h) vanno inclusi solo i prototipi (le dichiarazioni) dei metodi.

Nel *.cc va specificato il corpo di ciascun metodo.

Nel *.cpp va messo il main e tutte le istruzioni che vogliamo far eseguire al nostro programma.
Ovviamente dovremmo includere header e cc.

UNIVERSITÀ DEGLI STUDI ROMA TRE

*.h ; *.c e *.cpp (2)

```
class Rettangolo {
          int base, altezza;
public:
          void set_values (int,int);
          int area();
          void stampa();
          };

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
          cout<<"B= " <<base<< "H= " <<
          altezza<<endl;
          };

void Rettangolo::set_values (int x, int y) {
          base = x;
          altezza = y;
          };

int main () {
          Rettangolo rett;
          Rettangolo rett2;
          rett.set_values (3,4);
          rett2.set_values (10, 25);
          rett.stampa();
          rett2.stampa();
          return 0;
              }
```

Per vedere meglio la divisione, scriviamo la classe con le sole dichiarazioni dei metodi:

Nell'header (*.h) vanno inclusi solo i prototipi (le dichiarazioni) dei metodi.

Nel *.cc va specificato il corpo di ciascun metodo.

Nel *.cpp va messo il main e tutte le istruzioni che vogliamo far eseguire al nostro programma.
Ovviamente dovremmo includere il .cc.

*.h ; *.c e *.cpp (3)

```
class Rettangolo {
          int base, altezza;
public:
          void set_values (int,int);
          int area();
          void stampa();
          };

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
          cout<<"B= " <<base<< "H= " <<
          altezza<<endl;
          };

void Rettangolo::set_values (int x, int y) {
          base = x;
          altezza = y;
          };

int main () {
          Rettangolo rett;
          Rettangolo rett2;
          rett.set_values (3,4);
          rett2.set_values (10, 25);
          rett.stampa();
          rett2.stampa();
          return 0;
              }
```

Nell'header (**rett.h**) vanno inclusi solo i prototipi (le «dichiarazioni») dei metodi.

```
#ifndef RETT_H
#define RETT_H

class Rettangolo {
          int base, altezza;
public:
          void set_values (int, int);
          int area();
          void stampa();
          };

#endif
```

*.h ; *.c e *.cpp (4)

```
class Rettangolo {
        int base, altezza;
public:
        void set_values (int,int);
        int area();
        void stampa();
        };

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
        cout<<"B= " <<base<< "H= " <<
        altezza<<endl;
        };

void Rettangolo::set_values (int x, int y) {
        base = x;
        altezza = y;
        };

int main () {
        Rettangolo rett;
        Rettangolo rett2;
        rett.set_values (3,4);
        rett2.set_values (10, 25);
        rett.stampa();
        rett2.stampa();
        return 0;
            }
```

Nel **rett.cc** va specificato il corpo di ciascun metodo (implementazione).

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

int Rettangolo::area() {return
base*altezza;};

void Rettangolo::stampa() {
        cout<<"B= " <<base<< "H= "
        <<altezza<<endl;
        };

void Rettangolo::set_values (int x,
int y) {
        base = x;
        altezza = y;
        };
```

*.h ; *.c e *.cpp (5)

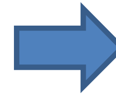
```
class Rettangolo {
         int base, altezza;
public:
         void set_values (int,int);
         int area();
         void stampa();
         };

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
         cout<<"B= " <<base<< "H= " <<
         altezza<<endl;
         };

void Rettangolo::set_values (int x, int y) {
         base = x;
         altezza = y;
         };

int main () {
         Rettangolo rett;
         Rettangolo rett2;
         rett.set_values (3,4);
         rett2.set_values (10, 25);
         rett.stampa();
         rett2.stampa();
         return 0;
             }
```

Nel **prog.cpp** va messo il main e tutte le istruzioni che vogliamo far eseguire al nostro programma.
Ovviamente dovremmo includere header e cc.

```
#include "rett.cc"
#include <iostream>

int main () {
         Rettangolo rett;
         Rettangolo rett2;
         rett.set_values (3,4);
         rett2.set_values (10, 25);
         rett.stampa();
         rett2.stampa();
         return 0;
             }
```

*.h ; *.c e *.cpp (6)

```
class Rettangolo {
            int base, altezza;
public:
            void set_values (int,int);
            int area();
            void stampa();
            };

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
            cout<<"B= " <<base<< "H= " <<
            altezza<<endl;
            };

void Rettangolo::set_values (int x, int y) {
            base = x;
            altezza = y;
            };

int main () {
            Rettangolo rett;
            Rettangolo rett2;
            rett.set_values (3,4);
            rett2.set_values (10, 25);
            rett.stampa();
            rett2.stampa();
            return 0;
                }
```



Il programma.cpp diventa:

```
#include "rett.cc"
#include <iostream>

int main () {
            Rettangolo rett;
            Rettangolo rett2;
            rett.set_values (3,4);
            rett2.set_values (10, 25);
            rett.stampa();
            rett2.stampa();
            return 0;
                }
```

in cui abbiamo incluso:

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

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
            cout<<"B= " <<base<< "H= "<<altezza<<endl;};

void Rettangolo::set_values (int x, int y) {
            base = x;
            altezza = y;};
```

rett.cc

```
#ifndef RETT_H
#define RETT_H

class Rettangolo {
            int base, altezza;
public:
            void set_values (int, int);
            int area();
            void stampa();
            };

#endif
```

rett.h

Compilazione

Abbiamo diviso il nostro codice in tre parti…
Se compilassimo solo il programma.cpp,
otterremmo degli errori!

```
#include "rett.cc"
#include <iostream>

int main () {
            Rettangolo rett;
            Rettangolo rett2;
            rett.set_values (3,4);
            rett2.set_values (10, 25);
            rett.stampa();
            rett2.stampa();
            return 0;
                }
```

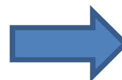
```
#include "rett.h"
#include <iostream>

int Rettangolo::area() {return base*altezza;};

void Rettangolo::stampa() {
             cout<<"B= " <<base<< "H= "<<altezza<<endl;};

void Rettangolo::set_values (int x, int y) {
            base = x;
            altezza = y;};
```

```
#ifndef RETT_H
#define RETT_H

class Rettangolo {
            int base, altezza;
public:
            void set_values (int, int);
            int area();
            void stampa();
            };

#endif
```

Quello che dobbiamo fare (dal terminale) è:

```
g++ -c rett.cc
g++ -c programma.cpp
g++ programma.o rett.o – o Prova
```

..e poi lanciare l'eseguibile «Prova»
(come al solito!)



```
./Prova
```