

# IMPARARE IL C++ in 2 ore e mezzo? Modulo-2

Roberto Santinelli-Perugia

# Outline Modulo2

- References
- Funzioni speciali
- Vettori
- Ineritanza singola
- Polimorfismo o multipla ereditanza
- Variabili statiche e puntatori di funzioni
- Macros e Template

# REFERENZE

# References

**Le references combinano parzialmente il potere dell'uso dei puntatori con una sintassi molto più facile e intuitiva**

**Reference=alias ad una variabile**

**Sintassi:**

```
int someInt;
```

```
int &rSomeRef= someInt; //sempre inizializzare un ref.
```

**Leggi come: rSomeRef è una referenza ad un intero che è inizializzato con la variabile someInt.**

# References: esempio 1

# References ad un oggetto

```
#include <iostream.h>
Class Cat
{
    public:
        Cat(int age,int weight); → prototipo di questo metodo da imple.
        ~Cat(){delete itsAge; } → implementazione inline
        int GetAge() {return *itsAge;}
        int GetWeight() {return itsWeight;}
    private:
        int *itsAge =new int;
        int itsWeight;
};

Cat::Cat(int age,int weight)
{
    *itsAge=age;
    itsWeight=weight
}

Int main ()
{
    Cat Melissa(5,8);
    Cat &rMelissa=Melissa;
    cout<<"Melissa pesa "<<Melissa.GetAge()<<endl;
    cout<<Melissa ha " <<rMelissa.GetWeight()<<endl;
    Return 0;
}
```

# Passare valori ad una funzione per referenza

I limiti di una funzione sono: passare valori e avere di ritorno un solo valore .Puntatori e references sorpassano questo limite come ad esempio

Esempio:

```
void swap (int &x,int &y); //prototipo dello swap
int main() {
    int x=5,y =10;
    cout <<x<<"\t"<<y<<endl; //5 10
    swap(x,y); //x e y sono in swap I riferimenti di x,y in main!
    cout <<x<<"\t"y<<endl; //10 5
    return 0; }
```

# Passare valori ad una funzione per riferimento.

```
Void swap(int &rx, int &ry) {  
    Int temp;  
    temp=rx;  
    rx=ry;  
    ry=temp;  
}
```

Poiche' I parametri in swap() sono dichiarati essere referenze (potevi usare anche i puntatori a x e y con l'ingombrante necessita' di dereferenzare l'indirizzo) tu agisci ora su questi e non su variabili locali che all'uscita di una funzione vengono rimosse e quindi risentono l'azione anche sul main()

# Funzioni con piu valori al ritorno (altra informazione appresa da swap() )

Come fare se ho bisogno di avere piu'valori in uscita da una funzione?

La filosofia e' quella dell'esempio di swap.

- 1.Immetto due oggetti come parametri nella funzione per riferimento (o i puntatori a)
  - 2.La funzione riempie I due riferimenti (e quindi le originali variabili globali in main() cambiano)
- Cosi' si bypassa il valore di ritorno che puo' essere usato per riportare altre informazioni come eventuali errori per esempio.

# Funzioni con piu' valori di ritorno (mix di reference e pointer)

```
#include<iostream.h>
typedef USHORT unsigned short int;
short Factor(USHORT, USHORT *, USHORT &); //prototipo di Factor
int main() { value     puntatore riferimento
USHORT numero,quadrato,cubo;
short error;
cout<<"Dammi un numero (0-20)<<endl;
cin>>numero;      indirizzo di quadrato, alias della variabile globale cubo
error=Factor(numero, &quadrato, cubo) (gli passo l'indirizzo del quadrato)
if(!error) {
cout<<numero<<"\t"<<quadrato<<"\t"<<cubo<<endl;
}
else
    cout<<"errore!<<endl;
Return 0; }
```

# Esempio funzione a multipli valori

```
Short Factor(USHORT n, USHORT *pQuadro, USHORT &rCubo)
{
    short Value=0;
    if(n>20) Value=1; //errore:tropo grande!
    else {
        *pQuadro= n*n;
        rCubo=n*n*n;
        Value=0;
    }
    return Value; //non e' questo che cambia il corso delle cose
}
```

# **FUNZIONI AVANZATE**

## **(OVERLOADING DI METODI & COPY COSTRUCTOR)**

# Overloading i membri di una classe.

Abbiamo già accennato al polimorfismo di funzioni (funzioni con stesso nome ma differenti parametri di ingresso e tipo di ritorno)

Vediamo questa dichiarazione di una classe e dei suoi metodi interni:

```
class rettangolo
```

```
{ public:
```

```
    rettangolo(){int larghezza=10;int lunghezza=5;} //default values
```

```
    ~rettangolo() {} //distruttore
```

```
    void disegna(); //disegna con * un rettangolo larghezzaXlunghezza  
                  // (default)
```

```
    void disegna(int larg2,int lung2); //disegna con * un rettangolo larg2Xlung2
```

```
private:
```

```
    int lunghezza;
```

```
    int larghezza;
```

```
};
```

Chiamare la funzione `rettangolo.disegna()` o `rettangolo.disegna(x,y)` sortisce risultati differenti

# Overloading il costruttore

**Il costruttore e' un particolare metodo di una classe che viene chiamato ogni volta che si istanzia un oggetto.**

**Prima che il costruttore funzioni, c'e' solo l'area di memoria su cui poi verra' messo un oggetto particolare.**

**Il costruttore stabilisce l'oggetto.**

**Se non si specifica cosa deve fare, quando crei un oggetto, un costruttore che non fa nulla di default viene preso.**

**TUTTAVIA PUOI DEFINIRE IL TUO DEFAULT per setting-up l'oggetto.**

**SOLO DOPO che il costruttore ha finito l'oggetto del tipo class esiste nel tuo HD.**

# Overloading il costruttore

Come ogni funzione, per il COSTRUTTORE, possiamo cambiare la lista di parametri in ingresso e le cose che fa, Per esempio possiamo avere per il nostro rettangolo due costruttori:

```
class Rectangle
{
    Public:
        Rectangle();
        Rectangle(int width,int length);
        ~Rectangle();
        int GetWidth() {return itsWidth;}
        int GetHeight() {return itsHeight;}
    private:
        itsWidth;
        itsHeight;
};
```

# Overloading il costruttore

```
Rectangle::Rectangle()  
{ itsWidth=5;  
itsHeight=10; }  
Rectangle::Rectangle(int width,int height)  
{  
itsWidth=width;  
itsHeight=height;  
}
```

.....  
**La chiamata Rectangle Rec1; vuol dire che chiama il costruttore senza parametri e di conseguenza ho un rettangolo 5X10, mentre la chiamata Rectangle Rec2(3,12) mi creera' un rettangolo 3X12 user defined**

# Inizializzazione di oggetti

Nel costruttore si inizializzano i membri della classe previo l'uso di metodi accessori quali es :`SetWidth(int width)`

Si possono avere due modi per inizializzare i membri di una classe:

- 1.Nel corpo del costruttore (vedi sopra) molto trasparente
- 2.Nella fase di inizializzazione o dichiarazione della classe con un'appropriata sintassi : `costrName():privateVariable(initValue)`

Es:

```
class Rectangle
{
    Public:
        Rectangle():itsWidth(5),itsHeight(10) { body del costruttore 1;}
        Rectangle(width,height):itsWidth(width),itsHeight(height) { body 2;}
    Private:
        Int itsWidth,itsHeight;
};
```

# Il costruttore “copy”

Oltre a provvedere ad un costruttore di default il compilatore fornisce anche un default copy costruttore.

**IL COPY costruttore VIENE CHIAMATO OGNI VOLTA CHE UNA COPIA DI UN OGGETTO ESISTENTE VIENE FATTA**

Il costruttore copy prende un unico parametro di ingresso che e'

Il riferimento ad un oggetto della stessa classe:

CAT(CAT & theCat);

Cioe' il costruttore CAT prende come input ogni membro e ogni metodo di un esistente oggetto di tipo CAT che e' stato creato precedentemente theCat.

In altri termini se theCat ha una variabile itsAge che punta ad una locazione di memoria (per esempio nel free store), il default copy constructor copiera' questo valore nel membro itsAge del nuovo CAT che si costruirà con una eventuale chiamata del copy nel main().

Per evitare stray pointer(puntatori a locazioni di memoria che vengono cancellate) si deve creare l'oggetto copia solo dopo aver allocato tutte le altre locazioni di memoria in cui storo i vari puntatori ai membri interni della classe.

## ESEMPIO DELL'USO DEL COPY COSTRUTTORE

```
#include<iostream.h>
Class CAT {
{
Public:
CAT();
CAT (CAT &); // copy constructor
int GetAge() {return *itsAge;}          dichiarazione della classe e inline implementazione dei metodi accessori
int GetWeight() {return *itsWeight;}
void SetAge(int age) {*itsAge=age;}
private:
int *itsAge;
int *itsWeight;
};
CAT::CAT()
{
itsAge= new int;           //implementazione costruttore
itsWeight= new int;
*itsAge =5;
*itsWeight=9;
}
CAT::CAT( CAT & rhs)
{
           //implementazione copy costruttore
itsAge= new int; //nuova memoria e' allocata
itsWeight= new int;
*itsAge =rhs.GetAge();//assegna a questa nuova memoria il valore storato nella memoria puntata dalla variabile membro itsAge dell'oggetto RHS!
*itsWeight=rhs.GetWeight();
}
CAT::~CAT()
{delete itsAge;           //implementazione distruttore
itsAge=0;
delete itsWeight;
itsWeight=0;
}
```

RHS=right handed side (CAT object in questo caso)

E' tutto cio che sta a destra in un' uguaglianza.

```
int main()
{
CAT frisky;
cout <<“frisky ha”<<frisky.GetAge()<<endl; (5 anni)
frisky.SetAGe(6);
CAT Foofy(frisky); //Foofy acquisisce tutte le caratteristiche attuali di friscky
cout <<“frisky ha”<<frisky.GetAge()<<endl; (6 anni)
cout <<“frisky ha”<<Foofy.GetAge()<<endl; (6 anni)
Frisky.setAge(7)
cout <<“frisky ha”<<frisky.GetAge()<<endl; (7 anni)
cout <<“frisky ha”<<Foofy.GetAge()<<endl; (6 anni)
return 0;
}
```

# Vettori in C++

# Vettori

Dichiari un vettore scrivendo il tipo seguito dal nome del vettore e dal suscritto che e' il numero di elementi

`int intArray[25];`

Si accede ai valori di un vettore riferendosi all'offset di quell'elemento ricordandoti che la numerazione va da 0 a n-1 (24)

**NB.puoi scrivere in una locazione di memoria oltre il limite (es nell'elemento intArray[25]) con imprevedibili risultati!!!**

## Esempio di scrittura oltre il limite (errore molto comune)

# Inizializzazione di vettori

Qualsiasi nome purché non conflitta con una variabile differente.

Puoi anche dichiarare e inizializzare nello stesso tempo un vettore

```
int Vettore[5]={10, 20, 30, 40, 50};
```

```
int Vettore[ ]={10, 20, 30, 40, 50}; //equivalente
```

NB. Importante la sintassi comma-spazio

E puoi ottenere la dimensione di un vettore con

```
int dimensione=sizeof(Vettore)/sizeof(Vettore[0]);
```

Puoi anche creare un vettore di puntatori (1) e assegnare un  
elemento ad un puntatore ad una variabile dello stesso tipo(2)

```
(1)long * VettorediPuntatori[20];
```

```
(2) long *plong=Vettoredi Puntatori[8];
```

# Vettori di oggetti

Dichiarare un vettore significa dire al compilatore quanto spazio ha ogni elemento (type) e quanti di questi cubicoli consecutivi assegnare al vettore (n).

Puoi fare in modo che ogni elemento sia un ente user defined (oggetto)

```
#include<iostream.h>
class CAT{
public:
    CAT() {itsAge=1;}      //dichiarazione della classe
    CAT ~CAT()             //implementazione inline
    int GetAge() {return itsAge;}
    void SetAge(int age) {itsAge=age;}
private: int itsAge;};
int main()
{
    CAT Litter[5];
    for(l=0;l<5;l++)
        Litter[l].SetAge(2*l+1);
    cout<<"Cat number 5 ha un'eta di "<<Litter[4].GetAge()<<endl;
    return 0;
}
```

# Array multidimensionali

Ogni dimensione e' rappresentata da un suscritto e non ci sono limiti sul numero delle dimensioni

Supponi di avere una classe chiamata **SQUARE**:

**SQUARE Board[8] [8];**

Per inizializzare un vettore a piu' dimensioni la sintassi e' la stessa con la regola che prima si riempiono gli elementi dell'ultimo suscritto tenendo fissi i primi

`int MultiVettore[5][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }`

che per esigenza di chiarezza si possono raggruppare con parentesi

`int MultiVettore[5][3]={ {1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15} }`

**IL COMPILATORE IGNORA LE PARENTESI INTERNE**

# Array di puntatori

Fino ad ora abbiamo creato vettori che storano i loro elementi nello stack.

Si potrebbe dichiarare un oggetto nel free store e quindi storare il puntatore all'oggetto come elemento di un vettore.

Riagganciamoci all'esempio di prima con la classe CAT già dichiarata

```
main()
{
CAT *Family[500]; //dichiara un 'array di 500 puntatori a CAT object
int I;
CAT * pCat;      // dichiara pCat essere puntatore ad un CAT nel free store
for(I=0;I<500;I++)
{pCat=new CAT;
pCat->SetAge(2*I+1);
Family[I]=pCat;
delete pCat;      //sempre cancellare un puntatore nel free store se vuoi riutilizzarlo!!!
cout<<"Cat # "<< I+1<<"di eta"<<Family[I]->GetAge()<<endl;
}
return 0;
}
```

# Dichiarare vettori nel free store

Si puo' mettere nel free store un intero vettore usando new e l' operatore suscritto.

CAT \* Family= new CAT[500];

Il risultato e' che Family e' un puntatore ad un area del free store che contiene un vettore.Nell'esempio dichiari Family essere un puntatore al primo elemento in un array di gatti.

In questo modo puoi usare un puntatore aritmetico per accedere ad ogni elemento cui punta Family

CAT \*Family= new CAT[500];

CAT \*pCat =Family; //pCat e' Family e quindi punta a Family[0]

pCat->SetAge(10); //Family[0] ha eta 10

pCat++; //pCat e' diventato Family[1]

pCat->SetAge(20); //setta Family[1] a 20

Ricordati che Family[] e' per ogni indice un puntatore nel free store e come tale va cancellato dopo l'uso.Prima del return 0; in main chiama

delete [] Family; //le parentesi quadre segnalano al compilatore che l'intero array viene rimosso dal free store

# Un attimo di attenzione

1: CAT FamilyOne[500];

2: CAT \*FamilyTwo[500];

3: CAT \*FamilyThree =new CAT[500]

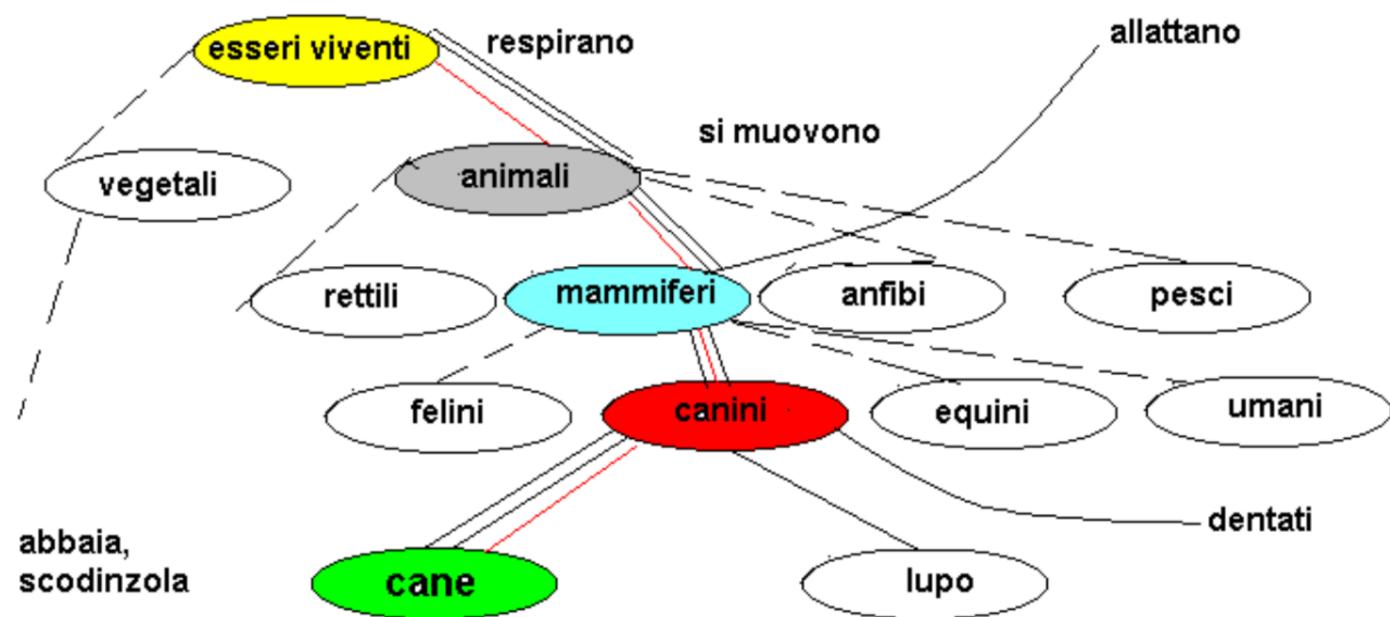
FamilyOne e' un vettore di 500 gatti

FamilyTwo un vettore di 500 puntatori a gatti

FamilyThree e' il puntatore a un vettore di  
500 gatti nel free store

# INHERITANCE

E' un aspetto dell'intelligenza umana cercare, riconoscere e creare relazioni fra le cose o fra concetti. Si costruiscono matrici, reti e altre interrelazioni per spiegare e comprendere il modo con cui interagiscono le cose>Il C++ tenta di riprodurre questa capacità umana tramite le relazioni di ereditanza



## INERITANZA:

un cane automaticamente eredita le caratteristiche di un canino e quindi di un mammifero e così via.

# ineritanza

Il C++ permette di riprodurre queste relazioni gerarchiche permettendo di definire classi che derivano da altre.

Così' per un cane che dici derivare da un mammifero, non hai bisogno di dire che allatta perche' questa caratteristica e' già insita nel concetto di mammifero da cui deriva e da cui **inerisce** le caratteristiche, nulla impedendo di aggiungerne delle altre (scodinzolare abbaiare)

# Ineritanza:sintassi

---

Mammifero si chiama classe base del cane.

Cane e' una classe derivata

Per ereditare si usa la sintassi:

**Class Dog: public Mammal**

```
#include<iostream.h>
enum BREED {YORKIE,DANDIE,SHETLAND,DOBERMAN}
class Mammal
{public:
    Mammal():itsAge(2),itsWeight(5) {}
    ~Mammal() {}
    int getAge(){return itsAge;}
    void setAge(int age){itsAge=age;}
    int GetWeigth(){return itsWeigth;}
    void setWeigth(int weigth) {itsWeigth=weigth;}
```

---

```
void Sleep(){cout<<“mammifero che dorme”<<endl;}
void Speak(){cout<<“mammifero che parla”<<endl;}
protected: //protected non e' private che non puo' essere vista da Dog
int itsAge; //ne public che puo essere vista da chiunque
int itsWeigth;
};

class Dog:public Mammal //ineritanza da un mammifero
{
public:
Dog():itsBreed(YORKIE){ }
~Dog() { }
BREED GetBreed() {return itsBreed;}
void SetBreed (BREED breed) {itsBreed=breed;}
void WagTail() {cout<<“tail wagging”<<endl;}
private:
BREED itsBreed;
};
```

```
Int main()
{
Dog Fido;
Fido.Speack(); //la classe Dog NON ha esplicitamente definito il metodo
Fido.WagTail(); //speack
cout<<"Fido e' un cane di "<<Fido.GetAge()<<"anni"<<endl;
return 0;
}
```

Oggetti tipo cane sono oggetti tipo mammifero.

Quando Fido viene creato il costruttore della base class viene prima chiamato (mammifero) e quindi quello della classe cane. In questo caso l'eta di Fido e' di due anni perche' nel costruttore di mammifero c'e' il default 2 per settare l'eta'

Alla distruzione di Fido prima il distruttore cane viene chiamato e poi quello del mammifero.

# Overloading costruttori in classi derivate

Nell'esempio di prima e' possibile che si voglia overloaddare il costruttore di Mammal per immettere una specia eta'e lo stesso avere un costruttore Dog da settare che specie di cane sia o altro.

```
class Mammal {  
public:  
Mammal();  
Mammal(int age);  
~Mammal(); .....} ;//e gli altri accessori
```

```
class Dog:public Mammal  
{  
public  
Dog();  
Dog(int age);  
Dog(int age,int weight);  
Dog(int age,BREED breed);  
~Dog();.....}; //e tutti gli altri accessori e membri interni di un cane.....
```

## IMPLEMENTIAMO I COSTRUTTORI del MAMMAL e del DOG

Mammal::Mammal():

itsAge(1),itsWeigh(5)

```
{cout<<"Primo tipo di mammal costructor"<<endl;}
```

Mammal::Mammal(int age):// =====

itsAge(age), itsWeight(6)

```
{cout<<"Secondo tipo di mammal costructor"<<endl;}
```

Mammal::~Mammal():// =====

```
{cout<<"distruttore di mammiferi"<<endl;}
```

Dog::Dog() : //=====

Mammal(),itsBreed(YORKIE)

```
{cout<<"primo tipo di costruttore di cani"<<endl;}
```

Dog::Dog(int age): //=====

Mammal(age),itsBreed(YORKIE)

```
{cout<<"secondo tipo di costruttore di cani"<<endl;}
```

Dog::Dog(int age,int weight): //=====

Mammal(age),itsBreed(YORKIE)

```
{itsWeight=weight;
```

cout<<"terzo tipo di costruttore di cani"<<endl;}

Dog::Dog(int age,BREEDbreed)://=====

Mammal(age),itsBreed(breed)

```
{cout<<"quarto tipo di costruttore di cani"<<endl;}
```

Dog::~Dog() {cout<<"distruttore di cani"<<endl;}

```
int main()
{
    Dog Fido;
    Dog Rover(5);
    Dog Buster(5,8); //comment
    Dog Dobbie(4,DOBERMAN);
    Fido.Speak();
    Rover.WagTail();
    cout<<"Buster e' un cane di"<<Buster.GetWeigth()<<"anni"<<endl;
    cout<<"Dobbie e' un tipico esempio di"<<Dobbie.GetBreed()<<endl;
    //non ti aspetterai mica che scriva DOBERMAN???
    cout<<"Dobbie ha"<<Dobbie.GetAge()<<"anni"<<endl;
    cout<<"Dobbie pesa"<<Dobbie.GetWeight()<<"chili"<<endl;
    return 0;
}
```

# Overriding funzioni di una base class in una classe derivata

Overriding =cambiare l'implementazione di una funzione ma lasciando immutata la **signatura**

Signatura=il nome della funzione+numero +tipo dei parametri che la funzione prende MA non il tipo di ritorno che puo' cambiare

Overloading=rimane solo il nome uguale

Es.

Nella classe Mammal c'e' una funzione (=metodo) Speak() che per esempio fa printare "Suono del Mammifero"

Se nella classe derivata Dog creo una funzione Speak che per esempio fa printare il "woof" tipico di un cane (overriding il metodo speak) alla riga Fido.Speak() io mi attendo woof e non come prima l'automatico Speak del mammifero.

Overriding implica quindi che l'originale metodo viene **ESCLUSO**

Se un metodo di una base class (Mammal viene overridden) si puo' comunque chiamare quel particolare metodo definendo il PATH completo :

**Fido.Mammal::Speack();**

che chiama esplicitamente il metodo Speack originario e non il woof del cane.

---

**METODI VIRTUALI :puntatori a mammiferi che posso usare per trovare un cane**

Si puo' anche avere il puntatore ad una classe di base assegnato ad una classe derivata.

**Mammal \* pMammal=new Dog;**

Un puntatore ad un Mammifero e' creato ma gli viene assegnato l'indirizzo di un oggetto Dog creato nel free store.

Un oggetto Dog viene creato, ma ho di ritorno un puntatore ad un mammifero, perche' un CANE e' un MAMMIFERO!

NB Questa e' l'essenza del polimorfismo:supponi che

classe base=WINDOWS derivate classi, CONGO BOX, SCROLLABLE WINDOWS, DIALOG BOX etc e il metodo draw() che disegna il tipo di finestra.

Se tu crei il puntatore generico ad una WINDOWS e lo assegni di volta in volta ad un suo derivato oggetto, NON TI PREOCCUPI DI QUALE metodo derivato draw() stai chiamando, perche' la corretta funzione draw() sara' chiamata e gli effetti saranno quelli che ti attendi.

Con funzioni VIRTUALI puoi chiamare il metodo corretto della classe derivata che "overrida" il metodo della base class sebbene stai usando il puntatore di un mammifero che quindi va a cercare il metodo nella classe mammifero.

## ESEMPIO DELL'USO DI METODI VIRTUALI

```
#include <iostream.h>
class Mammal {
public:
Mammal():itsAge(2){ }
~Mammal() { }
void Move() {cout<<"Mammifero che si muove";} //non e' virtuale
virtual void Speak() {cout<<"Mammal sound"<<endl;}
    //Virtual informa il compilatore di andare a guardare il metodo
    //Speak della classe derivata e non quello di mammal.
protected:          //Ovviamente virtual presuppone che nelle classi derivate da mammal
int itsAge;         // ci sia il metodo Speak().
};
class Dog:public Mammal {
public:
void Speak() {cout<<"Woof"<<endl;}
void Move() {cout<<"Cane che si muove";}
};
class Cat:public Mammal {
public:
void Speak() {cout<<"Meow"<<endl;}
};
class Horse:public Mammal {
public:
void Speak() {cout<<"Winnie"<<endl;}
};
```

```
class Pig:public Mammal {  
public:  
void Speak() {cout<<“Oink”<<endl;}  
};  
int main( )  
{  
Mammal* theArray[5]; //creato un vettore di puntatori a 5 oggetti Mammal  
Mammal *ptr; //puntatore ad un Mammal  
Int choice, I;  
For(I=0;I<5;I++)  
{  
cout<<“(1) Dog ,(2) Cat,(3) Horse,(4) Pig :”;  
cin >>choice;  
switch (choice)  
{  
case 1:ptr=new Dog;  
break;  
case 2:ptr=new Cat;  
break;  
case 3:ptr=new Horse;  
break;  
case 4:ptr=new Pig;  
break;  
Default:ptr=new Mammal;  
break; }
```

```
theArray[I]=ptr;
} // chiude il ciclo for aperto per memorizzare i vari elementi del vettore di puntatore user           //defined
for (I=0;I<5;I++)
theArray[I]->Speak(); "Abbaiera',miagolera' etc" perche' Speak in mammal e' virtuale e agisce quello overridden della
                           //relativa classe derivata cui punta theArray[I]
theArray[I]->Move(); Scrivera' sempre (anche nel caso del cane dove c'e' il metodo Move()) "Mammifero che si muove"
return 0;
}
```

L'uscita di questo programma la vedi come

```
(1) Dog (2) Cat (3)Horse (4)Pig : 1
(1) Dog (2) Cat (3)Horse (4)Pig : 2
(1) Dog (2) Cat (3)Horse (4)Pig : 3
(1)Dog (2) Cat (3)Horse (4)Pig : 4
(1)Dog (2) Cat (3)Horse (4)Pig : 5
```

Woof

Meow

Winnie

Oink

Mammal Sound

La parola chiave VIRTUAL permette di fare questo. Infatti se io non definivo virtual NON potevo usare le funzioni overridden delle rispettive classi derivate ma la generica funzione Speak del mammifero (Mammal sound)

In altri termini il compilatore sa' che ptr e' il puntatore ad un Mammifero e come tale va a cercare nell'oggetto mamifero la funzione chiamata (Speak()).

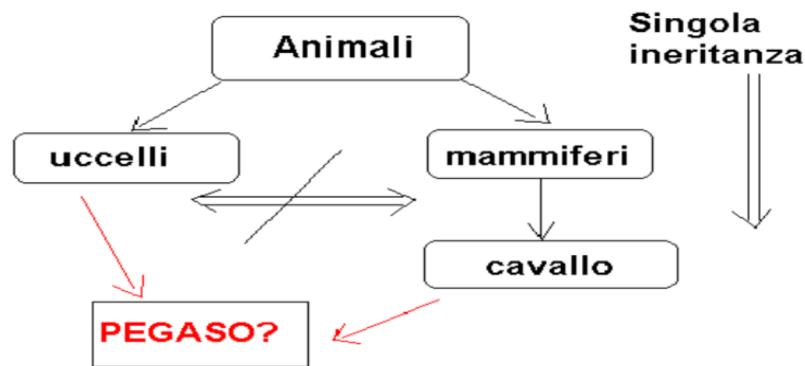
Con la parola VIRTUAL nel metodo Speak() informi il compilatore di invocare il metodo Speak overridden in DOG o in CAT o in ogni altra classe derivata dell'occorrenza.

L'uso di VIRTUAL FUNCTIONS e' possibile solo con Puntatori e references

# Polimorfismo

L'essenza del polimorfismo e' stata vista poco fa come la capacita' di legare oggetti di una specifica classe derivata a puntatori della classe base generica runtime (VIRTUAL).

Tuttavia ci sono dei limiti nella semplice ereditanza usata che non ci permettono di fare tutto quello che vogliamo.  
Vediamo cosa non si puo' fare!



La classe bird contiene il metodo volare() la classe horse il metodo galoppa() e ntrisce().  
 Cosa accade se hai bisogno di un oggetto che deve saper volare e galoppare e ntrire che chiameremo pegaso?????

Con la semplice ereditanza non puoi:

- 1 se pegaso fosse un uccello non sarebbe un mammifero e quindi non avrebbe le caratteristiche base di un cavallo
- 2 potrebbe essere un cavallo ma non saprebbe volare perche' un mammifero non puo' volare

**SI POTREBBE COPIARE IL METODO VOLARE IN UN NUOVO OGGETTO CHE DERIVA DA UN CAVALLO MA BISOGNA RICORDARSI CHE MODIFICANDO IL METODO VOLARE IN UN UCCELLO DEVI MODIFICARE IL METODO ANCHE IN PEGASO.**

**-→DIFFICOLTA' DI MANTENERE IL CODICE CON IL PASSARE DEL TEMPO**

Sicuramente si potrebbe overhidding la funzione fly() di un cavallo in una derivata classe pegaso in maniera che in una non vola e nell'altra si

```
class Horse {  
public:  
void Gallop() {cout <<“galoppando”<<endl;}  
virtual void fly() {cout<<“I cavalli non possono volare”<<endl;}  
private itsAge;  
};  
class Pegaso:public Horse  
{  
public:  
virtual void fly(){cout<<“Posso volare!!!!”<<endl;  
};  
//chiamando tramite puntatori l'una o l'altro metodo fly() hai le cose giuste.....
```

### POSSIBLE SOLUTIONS

.....a costo di avere nella classe dei cavalli il metodo volare che li' non fa quello che ti aspetti ed e' innaturale per come si progetta la classe cavallo!!!

Si potrebbe evitare questo mettendo il metodo volare in una comune classe di base quale ad esempio la classe animal da cui derivono sicuramente uccelli e cavalli. Cosi' corri il rischio di ritrovarti una unica classe originaria usata come spazio globale di tutte le funzioni che possono essere usate dalle sue classi derivate.Va contro la filosofia del C++.!!!!

IL MODO PIU SEMPLICE PER SORPASSARE QUESTO PROBLEMA E' LA

MULTIPLA INERITANZA CIOE':

inerire da due o piu' classi di base una nuova classe derivata

La sintassi e' assolutamente la stessa della semplice ereditanza

**class Pegaso: public Horse, public Bird**

```
#include <iostream.h>
class Horse
{
public:
Horse() {cout<<“..Horse constructor...”;}
virtual ~Horse() {cout<<..Horse destructor...";}
virtual void Whinny() {cout<<“Whinny!”;}
private: int itsAge ;
};
class Bird
{
public:
Bird() {cout<<“...Bird constructor...”;}
virtual ~Bird() {cout<<....Bird destructor...”;}
virtual void Chirp() {cout<<“chirp....”;}
virtual void Fly() { cout<<“POSSO VOLARE”;}
private:int itsWeight;
}
class Pegaso:public Horse,public Bird
{
Public:
virtual void Chirp(){Whinny();}
Pegaso() {cout<<“..pegaso constructor..”;}
~Pegaso() {cout<<“..Pegaso destructor..”;}
};
```

```
int MagicNumber=2;
int main()
{
Horse * Ranch[MagicNumber];
Bird * Aviary[MagicNumber];
    Horse * pHorse;
    Bird * pBird;
int l,choice;
for(l=0;l<MagicNumber;l++)
{
cout<<"(1)Cavalo (2) Pegaso: ";
cin >>choice;
if(choice==2)
    pHorse=new Pegasus;
else
    pHorse=new Horse;
Ranch[l]=pHorse;
}
// Lo stesso lo devo fare per l'aspetto uccello del pegaso

for(l=0;l<MagicNumber;l++)
{
cout<<"(1)Uccello (2) Pegaso: ";
cin >>choice;
if(choice==2)
    pBird=new Pegasus;
else
    pBird=new Bird;
Aviary[l]=pBird;
}
```

```
cout<<"\n";
for (l=0;l<MagicNumber;l++)
{
    Ranch[l]->Whinny();
    delete Ranch[l]
}
for (l=0;l<MagicNumber;l++)
{
    Aviary[l]->Chirp();
    Aviary[l]->Fly();
    delete Aviary[l]
}
Return 0;
}

Output::
(1) Horse (2) Pegaso 1          //Se creando un pegaso vuoi evitare di chiamare i
...Horse costructor...           //costruttori di cavallo e uccello (come in esempio)
(1) Horse (2) Pegaso 2          //puoi dichiarare virtual i costruttori di horse e bird!!!
..Horse costructor,... Bird Costructor.... Pegaso Costructor....
(1) Bird (2) Pegaso 1
...Bird costructor...
(1) Bird (2) Pegaso 2
...Horse costructor..., Bird Costructor..., Pegaso Costructor...
```

Whinny.. Horse destructor...  
Whinny ...Pegaso destructor,...Bird destructor,...Horse destructo...  
Chirp...POSSO VOLARE ...Bird destructor  
Whinny...POSSO VOLARE...Horse costructor..., Bird Costructor..., Pegaso Costructor...

# Multiple constructors

Se Pegaso deriva da uccello e cavallo che richiedono parametri di inizializzazione ,Pegaso li inizializza entrambi

```
#include <iostream.h>
Typedef int HANDS;
enum COLOR {red,green,blue,yellow,white,blackbrown};
enum BOOL {FALSE,TRUE};
class Horse
{
Public:
Horse(COLOR color,HAND height);
virtual ~Horse(){cout:::"horse destructor";}
virtual void Whinny() {cout<<"whinny!";}
virtual HANDS GetHeight() {return itsHeight;}
virtual COLOR GetColor() {return itsColor;}
private:
HANDS itsHeight;
COLOR itsColor;
};
Horse::Horse(COLOR color,HAND height):
itsColor(color),itsHeight(height){cout<<"Horse constructor";}
class Bird
{
Public:
Bird(COLOR color,HAND height);
virtual ~Bird(){cout:::"bird destructor";}
virtual void Chirp() {cout<<"Chirp!";}
virtual void fly() {cout<<"POSSO VOLARE";}
virtual COLOR GetColor() {return itsColor;}
virtual BOOL GetMigration() {return itsMigration;}
private:
COLOR itsColor;
BOOL itsMigration;
};
```

```

Bird::Bird(COLOR color,BOOL migrates):
itsColor(color),itsMigration(migrates){cout<<"Bird constructor";} //implementazione costruttore dell'uccello analogo

Class Pegaso:public Horse,public Bird
{
Public:
void Chirp () {Whinny();}
Pegaso(COLOR,HANDS,BOOL,long);
~Pegaso() {}
virtual long GetNumberBelievers()
{ return itsNumberofBelievers;}
private:
long itsNumberofBelievers;
};

Pegaso::Pegaso(COLOR acolor,HANDS aheight,BOOL amigrates,long NumBelieve): //implementazione inline e
    inizializzazione dei parametri delle classi base
Horse(aColor,aheight),Bird(aColor,amigrates),itsNumberofBelievers(NumBelieve)
{
Cout <<"Pegaso constructor";
}
int main()
{
Pegaso *pPeg=new Pegasus(Red,5,TRUE,10);
pPeg->Fly(); //POSSO VOLARE
pPeg->Chirp(); //Whinny!
pPeg->GetHeight() //5
delet pPeg;
===== IMPORTANTE DA NOTARE L'AMBIGUITA' GetColor() e' un
metodo sia del Cavallo che dell'uccello e quindi in Pegaso il suo utilizzo non si capisce bene come effettuarlo (cioe'
quale delle due componenti inherisce Pegaso per questo tipo di azione
COLOR currentcolor=pPeg->GetColor(); hai un errore di compilatore che non dsa se prendere Bird::GetColor o
Horse::GetColor
In Questo caso risvolvi con il PATH COMPLETO
COLOR currentcolor=pPeg->Horse::GetColor();

```

# Variabili statiche e puntatori di funzioni

# Variabili statiche

**Variabili globali(public)** → tutti possono utilizzarle (rischioso)

**Variabili locali(private,protected)** → solo alcuni oggetti (o catene di oggetti se ineriscono)

ESISTE UN 3 tipo di variabili che sono un compromesso fra le 2:

**Variabili statiche(static)** → visibili solo in un tipo di classe

Se per esempio volessi sapere quanti oggetti di un certo tipo (=classe) sono creati in un programma, piu' che creare una variabile che la puo' vedere solo il cliente di quel dato oggetto, creo una variabile che si puo' chiamare all'interno di tutti quegli oggetti di uno stesso tipo e quindi appartenenti ad una data classe.

## ESEMPIO SULL'UTILIZZO DI VARIABILI STATICHE

```
class Cat
{
Public:
Cat(int age):itsAge(age) {HowManyCats++;} //ogni volta che creo un oggetto gatto incremento di uno questa variabile
                                         //intera che posso vedere perche' e' un oggetto di tipo gatto!!
virtual ~Cat() {HowManyCats--;}
virtual int GetAge() {return itsAge;}
Virtual void SetAge(int age) {itsAge=age;}
static int HowManyCats; //non sta in un oggetto in particolare ma nella intera classe e siccome e' pubblica puo' essere
                       //raggiunta da qualunque funzione nel main program (telepaticafunction)
private:
int itsAge;
};

-----/inizia il programma
Int Cat::HowManyCats =0; //implementazione della variabile statica
int main()
{
int MaxCats=5; int I;
Cat *CatHouse[MaxCats];                                //array di 5 puntatori di gatti
for (I=0,I<MaxCats;I++)
{
    CatHouse[I]=new Cat(I);
    TelepaticaFunction( );
}
for (I=0; I<MaxCats;I++)
{
delete CatHouse[I];
TelepaticaFunction();
}
Return 0;
}
Void TelepaticaFunction() { cout <<"ci sono"<<Cat::HowManyCats<<"Gatti/n";}
```

# Metodi statici

Se **HowManyCats** fosse una variabile privata si avrebbe bisogno di Funzioni accessorie pubbliche e statiche:

nella dichiarazione di prima allora

**static int GetHowManyCats(){ return HowManyCats;}**

Nel main potresti chiamare questo metodo in due modi:

**Cat theCat;**

**int quanti=theCat.GetHowManyCats(); //accesso tramite un  
oggetto concreto (theCat)**

oppure

**Int quanti= Cat::GetHowManyCats(); //accesso tramite una classe  
//generica**

# Puntatori di funzioni

Si puo dichiarare una variabile puntatore ad una funzione e quindi invocare la funzione usando questo puntatore.

IL PRINCIPALE MOTIVO E' CHE HAI LA POSSIBILITA' DI CREARE PROGRAMMI CHE DECIDONO QUALE FUNZIONE CHIAMARE IN BASE ALLA DECISIONE DELL'UTENTE (PICK-UP IN UNA LISTA DI POSSIBILI FUNZIONI) Qui sta ancora l'essenza *event driven* di cui (pensa di nuovo ai vari tipi di finestre, dialog-box, liste ecc. che puoi invocare con un puntatore da una lista di possibili finestre=funzioni)

La sintassi e' analoga al puntatore di una variabile:

typo variabile cui punta \* pName ;

La differenza e' che di una funzione oltre al tipo di ritorno hai anche una lista di parametri e quindi la sintassi e' leggermente differente

return type function (\*pName) (type 1input ,type 2input, ...)

es: long (\*pFunc) (int) puntatore ad una funz. che prende un intero e ritorna un long

controesemp: long \* pFunc (int) funzione che prende un intero e ritorna il puntatore ad un long...

## ESEMPIO PUNTATORI DI FUNZIONI

```
#include<iostream.h>
void square(int &,int &);
void Cube(int &, int &);
void Swap(int&,int &);           //Prototipi di 5 funzioni tutte compatibili alla definizione del puntatore pFunc
void GetValues(int &,int &);
void PrintValues(int &,int &);
enum BOOL(FALSE,TRUE);

int main()
{
void (*pFunc) (int&,int &);
BOOL fQuit=FALSE;
int Val1=1,Val2=2;
while(fQuit==FALSE)
{
cout<<"(0)Quit (1)Cambia (2) Quadra (3) Cuba (4) Scambia: ";
cin>>choice;
switch (choice)
{
    case 1:pFunc=GetValues; break;
    case 2: pFunc=square;break;
    case 3: pFunc=cube;break;
    case 4: pFunc=Swap;break;
    default:fQuit=TRUE;break;
}
if(fQuit) break; //esce anche dal while
PrintValues(Val1,Val2);
pFunc(Val1,Val2);
PrintValues(Val1,Val2);
}
Return 0;
}                                //RISPARMIAMO L'IMPLEMENTAZIONE DELLE 5 FUNZIONI che sono auto esplicanti
//NB LA STESSA COSA LA POTEVI OTTENERE SENZA PUNTATORI MA IL PROGRAMMA E'+ESPLICITO E DIRETTO.
```

# Altri esempi di puntatori di funzioni

**Vettore di puntatori di funzioni:**

**void (\*pFuncArray[dim]) (int &,int \*)**

**Puntatore ad un metodo di una classe:**

**type (class\_name::\*pFunc) (type 1inpu, type 2inp....)**

**Vettore di puntatori di funzioni membro  
(metodo):**

**void (class\_name::\*pName[dim]) (int &,long &,BOOL)**

# Preprocessore, Macros & Templates

(Cenni)

Giusto per non trovarsi un giorno  
certi simboli senza sapere cosa  
significano!

**OGNI VOLTA CHE COMPILI PRIMA DI TUTTO ATTIVI IL PREPROCESSORE CHE GUARDA NEL CODICE SORGENTE LE ISTRUZIONI DOVE CI SONO #.**

**L'effetto di queste istruzioni e' un cambiamento del tuo sorgente file in un temporaneo e non visibile file (che sara' letto dal compilatore).**

**#include CHIEDE PER ESEMPIO DI ANDARE A CERCARE IL FILE CHE RISPONDE AL NOME <> E APPENDERLO IN QUEL PUNTO IN TUTTA LA SUA STESURA COME SE LO AVESSI DIGITATO TU.**

**Altro comando**

**#define definisce una sostituzione di stringa es:**

**#define big 512 sostituira' la stringa big con 512 ogni volta che ne vede una nel programma (int myArray[big]; → int myArray[512];)**

**Un uso tipico di #define e' comunque per dichiarare se una certa stringa e' stata definita e in seguito testare se questa stringa e' stata definita oppure no per fare le cose di conseguenza con l'altra istruzione :**

**#ifdef big**

**(TRUE OR FALSE??) fai le cose di conseguenza....o non farle**

**#endif**

**Puoi complicare la logica con un bel #else (un applicazione e' ad esempio nell'inclusione di file header.hpp per piu' di una volta che potrebbero portare ad errori, cosa possibile se hai una ragnatela di classi chiamate che spesso (a loro volta) possono fare capo a comuni classi:**

**QUINDI UN eventuale pippo.h, prima di ridefinirlo una seconda volta metti degli #ifdef che controllano se la definizione e' gia' avvenuta oppure no e ti regoli di conseguenza.**

#undef e' un'altra parola chiave che funziona da antidoto a #define  
#ifndef controlla se una stringa NON e' stata gia' definita  
Combinando #ifdef,#ifndef,#define,#else etc.etc Puoi ad esempio creare un programma che compila differenti codici a seconda se ti trovi su una piattaforma DOS o Linux o Windows e a seconda della versione del compilatore che usi.....

#define puo' essere usata per creare delle funzioni MACROs che sono simboli che prendono parametri (tutto in una linea!!!!)

Il preprocessore sostituirà la stringa ogni volta che ne riconoscera' il simbolo:  
`#define TWICE(x) ((x)*2)`  
e quindi se tu scrivi nel tuo codice `TWICE(4)` alla fine il compilatore vedrà `8!!`

Altra macro:

```
#define WRITESTRING(x) cout<<#x
```

Dove #x mette le " " alla x

Allora la chiamata WRITESTRING(ciao a tutti); si tradurrà :  
`cout<<"ciao a tutti";`

Ancora: (noto che##=operatore del preprocessore che concatena una due stringhe)

```
#define fPRINT(x) f ## x ##Print→fPRINT(1)=f1Print, fPRINT(2)=f2Print .....
```

In questo caso si potrebbe riservare molto utile la MACRO per la concatenazione di stringhe.

Supponi che hai una classe list che lavora molto bene su qualunque tipo di oggetto (animal,car,cat etc.)

Puoi usare la generica classe list applicandola singolarmente ai vari tipi di oggetti con l'incubo di dover cambiare non una volta ma n-volte (n=numero di oggetti cui applichi la classe CatList,CarList,DocList,AnimalList....) oppure istanziarla una volta per tutte in un'unica linea con una MACRO

```
#define Listof(type) class Type##List\
```

```
{\
```

```
Public:\
```

```
Type##List() {}
```

```
Private:\ \ server per andare a capo ma informare il  
preprocessore che tutto avviene in una linea
```

```
Int itsLength;\
```

```
}.....
```

La chiamata Listof(Animal) fa la cosa giusta su Animali e lo stesso dicasi per Listof(Car) ..

Basta solo cambiare una parola!!!

---

Altre MACRO provviste \_DATE\_ (sostituisce la data odierna) \_FILE\_  
(nome del file) \_TIME\_ (ora) \_LINE\_ (linea di codice)

# TEMPLATES

Come detto se suppongo che PartList e' un tipo di classe che lavora bene per un generico oggetto parte e la volessi applicare a cani gatti macchine devo :

- 1.creco una classe base list da cui derivo da essa partlist
- 2.derivo carlist nello stesso modo tagliando e copiando da partlist
- 3.ogni volta che cambio in una classe derivata cambio in ogni altra classe derivata..(NIGHTMARE)

Uno modo di sorpassare lo abbiamo visto con le MACROS che pero' come ogni cosa del preprocessore sono type-safe.(non specificano il tipo)

**Temlate ti permettono di isrtuire il compilatore come creare una lista di ogni tipo di cose piuttosto che creare un set di tipo-specifico liste.**

**Un comune componente delle librerie di C++ e' la classe array.**

**Come avrai capito e' tedioso e inefficiente creare una classe vettore che usi per gli interi, un'altra per I reali un 'altra per caratteri e cosi' via.**

**I templates ti permettono di dichiarare una parametrizzata classe di vettori con il tipo delle cose da mettere nell'array stabilito di volta in volta dal parametro tipo che dichiari quando instanzi il vettore specifico.**

**Instanziazione=operazione con cui crei uno specifico tipo da un generico template.**

**Si dichiara un parametrizzato template per la classe "vettore" (template per un array) con la seguente sintassi:**

```
template <class T> //prima della dichiarazione della classe di cui vuoi il template  
class array {
```

**Public:**

array();

.../dichiarazione completa/...

};

**I parametri che dichiari nel template sono quelle cose che diventano concrete quando instanzi il template: in questo caso T=type perche' dopo la parola chiave class che e' appunto il tipo di qualunque**

**Chiaramente dopo puoi usare la classe parametrizzata vettore applicandola a qualunque tipo di oggetti sia interi (array di interi) che Gatti (array di gatti) con la seguente sintassi di instanziazione:**

**Array<int> anIntArray;**

**Array<Cat> aCatArray;**

**for (1=0;I<5;I++) anIntArray[I]=I;**

**La dichiarazione di un template e' identica a quella di una classe con la caratteristica di genericita'del tipo degli oggetti che essa tratta (es.anziche avere int potrei trovarmi "T" che diventa int in un array di inter.**

**Non reputo sia questo il momento di parlare di come implementare un template ma immaginerai che prevede l'uso di copy constructor.**

**Il solo motivo per cui ho parlato di un template si basa sull'esperienza in ORCA in cui trovai strutture template ampliamente usate. Ora anche tu sai cosa sono.....**