Encapsulation

L'Encapsulation permette di nascondere i dettagli implementativi che caratterizzano una classe. Questa caratteristica mette in evidenza il concetto di '**astrazione**' infatti solo attraverso l'interfaccia degli oggetti è possibili accedere a metodi e attributi. I dettagli implentativi possono avere diversi gradi di visibilità tramite i modificatori di visibilità.

1. Public: sono visibibili all'esterno della classe.
2. Protected: sono visibili all'esterno della classe e anche nelle loro sottoclassi chde derivano da esse.
3. Private: questo modificatore di visibilità evita che il metodo o l'attributo sia visibile all'esterno della classe.

Funzioni e classi friend

Tale modificatore (“amico”) permette alle funzioni/classi di accedere a metodi/attributi privati e protetti definiti in altre classi. In C++ *gli amici si toccano le parti private*.

```
class B {
private: int i;
public: int f(int n, A* a);
}; class A { private: int i; public: friend int B::f(int n, A* a); };
int B::f(int n, A* a) {return i + a->i + n; }
```

Riceve un reference di tipo A e tramite questo reference accede all'attributo privato lo modifica sommando n ad i

Inheritance

Indica la possibilità di estendere una classe da un altra.

```
class Derived : inheritance_modifier Base
```

dove `inheritance_modifier` può essere

TIPO DI EREDITARIETA'				
VISIBILITA'		public	protected	private
	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

Una volta stabilito come definire l'inheritance_modifier si devono definire i costruttori e i distruttori. Per invocare il costruttore padre si fa in questo modo

```
Derived::Derived(base paramb, derived paramd, ...)
: Base(base param)
{ paramd = ...; }
```

Per quanto riguarda il distruttore esso in una classe base deve essere dichiarato virtual in modo da poter sfruttare il polimorfismo.

```
class Derived : public Base
{
     virtual ~Derived();
}
```

Objects, with dynamic lookup of virtual functions

In C++ come in altri linguaggi il polimorfismo può essere realizzato a compile-time attraverso l'uso dei template e a run-time attraverso le virtual functions e il metodo del dynamic lookup attraverso il quale un metodo vienen selezionato dinamicamente a run time in accordo con l'implementazione scelta a compile time.

Consente a oggetti di tipi diversi di rispondere in maniera differente alla medesima chiamata di funzione. Il dynamic lookup è reso possibile grazie al fatto che un puntatore ad un occorrenza di una classe base può puntare a qualsiasi occorrenza di una sottoclasse.

```
 class X {
 public void foo(){ cout<<”X.foo()”;}
 }
 class Y :public X{
 public void foo(){ cout<<”Y.foo()”;}
 }
 main(){
 X x;
 Y y;
 X* p;
 p->foo(); // p è di tipo X e quindi viene invocato il metodo foo() della
classe X
```

```
 p = &y; // p (di tipo X) punta ad un oggetto di tipo y
 p->foo(); // nonostante p punti ad un oggetto Y è ancora di tipo X e quindi
 //viene eseguito X::foo();
 }
```

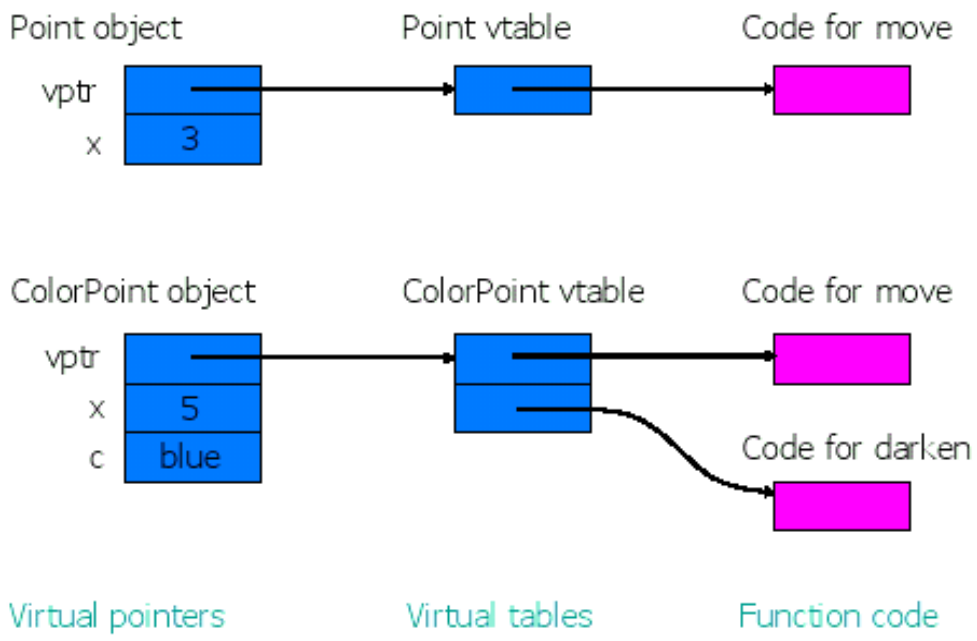
Questo perchè la funzione foo() non è virtuale e quindi viene chiamata quella di X.

Se la classe base viene modificata così...

```
class X {
 public virtual void foo(){ cout<<”X.foo()”;}
 }
main(){
 X x;
 Y y;
 X* p;
 p->foo(); // p è di tipo X e quindi viene invocato il metodo foo() della classe X
 p = &y; // p (di tipo X) punta ad un oggetto di tipo y
 p->foo(); // ora p punta ad un oggetto y, la funzione foo è virtuale e viene
 // eseguita Y::foo();
 }
```

si indica che foo() assume un comportamento virtuale.

Questo esempio illustra il polimorfismo, la medesima chiamata a p->f() provoca l'esecuzione di funzioni differenti; la funzione viene selezionata in accordo con la classe dell'oggetto a cui punta p. Questo comportamento è detto dynamic binding / dynamic lookup perchè l'associazione della chiamata con il codice da eseguire realmente viene rimadata al momento dell'esecuzione. La dichiarazione virtual delle funzioni elemento permette di eludere la regola che il tipo del puntatore definito staticamente determina la funzione elemento da eseguire.



La figura mostra come si possa schematizzare lo schema di una chiamata in C++. L'oggetto viene rappresentato solo tramite i suoi attributi. I metodi sono definiti in una tabella (vtable) composta dal metodo e dall'indirizzo del codice associatato alla funzione.

Se la funzione move() non fosse stata ridefinita nella classe ColorPoint la riga della vtable di move avrebbe puntato all'indirizzo di move della classe Point.

Non-virtual functions

Se le funzioni non sono virtuali come vengono risolte?

Una volta che il compilatore ha generato il codice della funzione esso la inserisce nella symbol table. Alla chiamata della funzione esso viene richiamato estraendo dalla symbol table l'indirizzo da passare alla call ed eseguito come codice normale.

CONTROLLARE QUESTA PARTE (non virtual e il commento alla figura)

Pointer this

Il puntatore this come in Java ha il compito di rappresentare l'oggetto.

Esempio virtual vs overloading/overriding

```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");}; };
class child : public parent { public:
```

```
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");};   };
main() {
    parent p; child c; parent *q;
                p.printclass();    p.printvirtual();    c.printclass();
     c.printvirtual();
     // ==========================================
     q = &p; q->printclass(); q->printvirtual();
     q = &c; q->printclass(); q->printvirtual();
}
Output: p p c c p p p c
```

Function call binding

- Early binding: a compile time
- Late binding: a run time, meno efficiente a causa di più chiamate assembler, gestione della memoria aggiuntiva a causa delle tabelle virtuali

Subtyping

Inheritance Is Not Subtyping

In C++ una classe X è un sottotipo di Y solo se essa deriva pubblicamente da Y

```
class Y{};  class X : public Y{}  // subtiping
```

Subtyping is a relation on types that allows values of one type to be used in place of values of another.

Abstract Classes

In alcune situazioni non è possibile definire un comportamento per le funzioni.

(L'area del poligono). Le funzioni astratte vengono realizzate tramite il costrutto

```
virtual function_decl = 0;
```

porta ad avere una classe incompleta. Tale funzione viene definita funzione virtuale pura.

Non è possibile instanziare una classe di questo tipo. E' necessario estenderla e implementare le funzioni virtuali pure.

Multiple Inheritance

C++ rispetto a Java permette di estendere una classe da più classi.

Ad esempio la classe ColoredRectangle estende le classi Poligon e Color.

La possibilità di estendere una classe da altre classi però comporta la nascita di diversi problemi.

1. Name Clashes

```
class A {
     public:
     void virtual f() { … }
};
class B {
     public:
        void virtual f() { … }
  };
class C : public A, public B { … };
C* p;
p->f(); // quale chiamo
```

In caso di chiamata ad f(), il compilatore non riesce a risolverla in quanto non sa a quale riferirsi e genera errore.

Sono state proposte tre soluzioni

Language resolves name conflicts with arbitrary rule

- Implicit resolution, attraverso la quale il linguaggio risolve il conflitto con regole arbitrarie.
- Explicit resolution, spetta al programmatore risolvere il conflitto che si genera.
- Disallow name clashes, non si permette al linguaggio di supportare l'ereditarietà multipla (Java)

Nessuna soluzione è la migliore! (C++ usa la explicit resolution)

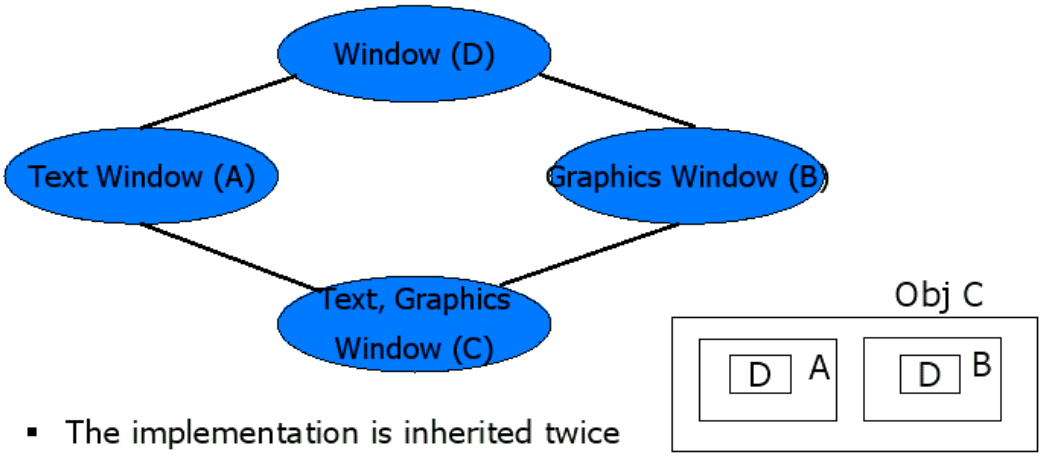
Ecco come in C++ si risolve l'ambiguità generata:

```
class C : public A, public B {
     public:
        void virtual f( ) {
                A::f( ); // Call A::f(), not B::f();}
}
```

L'operatore di scope resolution (:) permette di accedere al metodo specificato

dal programmatore in accordo con l 'explicit resolution.

2. Diamond inheritance in C++

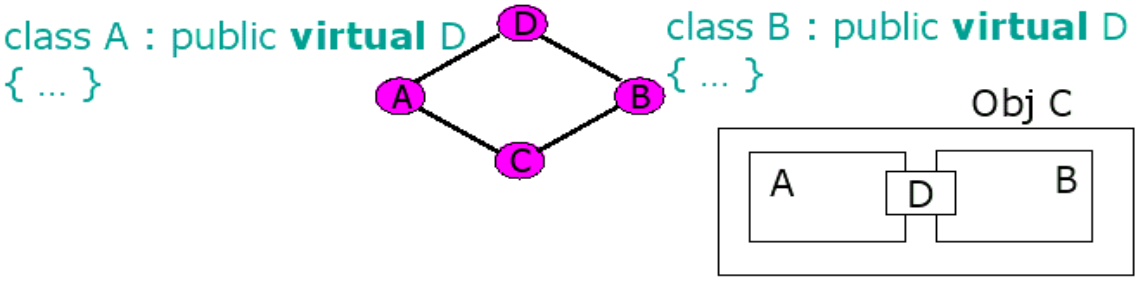


Questo caso mostra come possa sorgere il caso del diamond inheritance cioè la situazione in cui una classe deriva da altre due che a loro volta derivano da una stessa classe.Questa situazione determina la nascita di copie degli stessi metodi basi che collliderebbero generando l'interruzione della compilazione. La soluzione apportata a questa problema consta nell'utilizzo di *virtual base classes.*

Si devono dichiarare le classi A e B....

```
class A : public virtual D

class B : public virtual D
```



Additions not related to objects

Type bool

Introdotti per evitare l'ambiguita causata dagli int.

```
bool b1, b2, b3;

int j, k;
```

```
b1 = 3*5; // b1 = true        b2 = 0; // b2 = false

j = b1; //  j=1               j = b1 || b2; // j = 1

j = b1 && b2; // j = 0        b1 = j == 0; // b1 = true
```

Reference variables

Un *riferimento* (&) è come un puntatore costante, che viene automaticamente dereferenziato.

```
// Riferimento ordinario isolato:

int y; int& r = y; // Quando viene creato un riferimento

//deve essere inizializzato per riferirsi ad un oggetto vero.

//Tuttavia si può anche scrivere:

const int& q = 12;
```

Il compilatore alloca uno spazio di memoria, lo inizializza con il valore 12 e fissa un riferimento a questo spazio di memoria. Il punto è che un riferimento deve essere legato a *qualche altro* pezzo di memoria. Questo comporta che una reference variable sia inizializzata appena definita e sia costante (reassegnamento non permesso).

Function references

Il passaggio per riferimento è più performante rispetto a quello per valore perchè passa l'indirizzo dell'oggetto e non una copia. Una modifica al paramtro corrisponde a una modifica della variabile. Per sfruttare l'efficienza del passaggio per reference senza correre il rischio di modifica della variabile si può passare come constante il parametro

```
// A good style

int f(const int& t_in) {  t_in = 99; // ERROR}
```

Il passaggio per riferimento può essere realizzato nel modo precedente oppure nella modalità C-style

```
void foo(Param* p);

// pointer based

// vantaggi / svantaggi dell'uso dei pointer

void foo(Param& p);

// reference based

// no null checking
```

Return-by reference

E'possibile ritornare anche un reference

```
int& f(int& x){ return x};
```

Se si vuole evitare che l'indirizzo venga modificato

```
const int& g(int x); h(j) = 99;
```

Costruttore di copia

Uno dei costruttori più importanti da modificare tramite overloading è il costruttore di copia. Tale costruttore evita che sorgano problemi quando si usa un oggetto per inizializzarne un altro.

Normalmente quando si usa un oggetto per inizializzarne un altro, C++ prevedel'esecuzione di una copia bit a bit, cioè la destinazione sarà una copia identica dell'oggetto utilizzato per l'inizializzazione. Questo comportamento è appropriato nella maggior parte dei casi, in alcuni casi (es. allocazione dimanica) non si deve usare la copia bit a bit.

Si supponga che la classe Class allochi un'area di memoria. Sia A un'instanza di Class

```
Class A;

Class B = A;
```

Se viene eseguita una copia bit a bit, allora B sarà una copia esatta di A, cioè B utilizzerà la stessa memoria allocata per A, non una propria area distinta.

Se venisse invocato il distruttore di A (che rilascia l'area allocata) automaticamente verrebbe eliminata anche l'area “allocata” da B.

Per risolvere questo problema il C++ consente di creare un costruttore di copie che il compilatore impiega quando si usa un oggetto per inizializzarne un altro. Esso viene inmpiegato al posto del costruttore bit a bit.

```
classname (const classname& obj){}
```

oppure facendo l'overloading degli operatori in questo modo (ridefinisce il comportamento dell'operatore =)

```
classname& operator=(const classname& obj){};
```

E' importante comprendere che C++ definisce due diversi tipi di situazioni in cui a un oggetto viene assegnato il valore di un altro oggetto.

Il primo caso è l'assegnamento, il secondo è l'inizializzazione che può verificarsi in tre modi:

- oggetto inizializza esplicitamente un altro oggetto (class x = y;)
- eseguita copia di un oggetto che deve essere passato ad una funzione (func(y);)

- creazione di un oggetto temporane (val. restituito da una funzione) y = funct();

```
#include <iostream.h>
#include <string.h>
class Message {
     char *subject, *message;
     //A function to initiale data members
     init_message(const char *,const char *);
     public:
     //A constructor
     Message(const char *, const char * = "");
     //The copy-constructor
     Message(const Message & m);
     //Overloading of the assignment operator =
     const Message& operator=(const Message &);
     //The destructor
     ~Message()
};
////////////////////////////////////////////////////////////////////////
//                      IMPLEMENTATION FILE
////////////////////////////////////////////////////////////////////////
//A function to initiale data members
Message::init_message(const char *s, const char *m){
   subject = new char [strlen(s)+1];
   strcpy(subject,s);
   message = new char [strlen(m)+1];
   strcpy(message,m);
}
Message(const char *s, const char * m){  init_message(s,m);}
~Message(){  delete subject;  delete message;}
//The copy constructor
Message::Message(const Message & m){  init_message(m.subject,m.message);}
//Overloading of the assignment operator =
const Message& Message::operator=(const Message & m){
```

```
  // always check for self-assignment
  if (this == &m) return *this;
  delete subject;
  delete message;
  init_message(m.subject,m.message);
  return *this; //the left element is returned
}
```