

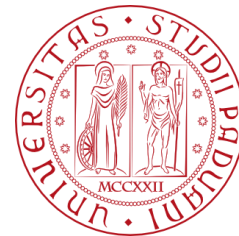
# Tutorato 4

22/11/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



```
1  class dataora: public Orario{
2      public:
3          int Giorno() const;
4          int Mese() const;
5          int Anno() const;
6      private:
7          int giorno;
8          int mese;
9          int anno;
10 }
```

- **orario**: classe  
base/superclasse/supertipo
- **dataora**: classe  
derivata/sottoclasse/sottotipo

Tutti i membri di **orario** sono ereditati da dataora.

Ogni oggetto della classe derivata è utilizzabile anche come oggetto della classe base.

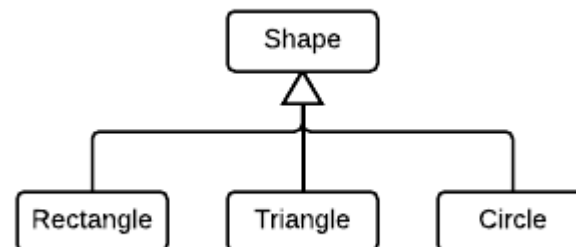
Relazione *is-a*: Dataora «è un» orario (relazione di *subtyping*) e sarà suo *sottooggetto*

In questo senso, si comincia a parlare di **gerarchie**.

Possiamo classificare i sottotipi *diretti* (subito derivati) da quelli *indiretti* (vari gradi di derivazione).

Inoltre, l'ereditarietà è utilizzata per:

- Estensione
  - `dataora <: orario`
- Specializzazione
  - `QPushButton :< QComponent`
- Ridefinizione
  - `Queue <: List`
- Riutilizzo di codice




# Ereditarietà e polimorfismo


Le relazioni vengono convertite in puntatori/riferimenti in modo **polimorfo** (cambiando tipo a seconda del tipo di contesto).

Possiamo distinguere:

- tipi statici – determinati univocamente a tempo di invocazione (a sx dell'uguale)
- tipi dinamici – determinati a runtime a seconda del contesto (a dx dell'uguale «se serve»)



```
1 D d; B b;  
2 D* pd = &d;  
3 B* pb = &b;  
4 pb = pd; // conversione D* -> B*
```



```
1 //conversione implicite che valgono  
2 D --> B // oggetti  
3 D& --> B& // riferimenti  
4 D* --> B* // puntatori
```

Una classe derivata ha accesso alla parte privata della classe base? No.

Possiamo usare la keyword **protected** per rendere inaccessibile all'esterno le variabili ma farle vedere alle sottoclassi.

```
1  dataora::set2024(){  
2      sec = 0; //illegale  
3      giorno = 1;  
4      mese = 1;  
5      anno = 2024;  
6  }
```

In questo modo, definiamo quali dati vogliamo rendere visibili all'esterno.

```
1 class B{
2     protected:
3         int i;
4         void protected_printB() const{cout << ' ' << i;}
5     public:
6         void printB() const{cout << ' ' << i;}
7 };
8
9 class D: public B{
10     private:
11         double z;
12     public:
13         void stampa(){
14             cout << i << ' ' << z;
15         }
16 };
```

```
1 static void stampa(const V& b, const D& d){
2     cout << ' ' << b.i; // Illegale - b.i è protetto
3     b.printB();
4     b.protected_printB(); // Illegale - b.protected_printB() è protetto
5     cout << ' ' << d.i;
6     d.printB(); // OK
7     d.protected_printB(); // OK
8 }
```

# Tipi di ereditarietà



Esistono diversi tipi di derivazione (tendenzialmente, si usa quella pubblica).

Derivazione	public	protected	private
Membro			
private	inaccessibile	inaccessibile	inaccessibile
protected	protetto	protetto	privato
public	pubblico	protetto	privato

*Note:*

- Derivazioni protette e private non supportano l'ereditarietà di tipo (e non inducono conversioni implicite; la derivazione pubblica le permette).
- I membri protetti rappresentano comunque una violazione dell'information hiding.
- L'ereditarietà privata significa semplicemente «esporre solo alcuni campi della classe base controllando precisamente a quali campi voglio accedere»



# Tipi di ereditarietà



```
1 class C{
2 private:
3     int priv;
4 protected:
5     int prot;
6 public:
7     int publ;
8 };
9
10 class D: private C{
11     // prot e publ diventano privati
12 };
13
14 class E: protected D{
15     // prot e publ diventano protetti
16 };
```

```
1 class F: public D{
2     // prot e publ sono qui inaccessibili
3 public:
4     void fF(int i, int j){
5         prot = i; // Illegale
6         publ = j; // Illegale
7     }
8 };
9
10 class G: public E{
11     // prot e publ rimangono qui protetti
12 public:
13     void fG(int i, int j){
14         prot = i; // OK
15         publ = j; // OK
16     }
17 };
```





# Tipi di ereditarietà



```
1  class C{
2      private:
3          int x;
4      protected:
5          char c;
6      public:
7          float f;
8  };
9
10 class D: private C {}; //derivazioone privata
11 class E: protected C {}; //derivazione protetta
12 // nessuna conversione implicita D -> C e E -> C
13 int main(){
14     C c, *pc;
15     D d, *pd;
16     E e, *pe;
17     // c = d; // Illegale: C inaccessibile
18     // c = e; // Illegale
19     // pc = &d; // Illegale
20     // pc = &e; // Illegale
21     // C& rc = d; // Illegale
22 }
```



# Tipi di ereditarietà



```
1  class Base{
2      int x;
3      public:
4      void f() { x = 0; }
5  };
6
7  class Derivata: public Base{
8      int y;
9      public:
10     void g() { y = 3; }
11 };
12
13 int main(){
14     Base b; Derivata d;
15     Base * p = &b; Derivata *q = &d;
16     p->f(); //OK
17     p = &d; // Derivata* è ora TD (Tipo Dinamico) di P
18     p->f(); //OK
19     p->g(); // ha senso? Passiamo da TD a TB (Tipo Base) ed è imprevedibile
20     q->g(); //OK
21     q->f(); //OK
22 }
```



# Tipi di ereditarietà



```
1 class C{
2     public:
3     int x;
4     void f() {x = 4;}
5 };
6
7 class D: public C{
8     public:
9     int y;
10    void g() {y = 5;}
11 };
12
13 class E: public D{
14     public:
15     int z;
16     void h() {z = 6;}
17 };
```

```
1 int main(){
2     C c; D d; E e;
3     c.f(); d.g(); e.h();
4
5     D* pd = static_cast<D*>(&c); //pericoloso ma legale
6     cout << pd->x << " " << pd->y << endl;
7     // errore runtime o stampa: 4 (Valore Intero imprevedibile)
8
9     E& pe = static_cast<E&>(&c); //pericoloso ma legale
10    cout << pe.x << " " << pe.y << " " << pe.z << endl;
11    // errore runtime o stampa: 5 (Valore Intero imprevedibile)
12
13    C* pc = &d; pd = static_cast<D*>(pc); //OK
14    cout << pd->x << " " << pd->y << endl; // stampa: 5
15
16    D& rd = e; E& re = static_cast<E&>(rd); //OK
17    cout << re.x << " " << re.y << " " << re.z << endl; // stampa: 6
18 }
```



# Tipi di ereditarietà



```
1 //le amicizie non si ereditano!
2
3 class C{
4     private:
5         int i;
6     public:
7         C(): i(1) {}
8         friend void print(C);
9 };
10
11 class D: public C{
12     private:
13         double z;
14     public:
15         D(): z(2.0) {}
16 };
17
18 void print(C x){
19     cout << x.i << endl;
20     D d;
21     cout << d.z << endl; //Illegale - D privato
22 }
23
24 int main(){
25     C c; D d;
26     print(c);
27     print(d);
28 }
```



```
1 //cerchiamo di definire la somma di orari
2 dataora dataora::operator+(const orario& o) const
3 {
4     dataora aux=*this;
5     //aux.sec = sec + o.sec; //errore - accediamo a campi inaccessibili
6     aux.sec=sec+3600*o.Ore()+60*o.Minuti()+o.Secondi();
7     if(aux.sec>=86400)
8     {
9         aux.sec=aux.sec-86400;
10        aux.AvanzaUnGiorno();
11    }
12    return aux;
13 }
14
15 orario o1, o2;
16 ...
17 dataora x = o1 + o2; // Illegale
```

# Ridefinizione - overloading



- Ridefinizione: prendo i metodi ereditati della classe base e «ridefinisco» il comportamento, estendendolo per i miei interessi.
- Viene anche definito come «overloading» (sovraccarico):
- Una ridefinizione del metodo nasconde sempre tutte le versioni «ridefinite prima» del metodo e «prendiamo quella che si serve» con scoping: *name hiding rule*

Two or more functions can have the same name but different parameters; such functions are called function overloading in c++.



# Ridefinizione - overloading



```
1  class B{
2      protected:
3          int x;
4      public:
5          B(): x(2) {}
6          void print() {cout << x << endl;}
7      }
8
9  class D: public B{
10     private:
11         int x;
12     public:
13         D(): x(5) {}
14         void print() {cout << x << endl;}
15         void printAll() {cout << B::x << " " << x << endl;}
16     }
17
18     int main(){
19         B b; D d;
20         b.print(); //stampa - 2
21         d.print(); //stampa - 5
22         d.printAll(); //stampa - 2 5
23     }
```



# Esercizio 1 – Cosa Stampa

```
1  #include "iostream"
2
3  using namespace std;
4
5  class C {
6      public:
7          int a;
8          void fC() { a=2; }
9  };
10 class D: public C {
11     public:
12         double a;
13         void fD() { a=3.14; C::a=4; }
14 };
15 class E: public D {
16     public:
17         char a;
18         void fE() { a='*'; C::a=5; D::a=6.28; }
19 };
```

```
int main() {
    C c; D d; E e;
    c.fC();
    d.fD();
    e.fE();
    D* pd = &d;
    E& pe = e;
    cout << pd->a << " " << pe.a << endl;
    cout << pd->a << " " << pd->D::a << " " << pd->C::a << endl;
    cout << pe.a << " " << pe.D::a << " " << pe.C::a << endl;
    cout << e.a << " " << e.D::a << " " << e.C::a << endl;
}
```



# Esercizio 1 – Soluzione

```
1  #include "iostream"
2
3  using namespace std;
4
5  class C {
6      public:
7          int a;
8          void fC() { a=2; }
9  };
10 class D: public C {
11     public:
12         double a;
13         void fD() { a=3.14; C::a=4; }
14 };
15 class E: public D {
16     public:
17         char a;
18         void fE() { a='*'; C::a=5; D::a=6.28; }
19 };
```

```
int main() {
    C c; D d; E e;
    c.fC();
    d.fD();
    e.fE();
    D* pd = &d;
    E& pe = e;
    cout << pd->a << " " << pe.a << endl;
    cout << pd->a << " " << pd->D::a << " " << pd->C::a << endl;
    cout << pe.a << " " << pe.D::a << " " << pe.C::a << endl;
    cout << e.a << " " << e.D::a << " " << e.C::a << endl;
}
```

```
1  /*
2  3.14 *
3  3.14 3.14 4
4  * 6.28 5
5  * 6.28 5
6  */
```

# Esercizio 2 – Cosa Stampa



```
1  #include "iostream"
2
3  using namespace std;
4
5  class C {
6      public:
7          void f() {cout << "C::f" << endl;}
8  };
9  class D: public C {
10     public:
11         void f() {cout << "D::f" << endl;}
12 };
13 class E: public D {
14     public:
15         void f() {cout << "E::f" << endl;}
16 };
```

```
1  int main() {
2      C c; D d; E e;
3      C* pc = &c;
4      E* pe = &e;
5      c = d;
6      c = e;
7      d = e;
8      d = c;
9      C& rc=d;
10     D& rd=e;
11     pc->f();
12     pc = pe;
13     rd.f();
14     c.f();
15     pc->f();
16 }
```



# Esercizio 2 – Soluzione



```
1  #include "iostream"
2
3  using namespace std;
4
5  class C {
6      public:
7          void f() {cout << "C::f" << endl;}
8  };
9  class D: public C {
10     public:
11         void f() {cout << "D::f" << endl;}
12 };
13 class E: public D {
14     public:
15         void f() {cout << "E::f" << endl;}
16 };
```

```
1  int main() {
2      C c; D d; E e;
3      C* pc = &c;
4      E* pe = &e;
5      c = d;
6      c = e;
7      d = e;
8      //d = c; NON COMPILA QUA
9      C& rc=d;
10     D& rd=e;
11     pc->f(); // C::f
12     pc = pe;
13     rd.f(); // D::f
14     c.f(); // C::f
15     pc->f(); // C::f
16 }
```



1. Viene sempre invocato per primo il *costruttore della classe base  $B$*
2. Successivamente, viene eseguito il *costruttore proprio della classe derivata  $D$*
3. Infine, viene eseguito il corpo del costruttore

Per quanto riguarda il costruttore standard:

1. Richiama il costruttore di default della classe base
2. Richiama i costruttori di default per tutti i campi dati di  $D$

1. Viene sempre invocato per primo il *distruttore della classe base  $B$*
2. Successivamente, viene eseguito il *distruttore proprio della classe derivata  $D$*
3. Infine, viene eseguito il corpo del costruttore

Per quanto riguarda il distruttore standard:

1. Richiama il distruttore di default della classe base
2. Richiama i distruttori di default per tutti i campi dati di  $D$

# Assegnazione con comportamento standard

```
1  #include <iostream>
2  using namespace std;
3
4  class B{
5      private:
6          int x;
7      public:
8          B(int k=1): x(k) {}
9          B& operator=(const B& a) {x = a.x;}
10         void print() const {cout << "x=" << x;}
11     };
12
```

```
1  class D: public B{
2      private:
3          int z;
4      public:
5          D(int k=2): B(k), z(k) {}
6          // assegnazione con comportamento standard
7          D& operator=(const D& x){
8              this->B::operator(x); // assegnazione per sottooggetto
9              z = x.z;
10         }
11         void print() const {B::print(); cout << "z=" << z;}
12     };
13
14 int main(){
15     D d1(4), d2(5);
16     d1.print(); cout << endl; // x=4 z=4
17     d2.print(); cout << endl; // x=5 z=5
18     d1=d2;
19     d1.print(); // x=5 z=5
20 }
```

# Esempio: Assegnazione standard

Si considerino le seguenti definizioni.

```
class Z {  
private:  
    int x;  
};  
  
class B {  
private:  
    Z x;  
};  
  
class D: public B {  
private:  
    Z y;  
public:  
    // ridefinizione di operator=  
    ...  
};
```

Ridefinire l'assegnazione `operator=` della classe `D` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `D`.



# Esempio: Copia standard

## Esercizio Costruttore

```
class A {  
private:  
    virtual void f() const =0;  
    vector<int*>* ptr;  
};  
  
class D: virtual public A {  
private:  
    int z;  
    double w;  
};  
  
class E: public D {  
private:  
    vector<double*> v;  
    int* p;  
    int& ref;  
public:  
    void f() const {}  
    E(): p(new int(0)), ref(*p) {}  
    // ridefinizione del costruttore di copia di E  
};
```

Si considerino le precedenti definizioni. Ridefinire (senza usare la keyword `default`) nello spazio sottostante il costruttore di copia della classe E in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di E.

**ridefinizione del costruttore di copia di E**



# Esercizio 3 – Cosa Stampa



```
1  #include <iostream>
2  using namespace std;
3
4  class C{
5      private:
6          int i;
7      protected:
8          int p;
9      public:
10         C() {cout << "C0 ";}
11         C(int x) {cout << "C1 ";}
12     };
13
14     class D: private C{
15         private:
16             int j;
17         public:
18             D(): C(2) {cout << "D0 ";}
19             D(int x): C(x) {cout << "D1 ";}
20     };
21
22     class E: public C{
23         private:
24             D d;
25         public:
26             int k;
27             E(): C(6) {cout << "E0 ";}
28     };
```

```
1  class F: public D{
2      protected:
3          E e;
4      public:
5          F(): D(3.2) {cout << "F0 ";}
6          F(float x) {cout << "F1 ";}
7      };
8
9      class G: public E{
10         private:
11             F f;
12             C c;
13         public:
14             G(): E() {cout << "G0 ";}
15             G(char x) {cout << "G1 ";}
16     };
17
18     int main(){
19         G g;
20     }
21
```



# Esercizio 3 – Soluzione



```
1  #include <iostream>
2  using namespace std;
3
4  class C{
5      private:
6          int i;
7      protected:
8          int p;
9      public:
10         C() {cout << "C0 ";}
11         C(int x) {cout << "C1 ";}
12     };
13
14     class D: private C{
15         private:
16             int j;
17         public:
18             D(): C(2) {cout << "D0 ";}
19             D(int x): C(x) {cout << "D1 ";}
20     };
21
22     class E: public C{
23         private:
24             D d;
25         public:
26             int k;
27             E(): C(6) {cout << "E0 ";}
28     };
```

```
1  class F: public D{
2      protected:
3          E e;
4      public:
5          F(): D(3.2) {cout << "F0 ";}
6          F(float x) {cout << "F1 ";}
7      };
8
9      class G: public E{
10         private:
11             F f;
12             C c;
13         public:
14             G(): E() {cout << "G0 ";}
15             G(char x) {cout << "G1 ";}
16     };
17
18     int main(){
19         G g;
20     }
21
```

```
1  C1 C1 D0 E0 C1 D1 C1 C1 D0 E0 F0 C0 G0
```



Possiamo fare in modo che quando il parametro è passato per riferimento, l'associazione tra oggetto di invocazione e metodo da invocare sia effettuata a tempo di invocazione. In questo senso, usiamo i metodi *virtual*.

```
1  orario::Stampa() {...}
2
3  dataora::Stampa() {...}
4
5  void printInfo(const orario& o) {
6      o.Stampa();
7  }
8  //creiamo un metodo virtuale che,
9  //in base al tipo di oggetto che viene passato, "cambia forma" =
10 // "polimorfismo"
11
12 class orario{
13     public:
14     virtual void Stampa();
15     ...}
16
17 void G (const orario& o) {
18     o.Stampa();
19 }
```

Una ridefinizione di un metodo virtuale viene definita overriding.

Si tratta del cosiddetto *legame dinamico (dynamic/late binding)*, quindi «associazione fatta a tempo di invocazione tra puntatore ed oggetto invocato».

In questo caso, occorre avere:

- Identica segnatura (tipo di ritorno e const incluso)
- Se lista argomenti identica ma cambia il tipo di ritorno → errore

```
1 class Base {
2 public:
3     virtual void f() { cout << "Base::f()" << endl; }
4 };
5
6 class Derived : public Base {
7 public:
8     virtual void f() { cout << "Derived::f()" << endl; } // overriding
9 };
```

Nota: alle volte, essendo che le gerarchie sono estese, è possibile «cambiare più forme» (si hanno a disposizione varie classi) → tipo *covariante* (da A, magari vado a B oppure a C)

# Esercizio 4: Modellazione

Si consideri il seguente modello concernente alcune componenti di una libreria grafica.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base astratta `Widget` i cui oggetti rappresentano un generico componente (un cosiddetto widget) di una Gui. Ogni `Widget` è caratterizzato da larghezza e altezza in pixels, e dall'essere visibile o meno.
  - `Widget` è astratta in quanto prevede il metodo virtuale puro `void setStandardSize()` che deve garantire il seguente contratto: `w->setStandardSize()` imposta la dimensione larghezza×altezza definita come standard per il widget `*w`.
  - `Widget` rende disponibile almeno un opportuno costruttore per impostare le caratteristiche dei widget.
2. Definire una classe `AbstractButton` derivata da `Widget` i cui oggetti rappresentano un generico componente pulsante. Ogni oggetto `AbstractButton` è caratterizzato dalla stringa che etichetta il pulsante.
  - `AbstractButton` rende disponibile almeno un opportuno costruttore per impostare le caratteristiche dei pulsanti.
3. Definire una classe `PushButton` derivata da `AbstractButton` i cui oggetti rappresentano un pulsante clickabile.
  - `PushButton` implementa il metodo virtuale puro `setStandardSize()` come segue: per ogni puntatore `p` a `PushButton`, `p->setStandardSize()` imposta la dimensione standard 80×20 per il pulsante clickabile `*p`.
  - `PushButton` rende disponibile almeno un opportuno costruttore per impostare le caratteristiche dei pulsanti clickabili.
4. Definire una classe `CheckBox` derivata da `AbstractButton` i cui oggetti rappresentano un pulsante checkabile. Ogni oggetto `CheckBox` è caratterizzato dall'essere nello stato “checked” o “unchecked”; inoltre, tutti gli oggetti `CheckBox` sono sempre visibili.
  - `CheckBox` implementa il metodo virtuale puro `setStandardSize()` come segue: per ogni puntatore `p` a `CheckBox`, `p->setStandardSize()` imposta la dimensione standard 5×5 per il pulsante checkabile `*p`.
  - `CheckBox` rende disponibile almeno un opportuno costruttore per impostare le caratteristiche dei pulsanti checkabili.

Il *dispatching* si riferisce solo all'azione di trovare la funzione giusta da chiamare. Nel caso generale, quando si definisce un metodo all'interno di una classe

In [computer science](#), **dynamic dispatch** is the process of selecting which implementation of a [polymorphic](#) operation ([method](#) or function) to call at [run time](#). It is commonly employed in, and considered a prime characteristic of, [object-oriented programming](#) (OOP) languages and systems.<sup>[1]</sup>

## Late binding in C++ [\[ edit \]](#)

In C++, late binding (also called "dynamic binding") refers to what normally happens when the `virtual` keyword is used in a method's declaration. C++ then creates a so-called virtual table, which is a look-up table for such functions that will always be consulted when they are called.<sup>[7]</sup> Usually, the "late binding" term is used in favor of "dynamic dispatch".



# Late binding/Dynamic dispatch



```
1  #include <iostream>
2
3  class B
4  {
5  public:
6      virtual void bar();
7      virtual void qux();
8  };
9
10 void B::bar()
11 {
12     std::cout << "B::bar" << std::endl;
13 }
14
15 void B::qux()
16 {
17     std::cout << "B::qux" << std::endl;
18 }
```

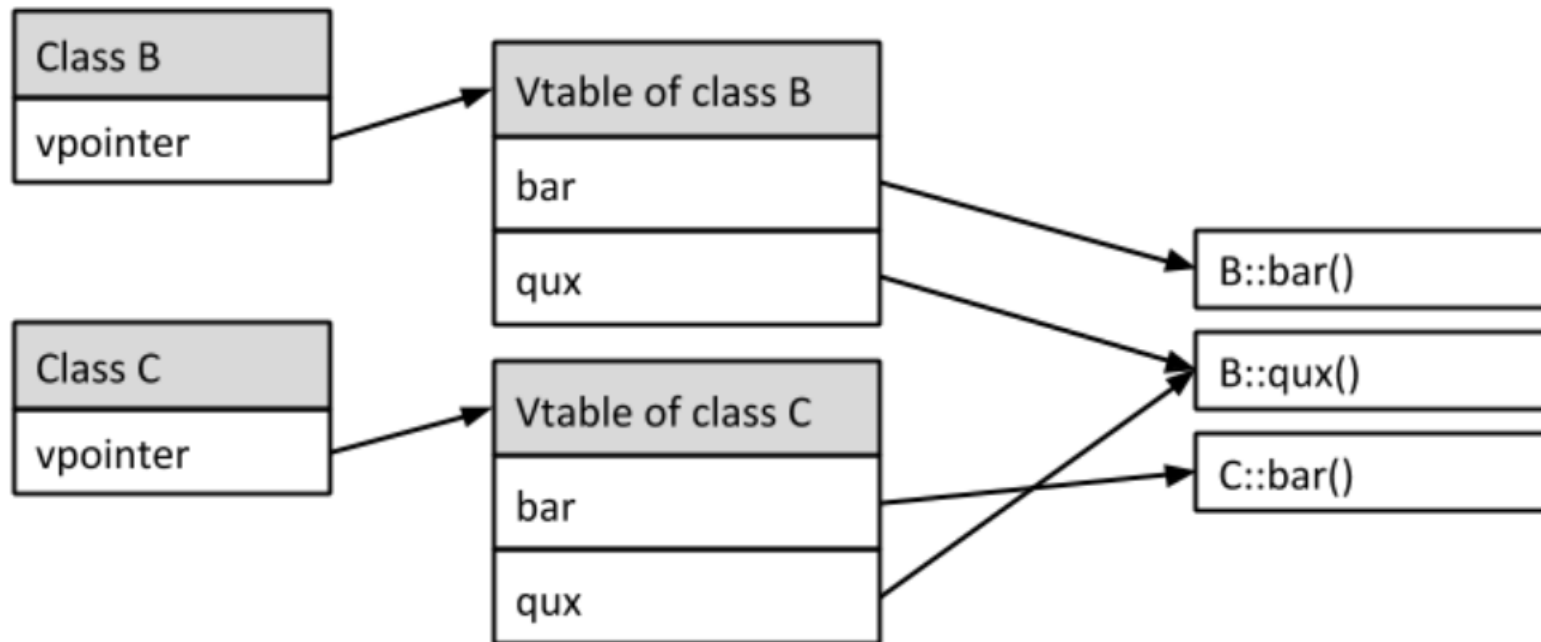
```
1  class C : public B
2  {
3  public:
4      virtual void bar();
5  };
6
7  void C::bar()
8  {
9      std::cout << "C::bar" << std::endl;
10 }
```



# Late binding/Dynamic dispatch




UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA





Avendo D classe derivata da B, consideriamo il seguente caso:

- D sottooggetto di B



```
1  D* pd = new D;  
2  B* pb = pd;  
3  delete pb;
```

Se il tipo statico dell'oggetto da cancellare è diverso dal suo tipo dinamico, il tipo statico deve essere una classe base del tipo dinamico dell'oggetto da cancellare e il tipo statico deve avere un distruttore virtuale o il comportamento è *indefinito*.  
Questo porta a possibili leak di memoria e risorse.

# Distruttore virtuale



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

```
1  class B{
2      ....
3      virtual ~B();
4  }
5
6  class C: public B{
7      ....
8      virtual ~C();
9  }
10
11 // Grazie ai distruttori virtuali,
12 // quando si distrugge un oggetto di tipo B*
13 // che punta ad un oggetto di tipo C,
14 // viene chiamato il distruttore di C
```



- Una classe astratta è una classe per la quale uno o più metodi sono dichiarati ma non definiti, il che significa che il compilatore sa che questi metodi fanno parte della classe, ma non quale codice eseguire per quel metodo.
  - Questi sono chiamati metodi astratti (o virtuali puri)

```
1  class shape {  
2  public:  
3      virtual void draw() = 0;  
4  };
```

# Classi astratte e classi concrete



Per poter utilizzare effettivamente il metodo *draw*, è necessario derivare da questa classe astratta delle classi che implementino il metodo *draw*, rendendo le classi concrete:

```
1  class circle : public shape {
2  public:
3      circle(int x, int y, int radius) {
4          /* set up the circle */
5      }
6      virtual draw() {
7          /* do stuff to draw the circle */
8      }
9  };
10
11 class rectangle : public shape {
12 public:
13     rectangle(int min_x, int min_y, int max_x, int max_y) {
14         /* set up rectangle */
15     }
16     virtual draw() {
17         /* do stuff to draw the rectangle */
18     }
19 };
```



# Classi astratte e classi concrete



```
1  #include <iostream>
2  using namespace std;
3
4  class B{    //classe base astratta
5      public:
6          virtual void f() = 0;
7  };
8
9  class C: public B{
10     public:
11         void G() {cout << "C::G"
12     };
13 }
```

```
1  class D: public B{
2      public:
3          virtual void f() {cout << "D::f" << endl;}
4  };
5
6  int main(){
7      // C c; //Illegale: C è astratta
8      D d; // Legale: D è concreta
9      B* p = &d; // Legale: p punta a un oggetto concreto
10     p = &d; // puntatore superpolimorfo
11     // cioè può puntare a oggetti di tipo diverso a runtime
12     p->f(); // Legale: D::f
13 }
```

