Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Inheritance

"In the one and only true way, the object-oriented version of 'Spaghetti code' is, of course, 'Lasagna code'. (too many layers)."
- Roberto Waltman.

Why inheritance ?

- For software re-use: re-use an existing class in new classes that are specializations of the base class
- A new class is derived from the base class and it inherits the facilities of the base class
  - A derived class may itself be the basis of further inheritance -forms a class hierarchy
  - The derived class extends the functionalities of the base class
- Inheritance is the second most important concept in object-oriented programming - the first is abstract data type

Why inheritance ? - cont.

- Inheritance allows us to avoid duplication of code or functions by getting all the features of another class simply by naming it in an inheritance.
    - Then, if the private data or the coding needed to implement any of the common features needs to be changed, it is changed only in the base class and not in the derived classes that obtain the changes automatically

When to use inheritance

- Use inheritance as a specification device.
- “Human beings abstract things on two dimensions: part-of and kind-of. A Ford Taurus is-a-kind-of-a Car, and a Ford Taurus has-a Engine, Tires, etc. The part-of hierarchy has been a part of software since the ADT style became relevant; inheritance adds "the other" major dimension of decomposition.”

From: C++ FAQ Lite - [19.2]

Using inheritance

- The design of class hierarchies is a key skill in object oriented design.
- Only use inheritance when there is a clear "is a" relationship between the derived classes and the base class
- Inheritance expresses the natural relationship that, for example, "a bus is a vehicle."
- An instance of a derived class could substitute an instance of a base class (derived is_a base)

Inheritance in C++ - example

```
class Person {
public:
const string& getName() const;
// ...
};


class Student : public Person {
// ...
};
class Staff : public Person {
// ...
};

class Permanent : public Staff {
// ...
};
class Casual : public Staff {
// ...
};
```

- After each derived class name there is a colon “`:`” followed by the keyword “`public`” and then the name of the class from which it is inheriting. The colon represents inheritance.

- The keyword `public` after the colon says that we are using public inheritance. This is the most common form of inheritance although it is possible to have `protected` and `private` inheritance.

- These different kinds of inheritance relate to whether the public members of the base class will or will not be accessible to the users of the derived class. With public inheritance the public members of the base class effectively become public members of the derived class.

Inheritance access specifiers

```
class B {
public:
  void pub();
protected:
  void prot();
private:
  void priv();
};
```

```
class D : public B {};
class D : protected B {};
class D : private B {};
```

		Base class member access		
		public	protected	private
Derived class inheritance access	public	whatever function	methods of D friends of D classes derived from D	not accessible
	protected	methods of D friends of D classes derived from D	methods of D friends of D classes derived from D	not accessible
	private	methods of D friends of D	methods of D friends of D	not accessible

Inheritance access specifiers

- a `public` derived class inherits the `public` and `protected` members of the base maintaining their access level
- a `protected` derived class inherits the `public` and `protected` members of the base but expose them as `protected`
- a `private` derived class expose the `public` and `protected` members of the base as `private`

Class interface

- A class has two distinct interfaces for two distinct sets of clients:
  - It has a `public` interface that serves unrelated classes
  - It has a `protected` interface that serves derived classes

Access control

- The private members of a class remain just private! A derived class CAN NOT access the private members of the base class, even though they do inherit them (they are included in an object of the derived class)
- Private members are only accessible via the public methods of the base class. They cannot be accessed directly by users of the derived class nor can they be accessed directly by the methods of the derived class.

Protected Members

- What if we want a class member to be visible to the methods of a derived class but not to be visible to users of either the base class or the derived class?
  - C++ `protected` members.
- If there are levels of indirect inheritance through a class hierarchy, protected members will be accessible throughout the class hierarchy.

Protected Members - cont.

```
class baseClass{
public:
  void method1();
protected:
  void method2();
};

class derivedClass :
public baseClass {
public:
  void method3() {
    method2(); // OK
  };
};
```

```
derivedClass d;

d.method1(); // OK

d.method3(); // OK

d.method2(); // ERROR!
method2 is protected
```

Access control hint

- Declare your base class's data members as private and use `protected` inline access functions by which derived classes will access the private data in the base class. This way the private data declarations can change, but the derived class's code won't break (unless you change the protected access functions)

From: C++ FAQ Lite - [19.7]

Accessing Base Class Members

- An object of a derived class inherits the members of the base class, eg.

-
  ```
  Casual cas;
  std::cout << "Name: " << cas.getName() << endl;
  ```
- Real power of inheritance is when we don't know the actual type of an object, e.g.

  -
    ```
    Person *p;
    p = findPerson(...);
    std::cout << "Name: " << p->getName() <<
    std::endl;
    ```
- This is an example of polymorphism.

Accessing Base Class Members

Where is it implemented ? Looks like implemented in Casual class, but could be also in base class

f the base class, eg.

-
  ```
  Casual cas;
  std::cout << "Name: " << cas.getName() << endl;
  ```

- Real power of inheritance is when we don't know the actual type of an object, e.g.

  -
    ```
    Person *p;
    p = findPerson(...);
    std::cout << "Name: " << p->getName() <<
    std::endl;
    ```

- This is an example of polymorphism.

Accessing Base Class Members

Where is it implemented ? Looks like impleme but could be

- casual cas std::cout
- Real power of object, e.g.

p is a pointer to the base class findPerson may return derived classes, but we can invoke methods of the base class without knowing what p has become.

There's need of a bit of work... see it in a few slides

-
  ```
  Person *p;
  p = findPerson(...);
  std::cout << "Name: " << p->getName() <<
  std::endl;
  ```

- This is an example of polymorphism.

Inheritance vs. Composition

- Why not this?

```
class Student {
public:
  Person details;
  // ...
};
```

- This is composition. It is used when objects of one class contain or comprise one or more objects of another class:

```
Student s;
cout << "Name: " << s.details.getName();
```

Inheritance vs. Composition

- Why not this?

```
class Student {
public:
  Person details;
  // ...
};
```

- This is composition. It is used when objects of one class contain or comprise one or more objects of another class:

```
Student s;
cout << "Name: " << s.details.getName();
```

Notice access using two levels of member selection

Inheritance vs. Composition - cont.

- Use inheritance for "is_a" relationships, composition for “has_a” or "contains" or "is_comprised_of" relationships.
- Consider the case of multiple instances of a class within another class, e.g.

```
class Person {
public:
  Address home;
  Address office;
  // ...
};
```

- Can't do this with inheritance!

Composition and relationships

- When an object contains another object there could be a relation that is different from *has_a* and is more like *is_implemented_in_terms_of*, e.g. when one class heavily relies on the behaviour of a contained class, modifying some of its features

Using derived classes

It is possible to use an object instantiated from a derived class whenever it is possible to use an object instantiated from the base class (because derived obj *is_a* base obj):

```
class Employee {
  string first_name, family_name;
  Date hiring_date;
  short department;
  // ...
};
class Manager : public Employee {
  set<Employee*> group;
  short level;
  // ...
};
```

```
void paySalary(Employee* e)
{
 //... code to pay salary
}
//...
Employee *e1;
Manager *m1;
//...
paySalary(e1);
paySalary(m1);
```

Public inheritance and is_a

- If D extends publicly B then D *is_a* B and any function that expects a B (or pointer to B or reference to B) will also take D (or pointer to D or reference to D)

```
class Person {...};
class Student : public Person {...};
void eat(const Person& p);
void study(const Student& s);
Person p;
Student s;
eat(p); // OK
eat(s); //OK: s is_a p
study(s); // OK
study(p); // bad: p is not an s
```

Liskov substitution principle

- Substitutability is a principle in object-oriented programming stating that if S is a subtype of T, then objects of type T may be replaced with objects of type S
- i.e. an object of type T may be substituted with any object of a subtype S without altering any of the desirable properties of the program (correctness, task performed, etc.).

Liskov substitution principle

- Substitutability is a principle in object-oriented programming stating that if S is a subtype of T, then objects of type T may be replaced with objects of type S

Original definition by Barbara Liskov (1994):

Subtype Requirement: Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S is a subtype of T.

Liskov substitution principle

- Substitutability is a principle in object-oriented

In other words, if you make a class that extends a base class, it should not alter its parents behavior significantly.

Original definition by Barbara Liskov (1994):

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Liskov substitution principle

- The Liskov substitution principle (LSP), identifies one particular mathematical notion of subtype with the programming notion of class extension such as C++ class derivation.
- It's a principle of generally good class design for polymorphism.
- When a class `Derived` extends a class `Base`, then with respect to the properties guaranteed by the `Base` specification, code that refers to a `Base` object should work just as well if that object is actually a `Derived` object.

Public inheritance and is_a - cont.

- But be careful with design:

```
class Bird {
public:
 virtual void fly();
...
};

class Penguin : public
Bird { ... };

Penguin p;
p.fly(); // but penguins
         // do not fly !
```

- Perhaps it’s better to have:

```
class Bird { ... };

class FlyingBird : public
Bird {
public:
 virtual void fly();
}

class Penguin : public
Bird { ... };
```

Public inheritance and is_a - cont.

- Public inheritance asserts that everything that applies to base object applies to derived object
  - it's up to you to design correctly the base class, so that penguins do not fly!

Private inheritance

- The behaviour is quite different when inheriting privately: we do not have anymore a *is_a* relation, the compiler will not convert the derived class to base:

```
class Student : private Person { ... };

void eat(const Person& p);

Student s;
eat(s); // error: now a Student is not a Person !
```

Private inheritance - cont.

- All that is inherited becomes private: it's an implementation detail
- Private inheritance means that the derived class D *is_implemented_in_terms_of* the base class B, not that D *is_a* B
- Use private inheritance if you want to inherit the implementation of the base class, use public inheritance to get also the interface

Private inheritance - cont.

- Remind that also composition let to implement a class in terms of another (composed) class
- Use composition whenever you can and private inheritance when you need, e.g. when you need to access protected parts of a class or redefine virtual methods (more on this later)

Constructors and inheritance

- When an object of a derived class is created, the constructors (if any) of each inherited class are invoked in sequence prior to the final class constructor (if any). It’s a bottom-up process.
- Default constructors are invoked automatically.
- If a base class does not have a default constructor, any other constructor must be invoked explicitly by the derived class's constructor in its initialisation list.

Constructors and inheritance - cont.

```
class Base {
public:
  Base();
  Base(int a, int b, int c);
};
```

```
class Derived: public Base {
private:
  int d;
public:
  Derived();
  Derived(int a, int b, int c, int d);
  void print();
};

Derived::Derived() { d=0;}
Derived::Derived(int a=0, int b=0, int c=0, int d=0) :
  Base(a,b,c) // Use a,b,c as parameters to the c’tor of Base
  {  this->d = d; }
Derived::Derived(int a=0, int b=0, int c=0, int d=0) :
  Base(a,b,c) , d(d) {}
```

Destructors and Inheritance

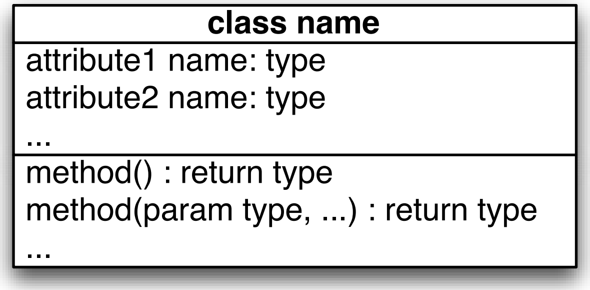
- Just like constructors, except the order is reversed! It’s a top-down process.
- When a derived class is destroyed, the derived class destructor (if any) will be invoked first and then the base class destructor (if any) will be invoked.
- Destructors are not overloaded or invoked explicitly so we don't have the confusion over which destructor is invoked!

Multiple inheritance

- A class may derive from several base classes
- Just report all the base classes after the ":", and state the access level, e.g.:

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
class Tsec : public Temporary, public Secretary
  { /* ... */ };
class Consultant : public Temporary, public Manager
  { /* ... */ };
```

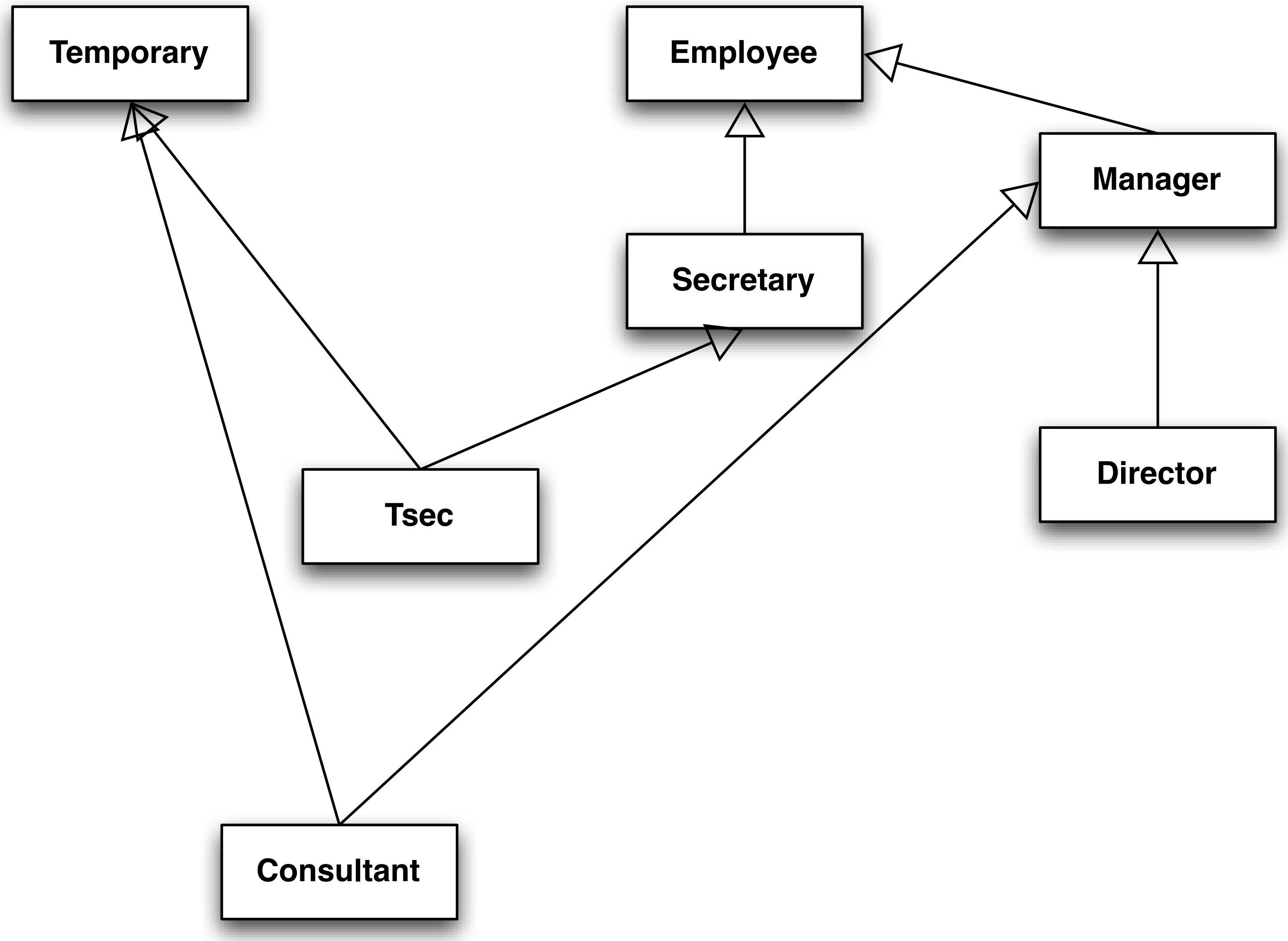
A bit of UML class diagram



class name

attribute1 name: type
attribute2 name: type
...

method() : return type
method(param type, ...) : return type
...



Temporary

Employee

Manager

Secretary

Director

Tsec

Consultant

Polymorphism

Polymorphism

- In programming languages and type theory, polymorphism (i.e."many forms", from Greek) is the provision of a **single interface** to entities of **different types**.
- There are several different kinds of polymorphism. C++ caters for all them using:
  - overloading
  - generic programming (templates)
  - subtyping: a name may denote instances of different classes if they share a common base class.

Polymorphism

- In programming languages and type theory, polymorphism (i.e. “many forms”, from Greek) is the provision of a **single interface** to entities of **different types**.
- Th
  C
  - overloading
  - generic programming (templates)
  - subtyping: a name may denote instances of different classes if they share a common base class.

In object-oriented programming, this is often referred to simply as polymorphism

Polymorphism

- The main concept behind subtype polymorphism is:

  we can **refer to an instance of a subclass** as if it were an **instance of its superclass**...

  ...but each object responds to method calls as specified by **its actual type**

Polymorphism

- A derived class can override a method inherited from a base class
  - the class should simply include a declaration of the method (and provide an implementation)
  - the overridden method often adds some behaviour according to the specialization of the derived class (may upcall the base method)
- The method is polymorphic because it has a different implementation depending if it's invoked on the base or the derived class

Override vs. overload

- Overloaded method: same method name but different parameters (in the same class)
- Overridden method: same name and parameters in a class hierarchy

Override and overload

- Overloaded methods in a base class can be overridden in a derived class
- A derived class can overload an overridden method, adding a new behavior to its interface

Override and overload

```
// OK
class B {
public:
     /* ... */
     virtual void foo() {
          std::cout << "B::foo()" << std::endl;
     }
     void foo(int i) { // overloaded
          std::cout << "B::foo(int) : " << i << std::endl;
     }
};
class D : public virtual B {
public:
     /* ... */
     virtual void foo() { // overridden
          std::cout << "D::foo()" << std::endl;
     }
     void foo(int i)  { // overloaded overridden
          std::cout << "D::foo(int) : " << i << std::endl;
     }
     void foo(float j)  { // overloaded
          std::cout << "D::foo(float) : " << j << std::endl;
     }
};
```

Override and overload

```
// OK
class B {
public:
	/* ... */
	virtual void foo() {
		std::cout << "B::foo()" << std::endl;
	}
	void foo(int i) { // overloaded
		std::cout << "B::foo(int) : " << i << std::endl;
	}
};
class D : public virtual B {
public:
	/* ... */
	virtual void foo() { // overridden
		std::cout << "D::foo()" << std::endl;
	}
	void foo(int i)  { // overloaded overridden
		std::cout << "D::foo(int) : " << i << std::endl;
	}
	void foo(float j)  { // overloaded
		std::cout << "D::foo(float) : " << j << std::endl;
	}
};
```

```
B* pb;
pb = new D;
pb->foo();
pb->foo(42);
D ad;
ad.foo();
ad.foo(42);
ad.foo(float(3.14));
```

Late binding

- The override feature lets different implementations of a method to exist: this introduces a problem of binding the invocation of a method to a particular implementation:
  - the decision is based on the type of the class used to refer to a method:
    - <var>.op() uses the op() of the class of <var>
    - <addr_expr>->op() uses the op() of the class of <addr_expr> that may be different from the class of the instantiated object

Late binding - example

```
class Base {
public:
    Base();
    ~Base();
    void foo() {
      std::cout << "Base::foo" << std::endl;
    };
    int foo2() {
      std::cout << "Base::foo2" << std::endl;
      return -1;
    }
};

class Derived1: public Base {
public:
    Derived1();
    ~Derived1();
    void foo() {
      Base::foo(); // upcall
      std::cout << "Derived1::foo" << std::endl;
    }
    int foo2() {
      std::cout << "Derived1::foo2()" << std::endl;
      return 1;
    }
};
```

```
Base *pBase;
Derived1 aD1;

cout << "pBase = &D1" << endl;
pBase = &aD1;  // Base pointer to derived class
pBase->foo();  // Base::foo() because of
               // static bind

cout << “D1.foo()" << endl;
aD1.foo();  // Derived::foo()

// cast to call the method of derived class
((Derived *)pBase)->foo2(); // Derived::foo2()
```

Virtual methods

- Virtual methods avoid the need for a client of a class to know the concrete type of the instance it is using
  - in the previous example we had to cast a base pointer to use a method overridden in the derived class
- One or more methods of a derived class can be declared as `virtual` adding the keyword in their declaration

Virtual methods - cont.

- A `virtual` method in the base class remains virtual in the derived classes (even if the `virtual` declaration is not expressly used)
- The `virtual` declaration modifies the binding: the implementation that is used is always that of the instantiated class

Virtual methods - example

```
class Base {
public:
     Base();
     virtual ~Base();
     void foo() {
        std::cout << "Base::foo" << std::endl;
     };
     virtual int bar(int i) {
         std::cout << "Base::bar" << i <<
               std::endl;
          return (i);
     }
};

class Derived1: public Base {
public:
    Derived1();
    virtual ~Derived1();
    void foo() {
      Base::foo(); // upcall
      std::cout << "Derived1::foo" << std::endl;
    };
    virtual int bar(int i) {
      std::cout << "Derived1::bar" << i <<
          std::endl;
      return (i+1);
    };
};
```

```
Base *pBase;
Derived1 aD1;

cout << "pBase = &D1" << endl;
pBase = &aD1;  // Base pointer to derived class
pBase->foo();  // Base::foo()

cout << "D1::foo()" << endl;
aD1.foo();  // Derived1::foo()

// NO need to cast the pointer: it’s a virtual
method
pBase->bar(1); // Derived1::bar()
```

Why virtual methods ?

- The use of virtual methods greatly reduces the coupling of a client and a hierarchy of classes developed from a base class
  - a pointer of base class type does not require to know what type it is pointing at: the late (dynamic) binding will solve the problem !
- Virtual methods are the key facility to polymorphism: the function that is invoked using a base class pointer (or reference) can have many form, depending upon the actual type of object that is being used.

Rules for Virtual Functions

- A virtual function must be marked virtual in the base class.
- A function in a derived class with the same signature as a virtual function in the base class will be virtual even if not marked virtual. Always mark it anyway.
- A separate definition (i.e. not within the class declaration) of a virtual function is not marked virtual.
- Top level functions cannot be virtual. It would not make any sense...
- Class functions (marked static) cannot be virtual. It would not make any sense...

`override` and `final` (C++11)

- The new C++11 standard introduces two specifiers for virtual functions:
    - `override`: indicates that a method in a derived class intends to be an override of a virtual method in the base class
    - `final`: indicates that a method in a base class can not be overridden in a base class

`override` and `final` (C++11)

- The new C++11 standard introduces two specifiers for virtual functions:
  - `override`: indicates that a method in a derived class intends to be an override of a virtual method in the base class
  - `final`: indicates that a method in a base class can not be overridden in a base class

Remember to tell the compiler to use the new standard

override (C++11)

```
class B {
public:
  virtual void f1(int)
    const;
  virtual void f2();
  void f3();
};
```

```
class D1 : public B {
public:
  void f1(int) const override;
  // ok: f1 matches f1 in the base
  void f2(int) override;
  // error: B has no f2(int)
  void f3() override;
  // error: f3 not virtual
  void f4() override;
  // error: B doesn't have a
  // function named f4
};
```

final (C++11)

```
class B {
public:
  virtual void f1(int)
    const;
  virtual void f2();
  void f3();
};

class D2 : public B {
public:
// inherits f2() and
// f3() from B and
// overrides f1(int)
  void f1(int) const
    final;
  // subsequent classes
  // can't override f1 (int)
};
```

```
class D3 : public D2 {
public:
  void f2();
  // ok: overrides f2
  // inherited from the
  // indirect base, B
  void f1(int) const;
  // error: D2 declared f1
  // as final

};
```

final (C++11)

- final can also block the possibility to derive from a class, e.g.:

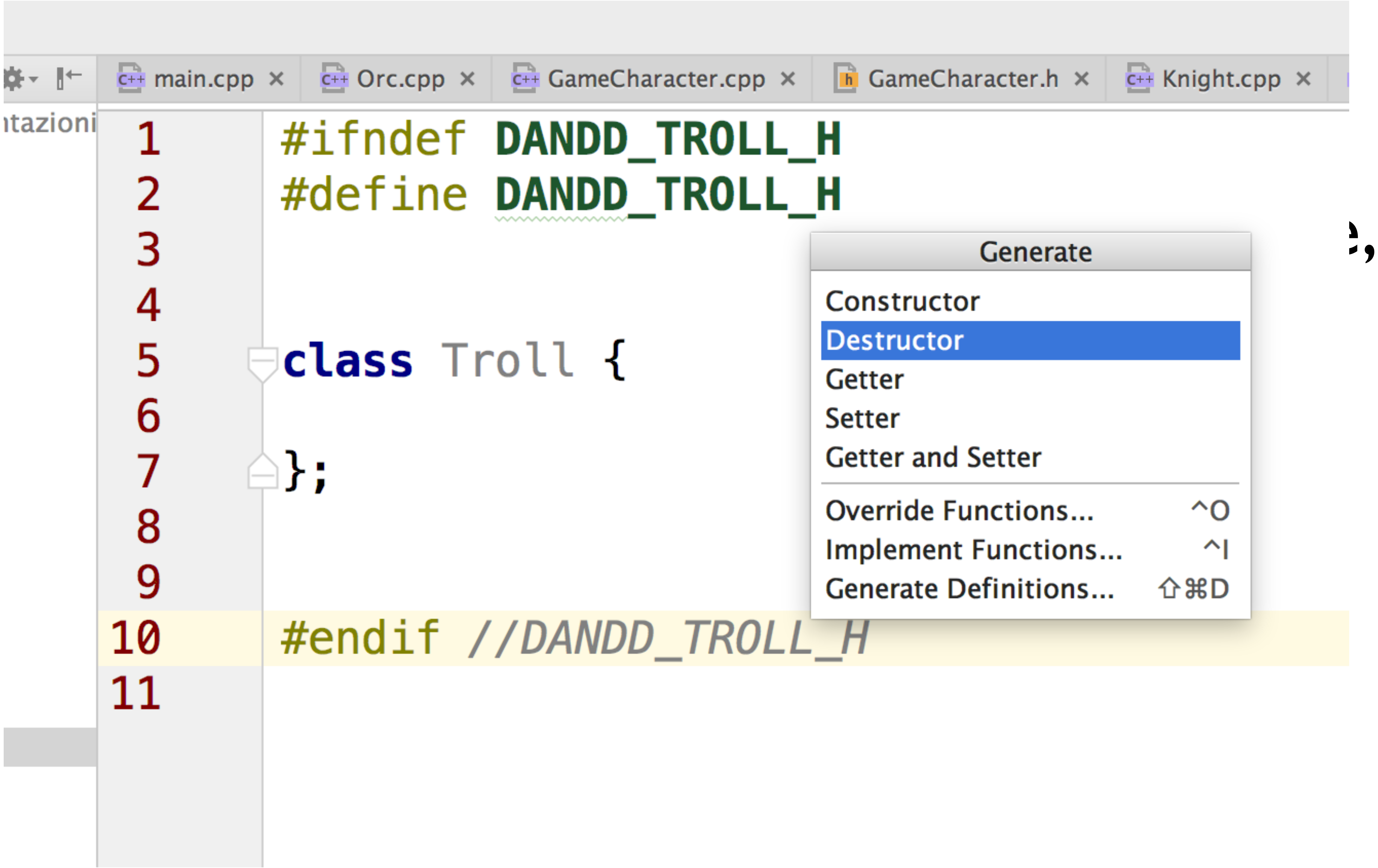
```
class SuperCar final : public Car
{
//
};
```

- ... it’s not possible to derive from SuperCar.

Constructors and Destructors

- Constructors cannot be virtual: a constructor is invoked on an explicit type, there is no need for polymorphism to be considered
- Destructors can be virtual. Making them virtual ensures that the correct ones are called if the object is identified by a base class reference or pointer.
  - Notice that the CLion and Eclipse class wizards always create virtual destructors !

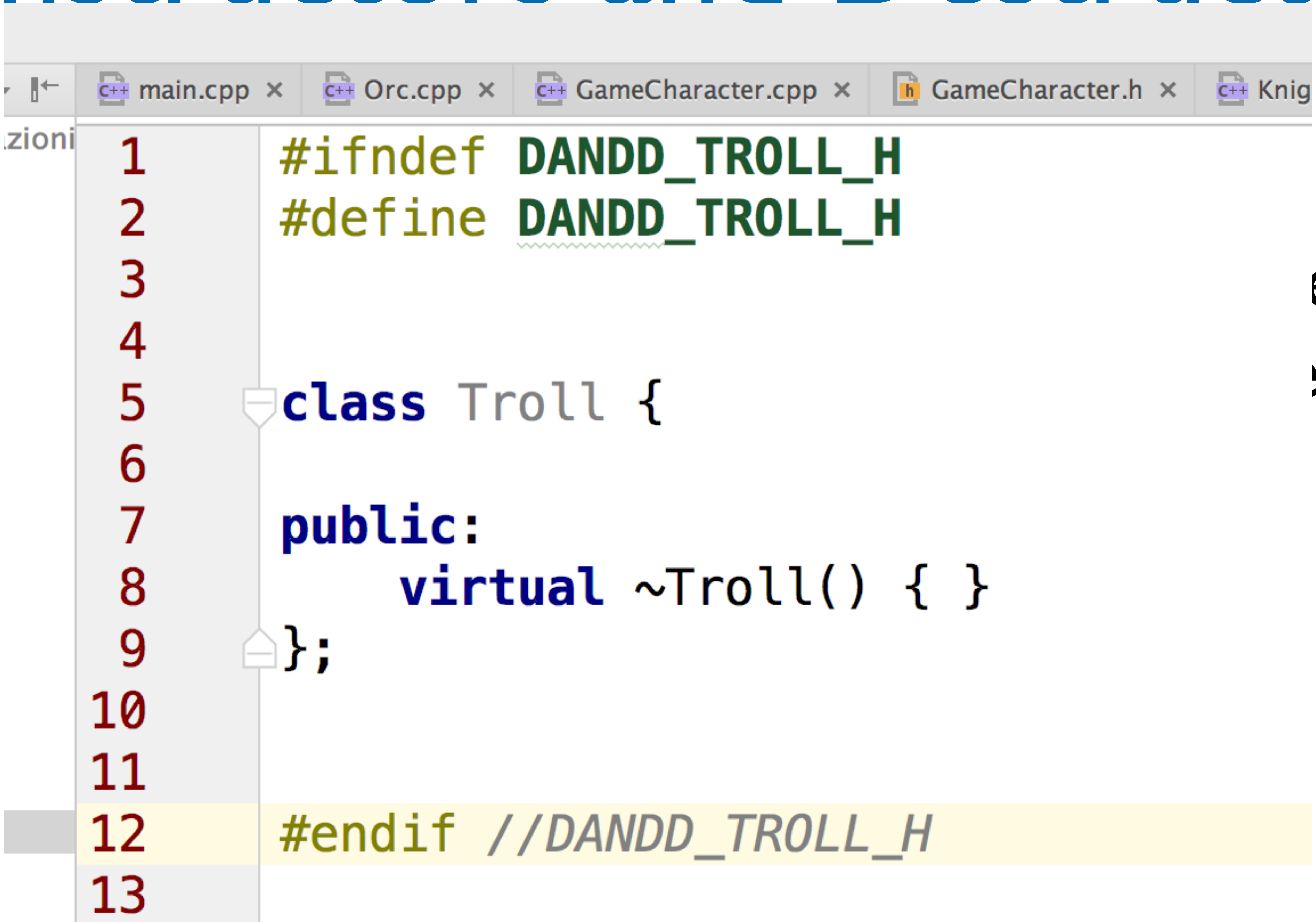
Constructors and Destructors



```
#ifndef DANDD_TROLL_H
#define DANDD_TROLL_H


class Troll {

};


#endif //DANDD_TROLL_H

```

- Notice that the CLion and Eclipse class wizards always create virtual destructors !

Constructors and Destructors



```
#ifndef DANDD_TROLL_H
#define DANDD_TROLL_H


class Troll {

public:
    virtual ~Troll() { }
};


#endif //DANDD_TROLL_H

```

- Notice that the CLion and Eclipse class wizards always create virtual destructors !

Virtual destructors

- Remind to declare virtual destructors in polymorphic base classes (i.e. those who have at least one virtual method)

-
  ```
  class TimeKeeper {
  public:
    TimeKeeper();
    ~TimeKeeper();
    virtual getCurrentTime();
    ..
  };

  class AtomicTimeKeeper :
  public TimeKeeper {...};

  class WristWatch : public
  TimeKeeper {...};
  ```

-
  ```
  // this function allocates
  // a watch on the heap with
  // a new
  TimeKeeper* getTimeKeeper();
  ...
  TimeKeeper* ptk =
    getTimeKeeper(); // get it
  ... // use it
  delete ptk; // release it
  ```

Virtual destructors

- Remind to declare virtual destructors in polymorphic base classes (i.e. those who have at least one virtual method)

-
```
class TimeKeeper {
public:
  TimeKeeper();
  ~TimeKeeper();
  virtual getCurrentTime();
```

The derived part of the object will not be released leaking resources

```
                                     .c
TimeKeeper {...};
```

-
```
// this function allocates
// a watch on the heap with
// a new
TimeKeeper* getTimeKeeper();
...
TimeKeeper* ptk =
  getTimeKeeper(); // get it
... // use it
delete ptk; // release it
```

Virtual destructors

- Remind to declare virtual destructors in polymorphic base classes (i.e. those who have at least one virtual method)

-
  ```
  class TimeKeeper {
  public:
    TimeKeeper();
    ~TimeKeeper();
    virtual getCurrentTime();
  ..
                                  .c
  TimeKeeper {...};
  ```

Solve the issue declaring a virtual destructor

-
  ```
  //                               s
  // a watch on the heap with
  // a new
  TimeKeeper* getTimeKeeper();
  ...
  TimeKeeper* ptk =
    getTimeKeeper(); // get it
  ... // use it
  delete ptk; // release it
  ```

The derived part of the object will not be released leaking resources

Virtual destructors - cont.

- Guideline: if a class does not contain a virtual method then probably it is not meant to be a base class (or it's a base class not to be used polymorphically)
- Guideline: it is not useful to declare a virtual destructor if there is no other virtual method in the class:
  - we waste memory for the creation of the virtual table used to manage virtual functions

Virtual destructors - cont.

- What happens if you derive from a class with no virtual destructor ?

```
class SpecialString : public std::string { // std::string
...                                        // has no virtual
};                                         // destructor

SpecialString* pss = new SpecialString(“Problems are
coming”);

std::string* ps;
...
ps = pss; // SpecialString is_a std::string
...
delete ps; // Ouch! We use the std::string destructor, any
           // resource managed by SpecialString is leaked
```

Factory

- A way to further exploit polymorphism achieved using virtual methods is the use of a factory class (covered later in the course) that instantiate objects depending on some conditions, e.g.:

```
class Factory {
public:
  Base* getInstance();
  ...
}
Base* Factory::getInstance() {
  if (...)
    return new Base;
  else
    return new Derived;
}
```

```
int main() {
  Base* pBase;
  Factory *pFactory;
  ...
  pBase = pFactory->getInstance();
  ...
  pBase->aVirtualMethod();
  ...
}
```

Covariant return type

- In object-oriented programming, a covariant return type of a method is one that can be replaced by a "narrower" type when the method is overridden in a subclass.
  - C++ allows it, Java allows it partially, C# doesn’t allow it
- Covariant (wide to narrower) return type refers to a situation where the return type of the overriding method is changed to a type related to (but different from) the return type of the original overridden method.
  - The relationship between the two covariant return types is usually one which allows substitution of the one type with the other, following the Liskov substitution principle.

Covariant return type

- An overridden method in a derived class can return a type derived from the type returned by the base-class method.

```
class A {};
class B : public A {};
class C : public B {};

class X {
  public:
    virtual B *m1()
      { return new B();}
};
class Y : public X {
  public:
    virtual C *m1()
      { return new C();}
};
```

```
int main() {
    X x; Y y;
    x.m1();
    y.m1();
}
```

Covariant return type

`Y::m1()` can not return a `B` object if `X::m1()` returns a `C` object

```
class A {};
class B : public A {};
class C : public B {};

class X {
  public:
    virtual B *m1()
      { return new B();}
};
class Y : public X {
  public:
    virtual C *m1()
      { return new C();}
};
```

```
int main() {
    X x; Y y;
    x.m1();
    y.m1();
}
```

Covariant return type

- Covariant return type is useful to implement some designs, like allowing a class to clone objects:

```
class Base {
public:
  virtual Base* clone() const;
};
Base* Base::clone() const {
  return new Base( *this );
}

class Derived : public Base {
public:
  virtual Derived* clone() const;
};

Derived* Derived::clone() const {
  return new Derived( *this );
}
```

```
Base* b1 = new Base;
base* b2 = b1->clone();
// b2 gets a clone of b1

Derived *d1 = new Derived;
Derived *d2 = d1->clone();
// d2 gets a clone of d1
```

Name hiding

- If a base class declares a member function and a derived class declares a member function with the same name but different parameter types and/or const*ness*, then the base method is “hidden” rather than “overloaded” or “overridden” (even if the method is virtual)

Name hiding - example

```
class Base {
 public:
   void f(double x); // doesn't matter whether or not this is virtual
 };

 class Derived : public Base {
 public:
   void f(char c);  // doesn't matter whether or not this is virtual
 };

 int main() {
   Derived* d = new Derived;
   Base* b = d;
   b->f(65.3); // okay: passes 65.3 to f(double x)
   d->f(65.3); // bizarre: converts 65.3 to a char ('A' if ASCII)
               // and passes it to f(char c); does NOT call f(double x)!!
   delete d;
   return 0;
}
```

Name hiding - example

```
class Base {
 public:
   void f(dc
 };

 class Deriv
 public:
   void f(ch
 };

 int main()
   Derived*
   Base* b =
   b->f(65.3
   d->f(65.3

   delete d;
   return 0;
}
```

Solutions:

```
class Derived : public Base {
 public:
   using Base::f; // This un-hides Base::f(double x)
   void f(char c);
 };
```

or otherwise:

```
class Derived : public Base {
 public:
   // a redefinition that simply calls Base::f(double x)
   void f(double x) { Base::f(x); }
   void f(char c);
 };
```

Name hiding - cont.

- The rationale of this behaviour is that it prevents from accidentally inheriting overloads from a distant base class when creating a new class, e.g. in a library
    - if you need those overloads use the `using` declaration seen before
    - it's something similar to name hiding of variables:

```
double x;

void someFunc() {
  int x; // hides the global variable declared before
  ...
}
```

Name hiding - cont.

- Name hiding and public inheritance do not mix well: remind that Derived object is_a Base object, but hiding names make this not to hold true !
- If you inherit publicly from a class and redefine a method perhaps you should have declared the method as virtual, when accessing derived class through a base class pointer, we may call the base class method instead of redefined one

Name hiding - cont.

```
class B {
public:
  void mf();
  ...
};

Class D: public B {
public:
  void mf(); // hides B::mf()
  ...
};
```

```
D x;
B* pB = &x;
D* pD = &x;

pD->mf(); // calls D::mf()
pB->mf(); // calls B::mf()
// should have been virtual
// to call D::mf()
```

This public inheritance does not behave like a is_a relationship: D should have inherited the implementation of B::mf() ! This name hiding is bad design !

Name hiding - cont.

```
class B {
public:
  void mf();
  ...
};

Class D: public B {
public:
  void mf(); // hides B::mf()
  ...
};
```

```
D x;
B* pB = &x;
D* pD = &x;

pD->mf(); // calls D::mf()
pB->mf(); // calls B::mf()
// should have been virtual
// to call D::mf()
```

This public inheritance does not behave like a is_a relationship: D should have inherited the implementation of B::mf() ! This name hiding is bad design !

A better design requires either to:
1. Avoid to redefine mf() in D, thus inheriting the implementation of B
or
2. Declare B::mf() as virtual and provide a new specialized version in D

In this way a D is_a B

Fragile base class

- Languages like C++ (and Java) suffer from a problem which is known as fragile base classes. Base classes are termed fragile when adding new features to a base class leads to breaking existing derived classes.
- When adding a new virtual method to a base class, existing methods with the same name in derived classes will automatically override the new method. If the semantics of the new method doesn't match the existing method in the derived class, which it almost certainly won't, then trouble ensues. This problem occurs because in C++ (and Java) the user cannot specify their intent with respect to overriding, so overriding happens silently by default.

Fragile base class

Using `override` keyword in derived class doesn’t helps as it only checks if the method in derived class has the same signature of the base class.

- When adding a new virtual method to a base class, existing methods with the same name in derived classes will automatically override the new method. If the semantics of the new method doesn't match the existing method in the derived class, which it almost certainly won't, then trouble ensues. This problem occurs because in C++ (and Java) the user cannot specify their intent with respect to overriding, so overriding happens silently by default.

Abstract classes

Why abstract classes ?

- There are many situations where the base class in a class hierarchy should be an abstract class, that is, no objects can be instantiated from it.
  - it includes special declarations of virtual methods but not their implementation
- An abstract class is a base from which defining other concrete classes
- A pure abstract class has no implementation of its methods

Why abstract classes ? - cont.

- A client may rely on the “interface” provided by an abstract class without need to know details on the classes that implement it
    - it’s a technique that decouples objects, especially when considering pure abstract classes that do NOT provide inheritance of the implementation but allow the substitution mechanism

Abstract classes: how

- An abstract base class is one that has at least one pure virtual function.
- A pure virtual function is declared using the special syntax:

```
virtual void abstractMethod() = 0;
```

- The above function does not need to be defined as it does not really exist and will never be called!
- A class derived from an abstract base class must override all of its pure virtual functions or it too will be an abstract base class.

Class Hierarchy example

```
class Vehicle {
public:
  virtual int getNumWheels() const = 0;
};

class MotorCycle: public Vehicle {
public:
  virtual int getNumWheels() const {
    return 2;
  }
};

class Car : public Vehicle {
public:
  virtual int getNumWheels() const {
    return 4;
  }
};
```

```
class Truck : public Vehicle {
public:
  Truck(int w = 10) : wheels(w) {}
  virtual int getNumWheels() const {
    return wheels;
  }

private:
  int wheels;
};
```

```
Vehicle* p = new Car();
std::cout << p->getNumWheels() <<
std::endl;
```

Back to Open-Closed Principle

- Let's review how inheritance and polymorphism help us to address the Open-Closed Principle in the problem:

- We have an application that must be able to draw circles and squares on a standard GUI. The circles and squares must be drawn in a particular order. A list of the circles and squares will be created in the appropriate order and the program must walk the list in that order and draw each circle or square. We want to be able to add new shapes.

OCP - cont.

```
class Shape {
public:
 virtual void draw() const = 0;
};

class Square : public Shape {
public:
 virtual void draw() const;
};

class Circle : public Shape {
public:
 virtual void draw() const;
};
```

```
void DrawAllShapes(Shape*
list[], int size) {
 for(int i=0;
  i < size;
  i++) {
  list[i]->draw();
 }
}
```

OCP - cont.

```
class Shape {
public:
  virtual void draw() const = 0;
};

class Square : public Shape {
public:
  virtual void draw() const;
};

class Circle : public Shape {
public:
  virtual void draw() const;
};
```

```
void DrawAllShapes(Shape*
list[], int size) {
  for(int i=0;
     i < size;
     i++) {
     list[i]->draw();
  }
}
```

We use an abstract class and virtual methods to be open to changes: new shapes have to extend the base abstract class, and `DrawAllShapes()` does not require to change.

Pure virtual destructor

- If you want to make a base class abstract but have no method that is pure virtual declare the destructor as pure virtual !
  See the trick:

-
  ```
  class AWOV { // Abstract W/O Virtuals
  public:
    virtual ~AWOV() = 0;
    ...
  };

  AWOV::~AWOV() {}  // REMIND: you HAVE to define the
                    // pure virtual destructor !
  ```

Pure virtual destructor

- If you want to make a base class abstract but have no method that is pure virtual declare the destructor as pu
See the trick:

```
●class AWOV { // Abstract
 public:
   virtual ~AWOV() = 0;
   ...
 };

 AWOV::~AWOV() {}   // REMIND: you HAVE to define the
                    // pure virtual destructor !
```

We have declared pure virtual but the compiler needs a destructor that is called when it reaches the base class. Forget it and the linker will complain.

RTTI

Run-time type identification

Why RTTI ?

- Once we have obtained a pointer to an object, it is possible to use it to invoke a polymorphic function without having to know the type of the object
  - the C++ late binding will ensure that the correct (virtual) function is called according to the actual type of object.
- But what if there are operations that are unique to a particular type ? If we have the wrong type then there is no point in invoking a function that does not exist! One possible solution to this problem is to be able to explicitly determine the type of objects pointed to at runtime.

How RTTI works

- We have a base class pointer, we can then cast it to a pointer to a specific derived class and then test to see if the cast worked or not.
  - If the actual object is of the desired type then the cast can work, if not, then the cast will fail. Such a cast is called a dynamic cast.
- We use the `dynamic_cast` to attempt to cast a pointer to a base class to point to an object of a derived class.

C++ dynamic_cast

- The `dynamic_cast` is used to check at run-time whether a cast is type safe.
- It is only legal on a polymorphic type, i.e. a class that has at at least one virtual method. More specifically:
  - The source type (in round brackets) must be a pointer or reference to a polymorphic type.
  - The target type (in angled brackets) must be a pointer or reference, but need not be polymorphic.
- We are working on pointers, therefore a failure results in a `0` pointer (always check if we got `0` as result!)

dynamic_cast example

```
class B {
public:
 virtual void f() {…}
};

class D1 : public B {
public:
 virtual void f() {…}
 void D1specific() {…}
};

class D2 : public B {
public:
...
};
```

```
B* bp;
D1* dp;
bp = new D1;
dp = dynamic_cast<D1*>(bp);
if (dp != 0) {
  dp->D1specific();
}
bp = new D2;
dp = dynamic_cast<D1*>(bp);
if (dp != 0) {
  dp->D1specific();
}
```

dynamic_cast example

```
class B {
public:
 virtual void f() {…}
};
```

More realistically: when using a Factory to get the instances, we do not know what is the real type of the object

```
};

class D2 : public B {
public:
...
};
```

```
B* bp;
D1* dp;
bp = new D1;
dp = dynamic_cast<D1*>(bp);
if (dp != 0) {
  dp->D1specific();
}
bp = new D2;
dp = dynamic_cast<D1*>(bp);
if (dp != 0) {
  dp->D1specific();
}
```

dynamic_cast to reference

- If we use dynamic_cast to reference we can not check for a 0, because a reference must always be valid
- C++ uses a different error handling mechanism we will see in a future lecture: exceptions:

```
try {
   T& tref = dynamic_cast<T&>(xref);
} catch(bad_cast) {
   // ...
}
```

typeid

- The `typeid` operator returns an identification of the type of a basic type, a class, a variable or any expression.
  May be useful to store objects to file, recording the type of each object.
- Requires `#include<typeinfo>`.
- `typeid` actually returns a reference to an object in the system class `type_info`, so compare results of `typeid()`.
- You don't need to know the internal details, e.g. to test if a variable is of a particular type:

```
if( typeid(x) == typeid(float) ) {
  // ...
}
```

Deleted methods

How ?

- Since C++11 it is possible to disallow the definition of some functions.
- Add `=delete` at the end of the declaration of a method
- Can be applied to any method of a class, like one of the methods created automatically by the compiler
- Can be applied to methods inherited from a base class

Example

```
class A {
public:
    A(int a){};
    A(double) = delete; // conversion disabled
    A& operator=(const A&) = delete;
      // assignment operator disabled
};
```

Example: inheritance

```
class A {
public:
  virtual void foo() {  }
  void bar() { }
  void foobar() { }
};

class B : public A {
public:
  virtual void foo() { }
  void bar() { }
  void foobar() = delete;
};

class C : public B {
public:
  C() {}
  virtual void foo() { }
  void bar() { }
  void foobar() {};
};
```

```
B b;
b.foo();
b.bar();
b.foobar(); // compile error:
            // it is deleted


A* pA;
pA = &b;
pA->foo();
pA->bar();
pA->foobar(); // foobar can not be
              // virtual: Calling
              // A::foobar()

C c;
c.foobar();  // OK
```

Example: inheritance

Be careful: now B is NOT anymore an A

```
class A {
public:
  virtual void foo() {  }
  void bar() { }
  void foobar() { }
};

class B : public A {
public:
  virtual void foo() { }
  void bar() { }
  void foobar() = delete;
};

class C : public B {
public:
  C() {}
  virtual void foo() { }
  void bar() { }
  void foobar() {};
};
```

```
B b;
b.foo();
b.bar();
b.foobar(); // compile error:
            // it is deleted


A* pA;
pA = &b;
pA->foo();
pA->bar();
pA->foobar(); // foobar can not be
              // virtual: Calling
              // A::foobar()


C c;
c.foobar();  // OK
```

Multiple inheritance

Multiple inheritance

- It's more complex than single inheritance: the inheritance hierarchy is no longer a strict hierarchy (tree) but becomes a network (or graph).
- There's the IS A relationship between a derived class and its base classes, e.g.:
  a tutor IS A student and
  a tutor IS A temporary employee

Multiple Inheritance Rules

- No real changes from single to multiple inheritance.
- The derived class inherits all the data members and methods from the bases classes.
- Protected members of base classes can be accessed by the derived class, as before.
- Name conflicts can result in members of the base classes have the same name (solve by appropriate using of declarations or by full qualification of the names).
- Constructors of each base class (if any) will similarly be invoked prior to the derived class constructor (if any). Destructors likewise but in the reverse order.

Multiple Inheritance characteristics

- Base c'tors are called in the order of the inheritance declared in the class declaration, e.g.:

```
class Bat : public Mammal, public Winged {
  public:
    Bat(); // the Mammal() c'tor is called
           // before Winged()
}
```

- Solve ambiguities using scope resolution, e.g.:

```
Bat aBat;
aBat.Mammal::eat();   // if both Mammal and Winged
                      // have a eat() method
```

Diamond problem

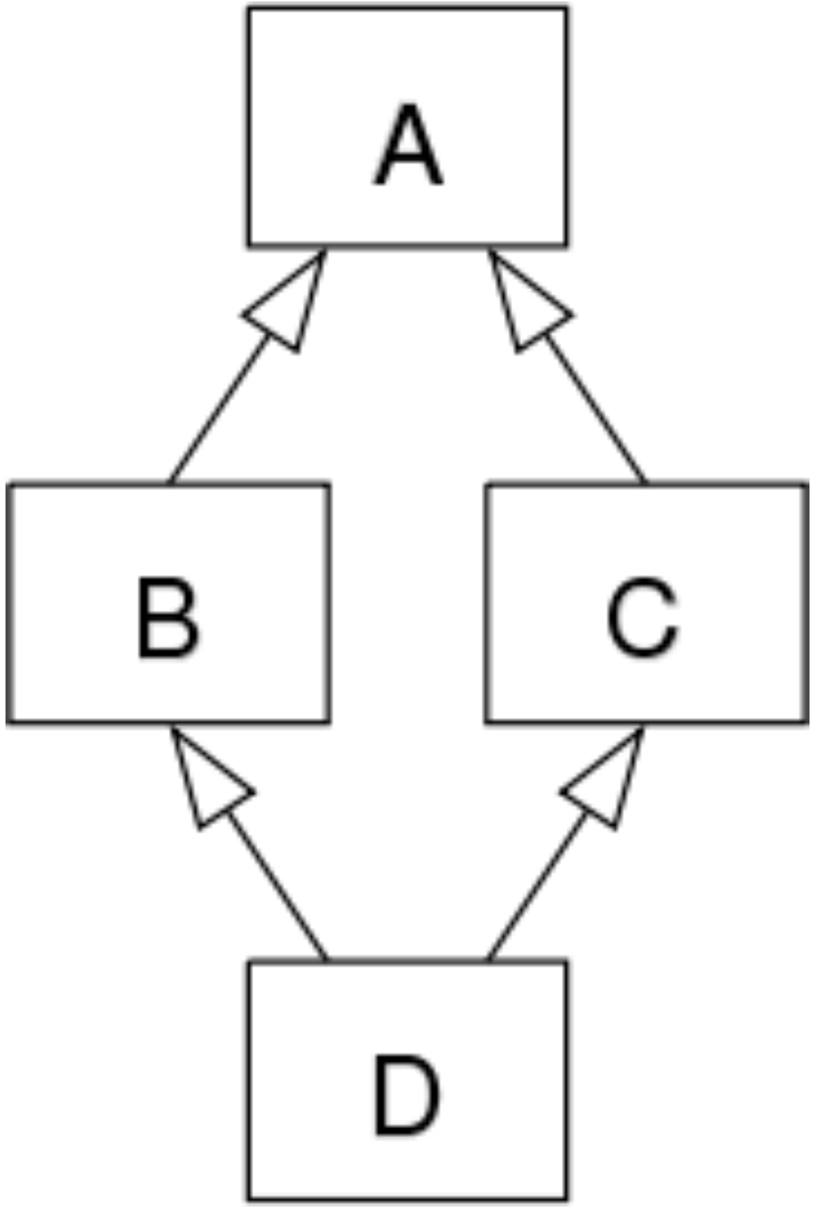
- The diamond problem is an ambiguity that arises with multiple inheritance when two classes B and C inherit from A, and class D inherits from both B and C.

- The result will be the replication of that base class in the derived class that uses multiple inheritance.

- If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?

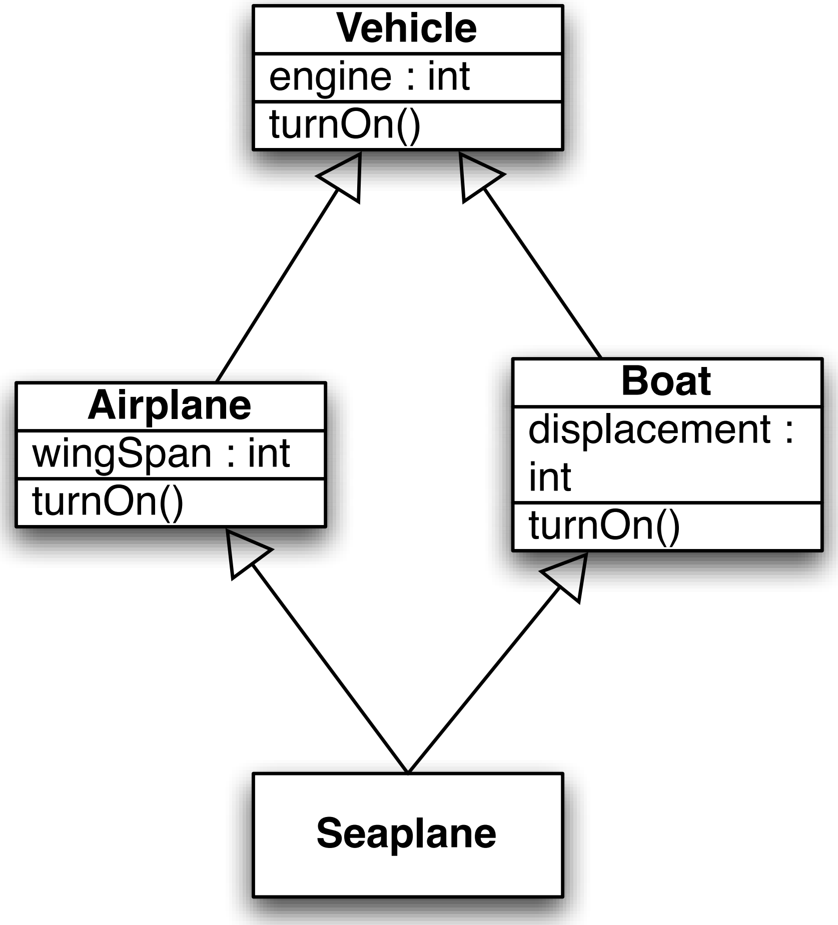
Diamond problem

- The diamond problem is an ambiguity that arises with multiple inheritance when two classes B and C inherit from A, and class D inherits from both B and C.
- The result will be the replication of that base class in the derived class that uses multiple inheritance.
- If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?



Diamond problem

- The diamond problem is an ambiguity that arises with multiple inheritance when two classes B and C inherit from A, and class D inherits from both B and C.
- The result will be the replication of that base class in the derived class that uses multiple inheritance.
- If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?



Virtual inheritance

- Virtual inheritance is a kind of inheritance that solves some of the problems caused by multiple inheritance (particularly the “diamond problem”) by clarifying ambiguity over which ancestor class members to use.
- A multiply-inherited base class is denoted as virtual with the `virtual` keyword.

Virtual inheritance example

```
class Base {
 public:
   ...
 protected:
   int data;
 };

class Der1 : public virtual Base {
 public:
   ...
 };

class Der2 : public virtual Base {
 public:
   ...
 };
```

```
class Join : public Der1, public Der2 {
 public:
   void method()
   {
      data = 1;
      // good: this is now
      // unambiguous, otherwise should
      // have used Der1::data|Der2::...
   }
 };

int main() {
   Join* j = new Join();
   Base* b = j;  // good: this is now
                 // unambiguous

 }
```

Virtual inheritance example

```
class Base {
 public:
   ...
 protected:
   int data;
 };

class Der1 : public virtual Base {
 public:
   ...
 };

class Der2 : public virtual Base {
 public:
   ...
 };
```

this is the key

```
class Join : public Der1, public Der2 {
 public:
   void method()
   {
      data = 1;
      // good: this is now
      // unambiguous, otherwise should
      // have used Der1::data|Der2::...
   }
 };

int main() {
   Join* j = new Join();
   Base* b = j;  // good: this is now
                 // unambiguous

 }
```

Pointer conversions

- Conversions (either implicit or explicit) from a derived class pointer or reference to a base class pointer or reference must refer unambiguously to the same accessible base class object, e.g.:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main () {
     Z z;
     X* pX = &z; // valid
     Y* pY = &z; // valid
     W* pW = &z; // error, ambiguous reference to class W
                 // X's W or Y's W ?
}
```

Pointer conversions

- Conversions (either implicit or explicit) from a derived class pointer or reference to a base class pointer or reference must refer unambiguously to the same accessible base class object, e.g.:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main () {
    Z z;
    X* pX = &z; // valid
    Y* pY = &z; // valid
    W* pW = &z; // error, ambiguous reference to class W
                // X's W or Y's W ?
}
```

Just use virtual inheritance to solve ambiguity

Liskov's Substitution Principle

Liskov substitution principle

- Rules to follow to implement LSP:
  - **Preconditions** cannot be strengthened in the subtype: you cannot require more than the parent
  - **Postconditions** cannot be weakened in the subtype: you cannot guarantee less than the parent
  - **Invariants** must be preserved in the subtype.
  - No new **exceptions** should be thrown, unless the exceptions are subtypes of exceptions thrown by the parent.

Preconditions

- A precondition is a condition which must hold true before executing a method.
- Usually preconditions are concerned with the arguments to a method, or the state of the object, e.g. a method requires that an argument is not null or 0
- If a method in the derived class adds preconditions code that works fine with the base class may not work with the derived one

Postcondition

- A postcondition is a condition which must hold true after running a method.
- Postconditions usually concern the result of a method, or the state of the object.
- If a base class method returns only non empty strings and a derived class method returns only strings that are not null the code using that method may fail.

Invariant

- An invariant is a condition which must hold true both before and after running a method.
- Usually invariants are concerned with the state of the object they're defined on.
    - This is actually one of the more difficult constraints to fulfill, because invariants are often not explicitly defined in code.
- Example: a `Date` class must make sure that there is no Feb. 30th, before and after any call to a method. A derived class should not eliminate this invariant.

Exceptions

- An exception is a way to signal a problem while executing code.
- Code executing a method that may launch an exception deals with these exceptions; exceptions are classes and inherit.
- Code catching exceptions can deal with subtypes exceptions, not with exceptions outside the inheritance hierarchy.

Exceptions

- An exception is a way to signal a problem while executing code.
- Code executing a method that may launch an exception deals with these exceptions; exceptions are classes and inherit.
- Code catching exceptions can deal with subtypes exceptions, not with exceptions outside the inheritance hierarchy.

If these sentences do not make sense to you now, do not despair, there will be a lecture both them.

Violations of LSP

- RTTI and `typeid`, i.e. checking of dynamic types in C++ are not compatible with LSP.
- The LSP makes clear that in OOD the ISA relationship pertains to behavior. Not intrinsic private behavior, but extrinsic public behavior; behavior that clients depend upon.

LSP

```
class Rectangle {
public:
      virtual void setWidth(double w)  {itsWidth=w;}
      virtual void setHeight(double h) {itsHeight=h;}
      double       getHeight() const   {return itsHeight;}
      double       getWidth() const    {return itsWidth;}
private:
   double itsHeight;
   double itsWidth;
};

class Square : public Rectangle {
public:
   virtual void setWidth(double w);
   virtual void setHeight(double h);
};

void Square::setWidth(double w) {
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
}

void Square::setHeight(double h) {
  Rectangle::setHeight(h);
  Rectangle::setWidth(h);
}
```

ng of dynamic
ble with LSP.

OD the ISA
or. Not intrinsic
public behavior;
pon.

LSP

```
class Rectangle {
public:
      virtual void setWidth(double w)  {itsWidth=w;}
      virtual void setHeight(double h) {itsHeight=h;}
      double       getHeight() const   {return itsHeight;}
      double       getWidth() const    {return itsWidth;}
private:
   double itsHeight;
   double itsWidth;
};

class Square : public Rectangle {
public:
   virtual void setWidth(double w);
   virtual void setHeight(double h);
};

void Square::setWidth(double w) {
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
}

void Square::setHeight(double h) {
  Rectangle::setHeight(h);
  Rectangle::setWidth(h);
}
```

```
// f expects that Rectangle can work after SetWidth()
// f may be fooled by the behavior of Square
void f(Rectangle& r) {
     r.setWidth(32); // calls Rectangle::SetWidth
}

// g expects that changing height or width leaves
// the other parameter unchanged
// g can not be fooled by Square
void g(Rectangle& r) {
     r.setWidth(5);
     r.setHeight(4);
     assert(r.getWidth() * r.getHeight() == 20);
}
```

In order for the LSP to hold, and with it the Open-Closed principle, all derivatives must conform to the behavior that clients expect of the base classes that they use.

```
      double        getWidth() const    {return itsWidth;}
private:
   double itsHeight;
   double itsWidth;
};

class Square : public Rectangle {
public:
   virtual void setWidth(double w);
   virtual void setHeight(double h);
};

void Square::setWidth(double w) {
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
}

void Square::setHeight(double h) {
  Rectangle::setHeight(h);
  Rectangle::setWidth(h);
}
```

```
// f expects that Rectangle can work after SetWidth()
// f may be fooled by the behavior of Square
void f(Rectangle& r) {
     r.setWidth(32); // calls Rectangle::SetWidth
}

// g expects that changing height or width leaves
// the other parameter unchanged
// g can not be fooled by Square
void g(Rectangle& r) {
     r.setWidth(5);
     r.setHeight(4);
     assert(r.getWidth() * r.getHeight() == 20);
}
```

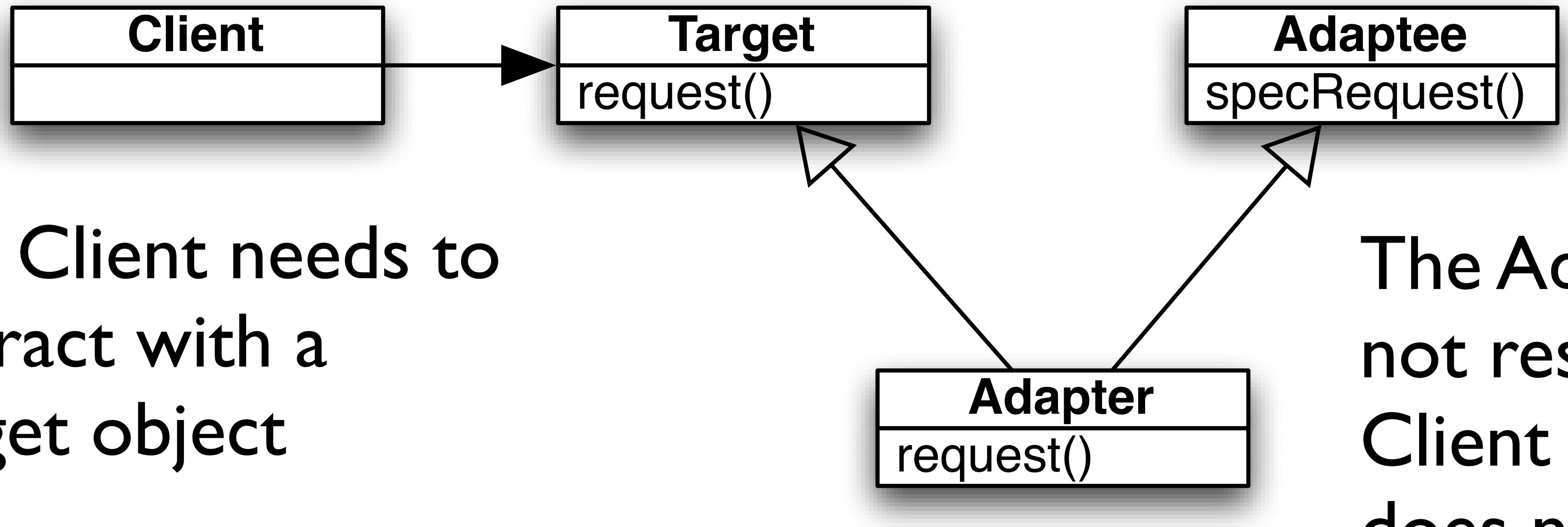
Class adapter

Virtual methods, private inheritance, abstract classes and multiple inheritance, all put together

Use of multiple inheritance

- In the following example is shown an interesting use of multiple inheritance, along with abstract class, virtual methods and private inheritance.
- A class (Adapter) adapts the interface of another class (Adaptee) to a client, using the expected interface described in an abstract class (Target)
  - This is the "Class Adapter" pattern: lets classes work together that couldn't otherwise because of compatible interfaces

“Class Adapter” UML class diagram



The Client needs to interact with a Target object

The Adaptee could not respond to Client because it does not have the required method

The Adapter lets the Adaptee to respond to request of a Target by extending both Target and Adaptee

Class Adapter example

```
class Adaptee {
public:
  getAlpha() {return alpha;};
  getRadius() {return radius;};
private:
  float alpha;
  float radius;
};

class Target {
public:
  virtual float getX() = 0;
  virtual float getY() = 0;
};
```

```
class Adapter : private Adaptee, public Target
{
public:
  virtual float getX();
  virtual float getY();
};
float Adapter::getX() {
 return
Adaptee::getRadius()*cos(Adaptee::getAlpha());
}
float Adapter::getY() {
 return
Adaptee::getRadius()*sin(Adaptee::getAlpha());
}
```

The Client can’t access Adaptee methods because Adapter has obtained them using private inheritance

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++", cap. 3, cap. 4 - pp. 95-104
- B. Stroustrup, "C++, guida essenziale per programmatori" - pp. 39-47
- B. Stroustrup, "C++ - Linguaggio, libreria standard, principi di programmazione" - pp. 525-556
- L.J. Aguilar, "Fondamenti di programmazione in C++. Algoritmi, strutture dati e oggetti" - cap. 8, 9, 12
- D.S. Malik, "Programmazione in C++" - cap. 6, 8
- Thinking in C++, 2nd ed. Volume 1, cap. 4, 5

Credits

- These slides are (heavily) based on the material of:
  - Dr. Ian Richards, CSC2402, Univ. of Southern Queensland
  - Prof. Paolo Frasconi, IIN 167, Univ. di Firenze
  - Scott Meyers, "Effective C++", 3rd edition, Addison-Wesley
  - Stanley B. Lippman, "C++ Primer", 5th edition, Addison-Wesley