EDAF50 – C++ Programming

8. Classes and polymorphism.

Sven Gestegård Robertz

Computer Science, LTH

2018

Outline

Polymorphism and dynamic binding

Polymorphism

Overloading	*Static binding*
Generic programming (templates)	*Static binding*
Virtual functions	*Dynamic binding*

Static binding: The meaning of a construct is decided *at compile-time*

Dynamic binding: The meaning of a construct is decided *at run-time*

Concrete and abstract types

A *concrete type* behaves "just like built-in-types":

- The *representation* is part of the *definition* [1]
- Can be placed on the stack, and in other objects
- can be directly refererred to
- Can be copied
- User code *must be recompiled* if the type is changed

An *Abstract types* isolates the user from implementation details and *decouples the interface from the representation*:

- The representation of objects (*incl. the size!*) is not known
- Can only be accessed through pointers or references
- Cannot be instantiated (*only concrete subclasses*)
- Code using the abstract type *does not need to be recompiled* if the concrete subclasses are changed

[1] can be private, but is known

Concrete and abstract types
A concrete type: Vector

```
class Vector {
public:
    Vector(int l = 0) :p{new int[l]},sz{l} {}
    ~Vector() {delete[] elem;}
    int size() const {return sz;}
    int& operator[](int i) {return elem[i];}
private:
    int *elem;
    int sz;
};
```

Generalize: *extract interface*

```
class Container {
public:
    int size() const;
    int& operator[](int o);
};
```


Concrete and abstract types
Generalization: an abstract type, Container

```
class Container {
public:
    virtual int size() const  =0;
    virtual int& operator[](int o) =0;
    virtual ~Container() {}
};
```

- *pure virtual* function
- Abstract class
- or interface in Java

```
class Vector :public Container {
public:
    Vector(int l = 0) :p{new int[l]},sz{l} {}
    ~Vector() {delete[] elem;}
    int size() const override {return sz;}
    int& operator[](int i) override {return elem[i];}
private:
    int *elem;
    int sz;
};
```

- extends (or implements) Container in Java
- **override** ⇔ @Override in Java (C++11)
- A polymorph type needs a virtual destructor

Concrete and abstract types
Use of an abstract class

```
void fill(Container& c, int v)
{
    for(int i=0; i!=c.size(); ++i){
        c[i] = v;
    }
}
void print(Container& c)
{
    for(int i=0; i!=c.size(); ++i){
        cout << c[i] << " " ;
    }
    cout << endl;
}
void test_container()
{
    Vector v(10);

    print(v);
    fill(v,3);
    print(v);
}
```


Concrete and abstract types
Use of an abstract class

Assume that we have two other subclasses to `Container`

```
class MyArray : public Container { ...};
class List : public Container { ...};

void test_container()
{
    Vector v(10);
    print(v);
    fill(v,7);
    print(v);

    MyArray a(5);
    fill(a,0);
    print(a);

    List l{1,2,3,4,5,6,7};
    print(l);
}
```

- Dynamic binding of `Container::size()` and `Container::operator[]()`

Concrete and abstract types
Variant, without changing Vector

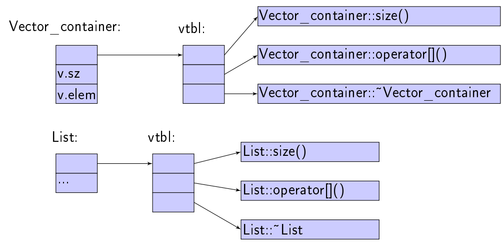
Instead of changing `Vector` we can use it in a new class:

```
class Vector_container :public Container {
public:
    Vector_container(int l = 0) :v{l} {}
    ~Vector_container() =default;
    int size() const override {return v.size();}
    int& operator[](int i) override {return v[i];}
private:
    Vector v;
};
```

- `Vector` is a concrete class
- Note that `v` is a `Vector` object, not a reference
  - Different from Java
- The destructor of a member variable (here, `v`) is implicitly called by the default destructor

Dynamic binding

- virtual function table (*vtbl*)
  - contains pointers to the virtual functions of the object
  - each class with virtual member function(s) has a vtbl
  - each object of such a class has a pointer to the vtbl of the class
  - calling a virtual function (typically) $< 25\%$ more expensive



Class templates
The Container classes

```
class Container {
public:
    virtual int size() const  =0;
    virtual int& operator[](int o) =0;
    virtual ~Container() {}
    virtual void print() const =0;
};

class Vector :public Container {
public:
    explicit Vector(int l);
    ~Vector();
    int size() const override;
    int& operator[](int i) override;
    virtual void print() const override;
private:
    int *p;
    int sz;
};
```

- generalize on element type

Class templates
Generic Container and Vector

```
template <typename T>
class Container {
public:
    using value_type = T;
    virtual size_t size() const =0;
    virtual T& operator[](size_t o) =0;
    virtual ~Container() {}
    virtual void print() const =0;
};
template <typename T>
class Vector :public Container<T> {
public:
    Vector(size_t l = 0) :p{new T[l]},sz{l} {}
    ~Vector() {delete[] p;}
    size_t size() const override {return sz;}
    T& operator[](size_t i) override {return p[i];}
    virtual void print() const override;
private:
    T *p;
    size_t sz;
};
```

Constructors and inheritance
Rules for the base class constructor

- The default constructor of the base class is implicitly called
  - if it exists!
- Arguments to the base class constructor
  - are given in the *member initializer list* in the derived class constructor.
  - *the name of the base class* must be used.
    (super() like in Java does not exist due to multiple inheritance.)

Constructors and inheritance

Order of initialization in a constructor (for a derived class)

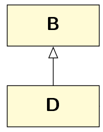
1. *The base class is initialized*: The base class ctor is called
2. *The derived class is initialized*: Data members (in the derived class) is initialized
3. The constructor body of the derived class is executed

Explicit call of base class constructor in the member initializer list

```
D::D(param...) :B(param...), ... {...}
```

Note:

- Constructors are not inherited
- *Do not call virtual functions in a constructor.*:
  In the base class B, **this** is of type B*.



Constructors and inheritance

Constructors are not inherited

```
class Base{
public:
    Base(int i) :x{i} {}
    virtual void print() {cout << "Base: " << x << endl;}
private:
    int x;
};

class Derived :public Base {
};

void test_ctors()
{
    Derived b(5); //no matching function for call to
                  //Derived::Derived(int)
    Derived b2; //use of deleted function Derived::Derived()
}
```


Constructors and inheritance

using: make the base class constructor visible (C++11)

```
class Base{
public:
    Base(int i) :x{i} {}
    virtual void print() {cout << "Base: " << x << endl;}
private:
    int x;
};

class Derived :public Base {
  using Base::Base;
};

void test_ctors()
{
    Derived b2(5); // OK!
    Derived b; //use of deleted function Derived::Derived()
    b.print();
}
```


Constructors vid inheritance

Now with a default constructor

```
class Base{
public:
    Base(int i=0) :x{i} {}
    virtual void print() {cout << "Base: " << x << endl;}
private:
    int x;
};

class Derived :public Base {
  using Base::Base;
};

void test_ctors()
{
    Derived b;     // OK!
    b.print();
    Derived b2(5); // OK!
    b2.print();
}
```


Inherited constructors
rules

- **using** makes all base class constructors inherited, except
  - those hidden by the derived class (with the same parameters)
  - default, copy, and move constructors
    ⇒ *if not defined, synthesized as usual*
- default arguments in the super class gives multiple inherited constructors

Copying and inheritance

- The copy constructor shall copy *the entire object*
  - typically: call the base class copy-constructor
- The same applies to **operator**=
- Different from the destructor
  - A destructor shall only deallocate what has been allocated in the class itself. The base class destruction is implicitly called.
- The synthesized default constructor or the copy control members are deleted in a derived class if the corresponding function is deleted in the base class.
  (i.e., **private** or =**delete**)
  - default constructor,
  - copy constructor,
  - copy assignment operator
  - (destructor, but avoid classes without a destructor)
- Base classes should define these =**default**

Destructors and inheritance

Destruction is done in reverse order:

Execution order in a destructor

1 The function body of the derived class destructor is executed
2 The members of the derived class are destroyed
3 The base class destructor is called

The base class destructor must be virtual

Accessibility

The different levels of accessibility

```
class C {
public:
      // Members accessible from any function
protected:
      // Members accessible from member functions
      // in the class or a derived class
private:
      // Members accessible only from member functions
      // in the class
};
```


Accessibility

Accessibility and inheritance

```
class D1 : public B { // Public inheritance
    // ...
};

class D2 : protected B { // Protected inheritance
    // ...
};

class D3 : private B { // Private inheritance
    // ...
};
```


Accessibility

Accessibility and inheritance

	Accessibility i B	Accessibility through D
Public inheritance	public protected private	public protected private
Protected inheritance	public protected private	protected protected private
Private inheritance	public protected private	private private private

The accessibility inside D is *not* affected by the type of inheritance

Function overloading and inheritance

Function overloading does not work as usual between levels in a class hierarchy

```
class C1 {
public:
    void f(int) {cout << "C1::f(int)\n";}
};

class C2 : public C1 {
public:
    void f(); {cout << "C2::f(void)\n";}
};

C1 a;
C2 b;
a.f(5); // Ok, calls C1::f(int)
b.f();  // Ok, calls C2::f(void)
b.f(2)  // Error! C1::f is hidden!
b.C1::f(10); // Ok
```

Function overloading and inheritance
Make base class names visible with `using`

Function overloading between levels of a class hierarchy

```
class C1 {
public:
    void f(int); {cout << "C1::f(int)\n";}
};

class C2 : public C1 {
public:
    using C1::f;
    void f(); {cout << "C2::f(void)\n";}
};

//...
C1 a;
C2 b;
a.f(5); // Ok, calls C1::f(int)
b.f();  // Ok, calls C2::f(void)
b.f(2)  // Ok, calls C1::f(int)
```


Inheritance and *scope*

- The *scope* of a derived class is *nested* inside the base class
  - Names in the base class are visible in derived classes
  - *if not hidden* by the same name in the derived class
- Use the *scope operator* :: to access hidden names
- Name lookup happens at compile-time
  - *static type* of a pointer or reference determines which names are visible (like in Java)
  - Virtual functions must have the same parameter types in derived classes.

Inheritance without virtual functions

In C++ member functions are *not virtual unless declared so*. (Difference from Java)

- It is possible to inherit from a class and *hide* functions.
- Base class funcions can be called explicitly
- can be used to “extend” a function. (Add things before and after the function.)

Inheritance without virtual functions
Example

```
struct Clock{
    Clock(int h, int m, int s) :seconds{60*(60*h+m) + s} {}
    Clock& tick();       // NB! Not virtual
    int get_ticks() {return seconds;}
private:
    int seconds;
};
struct AlarmClock : public Clock {
    using Clock::Clock;
    void setAlarm(int h, int m, int s);
    AlarmClock& tick(); // hides Clock::tick()
    void soundAlarm();
private:
    int alarmTime;
};

AlarmClock& AlarmClock::tick()
{
    Clock::tick();   // explicit call of base class function
    if(get_ticks() == alarmTime)  soundAlarm();
    return *this;
}
```


Pitfalls

- Type conversion
- Copying objects of polymorph types

Type conversion

- Be careful with type casts
  - In particular (Derived*) base_class_pointer
  - No safety net, no ClassCastException
- Use **dynamic_cast** (returns nullptr or throws if not OK)

```
Vector v;

Container* c = &v;

if(dynamic_cast<Vector*>(c)) {
    cout << " *c instanceof Vector\n";
}
```

- **typeid** corresponds to .getClass() in Java

```
if(typeid(*c) == typeid(Vector)) {
    cout << " *c is a Vector\n";
}
```

Object slicing
Example

```
class Point {...};
class Point3d : public Point {...};

Point3d b;
Point a = b;
```

Not dangerous, but a only contains the `Point` part of `b`

```
Point3d b1;
Point3d b2;

Point& point_ref = b2;
point_ref = b1;
```

Wrong! `b2` now contains the `Point` part of `b1` and the `Point3d` part of its old value.

Object slicing
Example

```
struct Point{
    Point(int xi, int yi) :x{xi}, y{yi} {}
    virtual void print() const; // prints Point(x,y)
    int x;
    int y;
};

struct Point3d :public Point{
    Point3d(int xi, int yi, int zi) :Point(xi,yi), z{zi} {}
    virtual void print() const; // prints Point3d(x,y,z)
    int z;
};

void test_slicing() {
    Point3d q1{1,2,3};
    Point3d q2{3,4,5};

    q2.print();      Point3d(3,4,5)
    Point& pr = q2;
    pr = q1;
    q2.print();      Point3d(1,2,5)
}
```

solution: `virtual operator=`

Object slicing
Solution with virtual `operator=`

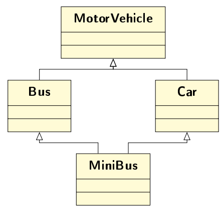
```
struct Point {
  ...
  virtual Point& operator=(const Point& p) =default;
};

struct Point3d :public Point{
  ...
  virtual Point3d& operator=(const Point& p) noexcept;
};

Point3d& Point3d::operator=(const Point& p) noexcept
{
  Point::operator=(p);
  auto p3d = dynamic_cast<const Point3d*>(&p);
  if(p3d){
    z = p3d->z;
  } else {
    z = 0;
  }
  return *this;
}
```

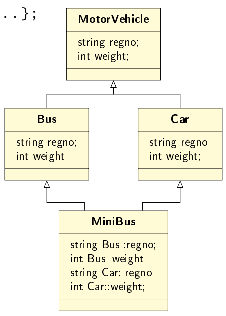

Multiple inheritance

- A class can inherit from multiple base classes
- cf. implementing multiple `interface`s in Java
  - Like in Java if at most one of the base classes have member variables
  - Can be tricky otherwise
- *The diamond problem*
  - How many `MotorVehicle` are there in a `MiniBus`?



Multiple inheritance
How many `MotorVehicle` are there in a `MiniBus`?

```
class MotorVehicle {...};
class Bus : public MotorVehicle {...};
class Car : public MotorVehicle {...};
class MiniBus : public Bus, public Car {...};
```



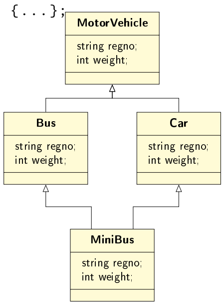
Multiple inheritance
The diamond problem

- A common base class is included multiple times
  - Multiple copies of member variables
  - Members must be accessed as `Base::name` to avoid ambiguity
- if *virtual inheritance* is not used

Multiple inheritance
Virtual inheritance

Virtual inheritance: Derived classes share the base class instance.
(The base class is only included once)

```
class MotorVehicle {...};
class Bus : public virtual MotorVehicle {...};
class Car : public virtual MotorVehicle {...};
class MiniBus : public Bus, public Car {...};
```



Example: A class hierarchy

```
class Animal{
public:
  void speak() const { cout << get_sound() << endl;}
  virtual string get_sound() const =0;
  virtual ~Animal() =default;
};

class Dog :public Animal{
public:
  string get_sound() const override {return "Woof!";}
};
class Cat :public Animal{
public:
  string get_sound() const override {return "Meow!";}
};
class Bird :public Animal{
public:
  string get_sound() const override {return "Tweet!";}
};
class Cow :public Animal{
public:
  string get_sound() const override {return "Moo!";}
```


Example
Use (not polymorphic)

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  d.speak();      Woof!
  c.speak();      Meow!
  b.speak();      Tweet!
  w.speak();      Moo!
}
```


Example
Call by reference

```
void test_polymorph(const Animal& a)
{
    a.speak();
}

int main()
{
    Dog d;
    Cat c;
    Bird b;
    Cow w;

    test_polymorph(d);          Woof!
    test_polymorph(c);          Meow!
    test_polymorph(b);          Tweet!
    test_polymorph(w);          Moo!
}
```


Example
Container with polymorph objects

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  vector<Animal> zoo{d,c,b,w};

  for(auto x : zoo){
    x.speak();
  };

}
```

```
error: cannot allocate an object of abstract type 'Animal'
```


Example
Must use container of pointers

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  vector<Animal*> zoo{&d,&c,&b,&w};

  for(auto x : zoo){
    x->speak();       Woof!
  };                  Meow!
                      Tweet!
}                     Moo!
```

Example
A class hierarchy

```
struct Foo{
    virtual void print() const {cout << "Foo" << endl;}
};

struct Bar :Foo{
     void print() const override {cout << "Bar" << endl;}
};

struct Qux :Bar{
     void print() const override {cout << "Qux" << endl;}
};
```


Polymorph class
example, *object slicing*

What is printed?

```
void print1(const Foo* f)
{
  f->print();
}
void print2(const Foo& f)
{
  f.print();
}
void print3(Foo f)
{
  f.print();
}
```

```
void test()
{
   Foo* a = new Bar;
   Bar& b = *new Qux;
   Bar  c = *new Qux;

   print1(a);  Bar
   print1(&b); Qux
   print1(&c); Bar
   std::cout << std::endl;
   print2(*a); Bar
   print2(b);  Qux
   print2(c);  Bar
   std::cout << std::endl;
   print3(*a); Foo
   print3(b);  Foo
   print3(c);  Foo
}
```


Next lecture
Standard library containers. More about inheritance.

References to sections in Lippman

Sequential containers 9.1 – 9.3

Container Adapters 9.6

Associative containers chapter 11

Type conversions and polymorphism 15.2.3

Swap 13.3

Suggested reading

References to sections in Lippman

Dynamic polymorphism and inheritance chapter 15 – 15.4

Accessibility and scope 15.5 – 15.6

Type conversions and polymorphism 15.2.3

Inheritance and resource management 15.7

Polymorph types and containers 15.8

Multiple inheritance 18.3

Virtual base classes 18.3.4 – 18.3.5