

EDAF50 – C++ Programming

8. Classes and polymorphism.

Sven Gestegård Robertz
Computer Science, LTH

2024



Outline

- 1 **Classes and const**
 - const for objects and members
- 2 **Polymorphism and inheritance**
 - Concrete and abstract types
 - Virtual functions
 - Constructors and destructors
 - Accessibility
 - Inheritance without polymorphism
- 3 **Usage**
- 4 **Pitfalls**
- 5 **Multiple inheritance**

Constant objects

- ▶ **const** means “I promise not to change this”
- ▶ Objects (variables) can be declared **const**
 - ▶ “I promise not to change the variable”
- ▶ References can be declared **const**
 - ▶ “I promise not to change the referenced object”
 - ▶ a **const&** can refer to a non-**const** object
 - ▶ common for function parameters
- ▶ Member functions can be declared **const**
 - ▶ “I promise that the function does not change the state of the object”
 - ▶ *technically: implicit declaration* **const T* const this;**

Constant objects

Example

const references and const functions

```
class Point{
public:
    Point(int xi, int yi) :x{xi},y{yi}{}
    int get_x() const {return x;}
    int get_y() const {return y;}
    void set_x(int xi) {x = xi;}
    void set_y(int yi) {y = yi;}
private:
    int x;
    int y;
};

void example(Point& p, const Point& o) {
    p.set_y(10);
    cout << "p: " << p.get_x() << ", " << p.get_y() << endl;

    o.set_y(10);
    cout << "o: " << o.get_x() << ", " << o.get_y() << endl;
}

passing 'const Point' as 'this' argument discards qualifiers
```

Constant objects

Example

Note **const** in the declaration (and definition!) of the member function **operator[](int) const**: (“*const is part of the name*”)

```
class Vector {
public:
    //...
    double operator[](int i) const;    // function declaration
    //...
private:
    double* elem;
    //...
};

double Vector::operator[](int i) const    // function definition
{
    return elem[i];
}
```

Constant objects

Example: `const` overloading

The functions `operator[](int)` and `operator[](int) const` *are different functions*.

Example

```
class Vector {  
    double& operator[](int i)      {return elem[i];}  
    double  operator[](int i) const {return elem[i];}  
private:  
    double* elem;  
    //...  
};
```

- ▶ If `operator[]` is called on a
 - ▶ non-`const` object, a *reference* is returned
 - ▶ `const` object, a *value* is returned
- ▶ The assignment `v[2] = 10;` only works on a non-`const` `v`.

Polymorphism

Overloading	<i>Static binding</i>
Generic programming (templates)	<i>Static binding</i>
Virtual functions	<i>Dynamic binding</i>

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

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

Polymorphism

Static

```
void foo(int);  
void foo(double);  
  
foo(17);  
  
std::vector<int> v;  
  
std::sort(begin(v), end(v));
```

Dynamic

```
struct Animal{  
    virtual void speak();  
};  
  
struct Dog :public Animal{  
    void speak();  
};  
  
struct Cat :public Animal{  
    void speak();  
};  
  
void use(Animal& a)  
{  
    a.speak();  
}  
  
use(Dog{});
```


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 referred to
- ▶ Can be copied
- ▶ User code *must be recompiled* if the type is changed

An *Abstract types* isolates the user from implementation details

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

¹can be private, but is known

Concrete and abstract types

A concrete type: Vector

```
class Vector {  
public:  
    Vector(int l = 0) : elem{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:  
    virtual int size() const;  
    virtual 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() =default;
    // copy and move...
};
```

- ▶ *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** \Leftrightarrow @Override in Java (C++11)
- ▶ A polymorph type needs a virtual destructor

Destructors must be `virtual`

Polymorph types are used through base class pointers:

```
Container* c = new Vector(10);
```

```
// use...
```

```
delete c;
```

- ▶ The destructor is called through a `Container*`.
- ▶ `~Container()` is called.
- ▶ If not `virtual`, `~Vector()` is never called \Rightarrow memory leak.

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(const 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:

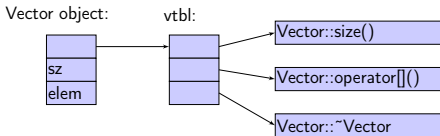
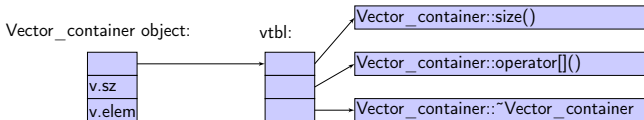
```
class Vector_container :public Container {
public:
    Vector_container(int l = 0) :v{1} {}
    ~Vector_container() =default;
    int size() const override {return v.size();}
    int& operator[](int i) override {return v[i];}
private:
    Vector v;
};
```

- ▶ 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

```
int example(Container& c)
{
    return c.size();
}
```



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)

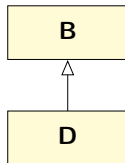
- 1 *The base class is initialized*: The base class ctor is called
- 2 *The derived class is initialized*: Data members (in the derived class) is initialized
- 3 The constructor body of the derived class is executed

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

Constructors and 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 destructor is implicitly called.
- ▶ The synthesized special member functions are *deleted in a derived class* if the corresponding function is *deleted in the base class*.
(i.e., **private** or **=delete**)
 - ▶ default constructor,
 - ▶ copy (and move) constructor,
 - ▶ copy (and move) assignment operator
 - ▶ (destructor, but avoid classes without a destructor)
- ▶ Base classes should define these **=default**

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

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 and inheritance

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

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

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

Accessibility and inheritance

	Accessibility in 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 functions 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;
}
```


Example

Container with polymorph objects

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

    std::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;

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

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

- ▶ Type conversion
- ▶ Non-virtual destructor
- ▶ Copying objects of polymorph types

Type conversion and run-time type info

- ▶ Be careful with type casts
 - ▶ In particular (Derived*) base_class_pointer
 - ▶ and **static_cast**<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() comparison in Java

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

Destructors must be virtual

Example: memory leak

```
struct Base {
    Base() = default;
    ~Base() = default;
    virtual void do_stuff();
    ...
};
struct Derived : public Base {
    Derived() :Base(), f{new Foo()} {}
    ~Derived() {delete f;}
    void do_stuff();
    ...
private:
    Foo* f
};
```

```
Base* p = new Derived();
...
delete p;
```

As `p` has static type `Base*`, the destructor `~Base()` is run when `delete p` is called. If that is not virtual, `~Derived()` is not run \Rightarrow memory leak. (The standard says Undefined Behaviour)

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=

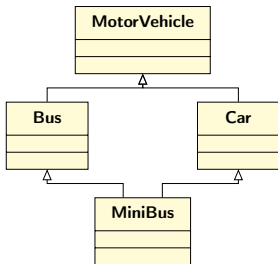
```
struct Point {
    ...
    virtual Point& operator=(const Point& p) =default;
};

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

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


Multiple inheritance

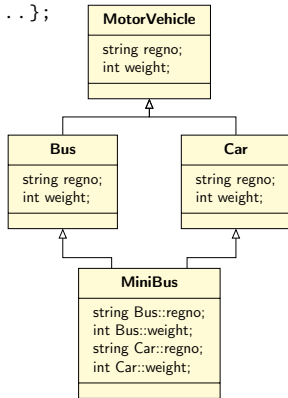
- ▶ A class can inherit from multiple base classes
- ▶ cf. implementing multiple interfaces in Java
 - ▶ Like in Java if at most one of the base classes have member variables
 - ▶ Can be tricky otherwise
- ▶ *The diamond problem*
 - ▶ How many MotorVehicle are there in a MiniBus?



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

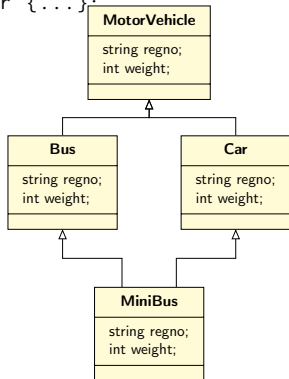
Multiple inheritance

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 {...};
```

The *most derived class* (Minibus) must call *the constructor of the grandparent* (MotorVehicle).



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

Tuples 17.1

Swap 13.3

Moving objects 13.6

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