

81. Polymorphism

Objectives

- Write virtual methods
- Understand and use polymorphism
- Understand and use virtual destructor

Constructor

Polymorphism = “many forms”

VERY IMPORTANT!!! LISTEN!!!

Suppose class `C` is a subclass of `P`. You already know that the following is valid:

```
C c;  
P p;  
p = c;
```

However the following will not compile:

```
c = p;
```

In particular the following is also valid:

```
C c;  
P * p = &c; // p points to c
```

And this is also valid:

```
C c;  
P & ref = c; // ref references c
```

Suppose both `P` and `C` have methods `f()`. If I call

```
p->f();
```

which `f()` is invoked? (or `ref.f()` in the case where `ref` is a reference).

All you need to do is to do this experiment:

```

#include <iostream>

class P
{
public:
    void f()
    {
        std::cout << "P::f\n";
    }
};

class C: public P
{
public:
    void f()
    {
        std::cout << "C::f\n";
    }
};

int main()
{
    C c;
    P * p = &c;
    p->f();           // parent's f
    return 0;
}

```

AHA! ... the parent's `f()` is called! This should be a surprise to you.

After all we always think of

```
p->f();
```

as

```
(*p).f();
```

and since `(*p)` is the object `c`, then should the above be

```
c.f();
```

and therefore should the code be calling the `f()` in `C`. But remember that from the previous set of notes on inheritance, when you perform

```
p = c;
```

what happens is that there's type conversion happening behind your back:

```
p = P(c);
```

so that the information in `C` and not in `P` is in fact lost. That's one way of

remembering that the `f()` called is `P::f()`.

Summary time ... so remember that our `p`

```
P * p = &c;
```

is very special. In a sense it has **two natures**:

- From the type in the declaration `P * p`, `p` is related to type `P`.
- However from the value that `p` points to, `p = &c`, `p` is related to value of type `C`.

And for the choice of method `f()`,

```
p->f();
```

the method in the parent is chosen.

Exercise. Suppose `G` is the parent class of `P` (using public inheritance) and `P` is the parent class of `C` (using public inheritance). Suppose in `G`, `P`, and `C` classes, there is a method

```
void m() { ... }
```

Suppose have this code fragment:

```
G * p;
C c;
p = &c;
p->m();
```

Which `m()` was called?

Virtual Methods


Recall that we have this

```
#include <iostream>

class P
{
public:
    void f()
    {
        std::cout << "P::f\n";
    }
};

class C: public P
{
public:
    void f()
    {
        std::cout << "C::f\n";
    }
};

int main()
{
    C c;
    P * p = &c;
    p->f();
    return 0;
}
```



Parent's f() called

The `f()` that is called is `P::f()`.

Now what if I actually want `C::f()`, i.e., the `f()` in the child class?

Now we declare a method to be **virtual** in the base class.

```
#include <iostream>

class P{
public:
    virtual void f() { std::cout << "D::f\n"; }
};

class C: public P
{
public:
    void f() { std::cout << "C::f\n"; }
};

int main()
{
    C c;
```

```
D * p = &c;  
p->f();  
return 0;  
}
```



Child's `f()` called

We say that `P::f()` is **virtual**.

The fact that `P::f()` is virtual also makes `C::f()` virtual as well. This means that if `C1` is a subclass of `C` and we do

```
C c;  
C1 c1;  
P * p = c;  
p->f(); // C::f() is called  
P * p1 = c1;  
p1->f(); // C1::f() is called
```

Because of that, although it's not necessary, it's common to put the word `virtual` next to the `f()` in `C` as well:

```
class P  
{  
public:  
    virtual void f() { std::cout << "P::f\n"; }  
};  
  
class C : public P  
{  
public:  
    virtual void f() { std::cout << "C::f\n"; }  
};
```

Dynamic & Static Binding

In the earlier sections, we saw that there are two ways to choose the `f()` to execute, either `P::f()` or `C::f()`.

Dynamic binding refers to the choice of method (between the parent method and the child method with the same name) where the method is virtual. This is also called **late binding**

Static binding refers to the choice of method (between the parent method and the child method with the same name) where the method is not virtual. This is also known as **early binding**.

```
C c;
P * p = c;
```

Why is it called dynamic (or late) or static (or early)?

Make sure you run the following really important example ...

Suppose `C0` and `C1` are both subclass of `P`. There are virtual functions `f` defined in `P`, `C0`, `C1`.

```
#include <iostream>
class P
{
public:
    virtual void f() { std::cout << "P::f\n"; }
};

class C: public P
{
public:
    virtual void f() { std::cout << "C::f\n"; }
};

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

int main()
{
    P * p;
    C c;
    C1 c1;
    int i;
    std::cout << "Do you want C or C1? (0 or 1) ";
    std::cin >> i;
    if (i == 0)
```

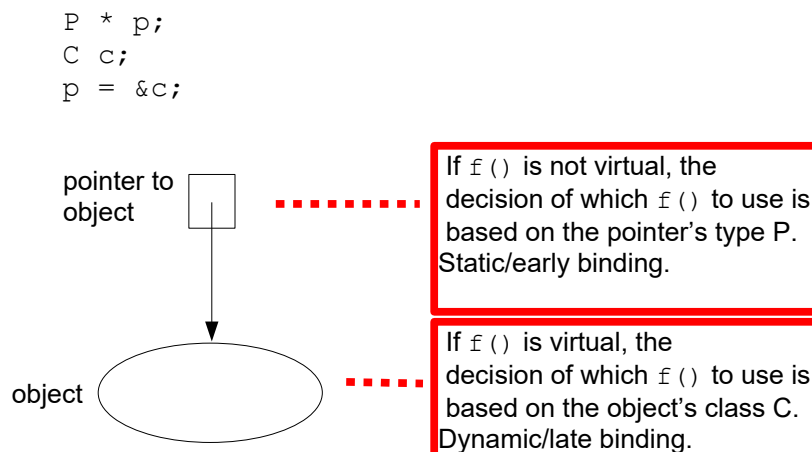
```

{
    p = &c;
}
else
{
    p = &c1;
}
p->f();
return 0;
}

```

In the above, the decision of which $f()$ ($C::f()$ or $C1::f()$) to use is **not known during compilation** but is known **only during runtime**. Therefore the binding is **late** and is **dynamic** (depending on runtime situation).

On the other hand, if $f()$ is not virtual – remove the `virtual` keyword – and no matter what you enter, you always execute $P::f()$. Therefore which $f()$ is used is known during compilation (it's $P::f()$) – it's known even before you execute the program since it's already known during compile time. Therefore the decision is **early**. After the program is compiled, you **cannot** change the $f()$ – the choice of $f()$ is **static** and **not dynamic**.



WARNING: Static here refers to static **binding**, i.e., which $f()$ is the $p \rightarrow f()$ bound to. Do not be confused with static members!!!

Make sure you do the following exercises!

Exercise. What will happen if you have a stack of classes in an inheritance hierarchy like this:


```
class P
{
public:
    virtual void f() { std::cout << "P::f\n"; }
};

class C: public P
{
public:
    virtual void f() { std::cout << "C::f\n"; }
};

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

and you have

```
C1 c1;
P * p;
p = &c1;
p->f();
```

What is the output? (Or is there a compilation or runtime error?)

Exercise. What will happen if you have a stack of classes in an inheritance hierarchy like this:

```
class P
{
public:
    virtual void f() { std::cout << "P::f\n"; }
};

class C: public P
{
public:
    virtual void f() { std::cout << "C::f\n"; }
};

class C1: public C
{
public:
};
```

and you have

```
C1 c1;
P * p;
p = &c1;
p->f();
```

What is the output? (Or is there a compilation or runtime error?)

Exercise. Continuing with the above classes, can you run this program

with input of 0? What is the output?

```
P * p;  
C c0;  
C1 c1;  
int i;  
std::cout << "Do you want C0 or C1? (0 or 1) ";  
std::cin >> i;  
p = (i == 0 ? &c0 : &c1);  
p->f();
```

No virtual ... or what if there are no virtual methods

What if C++ does not have dynamic binding? You're forced to write ugly code, placing some method selection computation within your code. This is sometimes known as the **"type field"** method.

Here's the code from before where we avoid using the virtual keyword:

```
#include <iostream>

class P
{
public:
    void f() { std::cout << "P::f()\n"; }
};

class C: public P
{
public:
    void f() { std::cout << "C::f()\n"; }
};

int main()
{
    C c;
    P * p = &c;
    p->f();
    return 0;
}
```

How can we "reach" C::f()??

We add a field to keep track of the class an object comes from and instead of calling we create a function that accepts and determine (based on the type field) which to execute.

```
#include <iostream>

const int PTYPE = 0;
const int CTYPE = 1;

class P
{
public:
    P() : type_(PTYPE) {}
    void f() { std::cout << "P::f()\n"; }
    void set_type(int t0) { type_ = t0; }
    int get_type() { return type_; }
private:
    int type_;
};

class C : public P
```

```
{
public:
    C() { set_type(CTYPE); }
    void f() { std::cout << "C::f()\n"; }
};

void f(const P * p)
{
    switch (p->get_type())
    {
    case PTYPE:
        p->f();
        break;
    case CTYPE:
        const C * q = (const C *) (p);
        q->f();
        break;
    }
}

int main()
{
    C c;
    P * p = &c;
    f(p);
    return 0;
}
```

Why is it important? Polymorphism

Whenever you have an object with methods with a similar name and you can't decide which one method to call until during runtime, then you will need polymorphism.

Suppose you have an array of pointers to the `BasicSpaceship` class. Also, `SpecialSpaceship` is a subclass of `BasicSpaceship`. In both classes you have an `update()` method. The `update()` method is different for the two classes: the method basically moves the objects to a new position on the screen.

```
BasicSpaceship * s = new BasicSpaceship*[100];
int M = randint(); // random int from 0 to 99
for (int i = 0; i < M; i++)
{
    s[i] = new SpecialSpaceship();
}
for (int i = M; i < 100; i++)
{
    s[i] = new BasicSpaceship();
}
while (1)
{
    for (int i = 0; i < 100; i++)
    {
        s[i]->update();
    }
}
```

Note that in the while-loop:

```
while (1)
{
    for (int i = 0; i < 100; i++)
    {
        s[i]->update();
    }
}
```

You do not know what type of spaceship is `s[3]` and therefore you do not know which update method was invoked. The update method is selected during runtime.

Why is this important? Because if one day you want to add another type of spaceship:

```
BasicSpaceship * s = new BasicSpaceship*[100];
int M = randint(); // rand int from 0 to 99
for (int i = 0; i < M/2; i++)
{
    s[i] = new SpecialSpaceship();
}
```

```

for (int i = M/2; i < M; i++)
{
    s[i] = new BasicSpaceShip();
}
for (int i = M; i < 100; i++)
{
    s[i] = new VeryBasicSpaceShip();
}

```

This means that the while loop does not need to change:

```

while (1)
{
    for (int i = 0; i < M; i++)
    {
        s[i]->update();
    }
}

```

In fact `s[3]` can even point to something different within the loop and everything still works:

```

while (1)
{
    for (int i = 0; i < M; i++)
    {
        s[i]->update();
    }
    delete s[3];
    switch (rand() % 3)
    {
        case 0: s[3] = new SpecialSpaceShip(); break;
        case 1: s[3] = new BasicSpaceShip(); break;
        case 2: s[3] = new VeryBasicSpaceShip(); break;
    }
}

```

Or for instance in your simulation, if `s[3]` is initially pointing to a `SuperShip` object but when hit, it points to a `VeryBasicShip` object.

As long as `s[3]` points to a subclass of `BasicSpaceShip` that has an `update()` method, the while loop will work.

In general:

- You analyze a system and look for objects and classify them into classes.
- You then study what the objects do – as general as possible within the class.
- Design the most general parent classes. Write down relevant methods in parent classes and write specific methods in the children classes.

In your main program:

- In the initialization, you might have pointers (or references) to

- refer to specific children
- After the initialization code, refer as much as possible to the methods in the parents

Going back to the simple game. Suppose game objects (spaceships) move about on their own, and dies when they collide. Thinking as generally as possible, the while loop might look something like this:

```
while (1)
{
    for (int i = 0; i < M; i++) ship[i]->move();
    for (int i = 0; i < M; i++)
    {
        for (int j = i+1; j < M; j++)
            if (ship[i]->collideswith(ship[j]))
            {
                ship[i]->dies();
                ship[j]->dies();
            }
    }
    for (int i = 0; i < M; i++) ship[i]->draw();
}
```

This tells you that the parent class must have the following methods which the specific child classes must overwrite:

- move
- collideswith
- dies
- draw

In the initialization, as long as the pointers `ship[i]` point to a class that subclasses the parent class, the program would work. In particular

- You can switch out subclasses
- You can add new subclasses (as long as they subclass the parent class and has the above four methods)

This is a simple example where there's only one main parent class. In general there are (of course) many parent classes.

For more complicated examples you would have to do into OO analysis/design and design patterns. Take CISS438.

Note from the above example, the methods of the parent class (example: `draw`) might do nothing:

```
class BasicSpaceShip
{
public:
    void draw() {} // empty body
};

class SpecialSpaceShip
{
public:
    void draw() { /* actual drawing code */ }
};
```

This might be the case where every subclass of `BasicSpaceShip` has their own `draw()` method, which means that the `draw()` of `BasicSpaceShip` class is never used anyway.

The only purpose of the `draw` method in the parent is to allow polymorphism to select the child's `draw` method.

There's a way to tell C++ that a method in a parent is meant to be empty and to be fulfilled by a child – see the next section on pure virtual method and abstract base class. Abstract base class are abstract in the sense that you cannot use them to instantiate objects. Their purpose is only to act as a parent for subclasses so that you can perform polymorphism.

This style of programming is called polymorphism. One would say the code is polymorphic. Or, if the pointer points to the child but was declared as a pointer to a parent class:

```
P * p;
```

one would say that is polymorphic, or the object pointed to is polymorphic, i.e., sometimes `p` points to a class `C0` object and sometimes it points to a class `C1` object.

Some people also say the class in this case is a polymorphic type.

Don't forget that I mentioned earlier that you can also use reference for polymorphic programming:

```
P & ref = c0;
```

In this case you would say `ref` is polymorphic.

Virtual constructors and destructors

Constructors cannot be virtual ... try this:

```
class P
{
public:
    virtual P() {}
};

class C: public P
{
public:
    virtual C() {}
};

int main()
{
    P * p = new C;
    return 0;
}
```

A virtual function is about choosing the right method when an object invokes the method. But that implies that the object exists. In the above:

```
P * p = new C;
```

has nothing to do with executing a method through p.

Therefore ... **if you plan to use polymorphism, make sure your constructors are not virtual.**

The above can be caught by the compiler. So it's not as serious as the following problem. Make sure you run it!

```
#include <iostream>

class P
{
public:
    P() : p(new int) {}
    ~P() { std::cout << "P::~~P()\n"; }
private:
    int * p;
};

class C: public P
{
public:
    C() : q(new int) {}
    ~C() { std::cout << "C::~~C()\n"; delete q; }
    int * q;
};
```

```
int main()
{
    P * p = new C;
    delete p;
    return 0;
}
```

Read the output very carefully.

Why is this a problem? The destructor called is the destructor in the parent. Of course the object to destroy is a child class object (which includes a parent part), not just the parent class object. The above code therefore will cause a **memory leak** because the resource used by the child class is not released.

Now try this:

```
#include <iostream>

class P
{
public:
    P() : p(new int) {}
    virtual ~P() { std::cout << "P::~~P()\n"; }
private:
    int * p;
};

class C: public P
{
public:
    C() : q(new int) {}
    ~C() { std::cout << "C::~~C()\n"; delete q; }
    int * q;
};

int main()
{
    P * p = new C;
    delete p;
    return 0;
}
```

You should see two outputs. Why? Because of polymorphism, when you do

```
delete p;
```

the destructor used is `C::~~C()`. But remember from the notes on inheritance that this would execute the body of `C::~~C()` and then `P::~~P()` would be executed.

Therefore ... **if you plan to use polymorphism**

with your (parent) class, and you perform delete on your pointer, you must have a public virtual destructor.

Of course if the child object executes its destructor on its own, then that would be a different story since the pointer is then not responsible for the deallocation. For instance

```
if (true)
{
    C c;
    P * p = &c;
} // c goes out of scope and called c::~~C()
```

Packaging auto destructor

Now look at this:

```
#include <iostream>

class P
{
public:
    virtual ~P() { std::cout << "P::~~P()\n"; }
    void m() { std::cout << "P::m()\n"; }
private:
};

class C: public P
{
public:
    C() : q(new int) {}
    ~C() { std::cout << "C::~~C()\n"; delete q; }
    void m() { std::cout << "C::m()\n"; }
    int * q;
};

int main()
{
    P * p = new C;
    p->m();
    delete p;
    return 0;
}
```

This allows us to work with objects through polymorphism. But this is achieved through pointers. Recall a benefit of classes – you can auto execute the destructor when an object is going out of scope. In the above example, we have to remember to deallocate the memory used by the object that `p` points to:

```
delete p;
```

Using the idea in the chapter on destructor, we do this:

```
#include <iostream>

class P
{
public:
    virtual ~P() { std::cout << "P::~~P()\n"; }
    void m() { std::cout << "P::m()\n"; }
private:
};
```

```

class C: public P
{
public:
    C() : q(new int) {}
    ~C() { std::cout << "C::~~C()\n"; delete q; }
    void m() { std::cout << "C::m()\n"; }
    int * q;
};

class autoP
{
public:
    autoP()
        : p_(new C)
    {}
    ~autoP()
    { delete p_; }
    void m() { p_->m(); }
private:
    P * p_;
};

int main()
{
    autoP p;
    p.m();
    return 0;
}

```

Of course we need to figure out what to do with the copy constructor and operator=. If you don't plan to do that, you can put the prototypes in the private section.

```

#include <iostream>

class P
{
public:
    P() : p(new int) {}
    virtual ~P() { std::cout << "P::~~P()\n"; }
    void m() { std::cout << "P::m()\n"; }
private:
    int * p;
};

class C: public P
{
public:
    C() : q(new int) {}

```

```

    ~C() { std::cout << "C::~~C()\n"; delete q; }
    void m() { std::cout << "C::m()\n"; }
    int * q;
};

class autoP
{
public:
    autoP()
        : p_(new C)
    {}
    ~autoP()
    { delete p_; }
    void m() { p_->m(); }
private:
    autoP(const autoP & p);
    const auto P & operator=(const autoP &);
    P * p_;
};

int main()
{
    autoP p;
    p.m();
    return 0;
}

```

If you do want to allow copy constructor and `operator=`, you have to decide on what they should do, i.e., after

```

autoP p0;
autoP p1;
p1 = p0;

```

does `p1` points to the same object that `p0` points to or is the object that `p1` points to have the same values as the object that `p0` points to (i.e., it's a copy)?

By the way, since the parent destructor is virtual, calling `delete` in `autoP` will execute the destructor in the child class. But what is the parent also has some memory allocation in its constructor like this:

```

#include <iostream>

class P
{
public:
    P() : p(new int) {}
    virtual ~P() { std::cout << "P::~~P()\n"; }
    void m() { std::cout << "P::m()\n"; }
private:

```

```

    int * p;
};

class C: public P
{
public:
    C() : q(new int) {}
    ~C() { std::cout << "C::~~C()\n"; delete q; }
    void m() { std::cout << "C::m()\n"; }
    int * q;
};
...

```

Then the child has to deallocate the memory allocation in the parent.
Assuming the pointer in the parent is private, one might want to do this:

```

#include <iostream>

class P
{
public:
    P() : p(new int) {}
    virtual ~P() { std::cout << "P::~~P()\n"; }
    void m() { std::cout << "P::m()\n"; }
    void deallocate() { delete p; }
private:
    int * p;
};

class C: public P
{
public:
    C() : q(new int) {}
    ~C()
    {
        std::cout << "C::~~C()\n";
        deallocate();
        delete q;
    }
    void m() { std::cout << "C::m()\n"; }
    int * q;
};
...

```

Why is inheritance “less flexible” than object composition?

You will see, after you have written more programs using OOP, that inheritance is not as flexible as object composition.

Why is that?

The reason is because **inheritance cannot be changed during runtime**.

Abstractly speaking if class C is a subclass of D , then a C object c has

```
c.D::x
```

where x is a member (whether it is a member variable or a method). That cannot be changed. However if C has a pointer x that can point to either D or E , at one point in time, you can have

```
c.x    points to a D object
```

and at another time

```
c.x    points to an E object
```

This is possible if $c.x$ is a P pointer where P is a parent of both D and E .

As an example, suppose you have

```
class Car
{
private:
    SteeringWheel steering_wheel_;
};
```

`SelfDrivingCar` wants to have a steering wheel and since a self driving car is a car, you would do this:

```
class Car
{
private:
    SteeringWheel steering_wheel_;
};

class SelfDrivingCar: public Car
{
};
```

But what if you want to model a self driving car that has a joystick instead of a steering wheel? Then you can't. Unless you do


```

class Car
{
private:
    SteeringWheel steering_wheel_;
};

class SelfDrivingCar: public Car
{
private:
    SteeringJoystick * steering_joystick_;
};

```

If a self driving car `self-driving_car` wants a steering joystick, the object would then create a steering joystick object from

```

class SteeringJoystick
{ ... };

```

on the heap and point it's `steering_joystick_` pointer to this object.

BUT ... then the `self-driving_car` object would have a steering wheel:

```
self-driving_car.steering_wheel_
```

and a steering joystick as well:

```
*(self-driving_car.steering_joystick_)
```

Assuming you only want one steering device, this would be a waste of memory, and plus it would open up lots of confusion and incorrect code in the future. Redundant data is always bad.

The point is that your self driving car is of type `SelfDrivingCar` and it has inherited a steering wheel from the `Car` class. This cannot change unless you change the code.

How would we solve this problem where a self driving car can have either a steering wheel or a steering joystick? Furthermore you want the flexibility of changing it during your program's runtime (dynamic).

First you do this:

```

class SteeringDevice
{ ... };

class SteeringWheel: public SteeringDevice
{ ... };

class SteeringJoystick: public SteeringDevice
{ ... };

```

This combines steering wheels and steering joysticks under a common “concept” or class. You can then do this:

```
class Car
{
public:
    Car():
        : steering_device_(new SteeringWheel)
    {}
    // Of course you need to overload the
    // destructor, copy constructor, and
    // operator=.
private:
    SteeringDevice * steering_device_;
};
```

Now a car can have the choice of changing its steering device to either a steering wheel or a steering joystick during runtime:

```
class Car
{
public:
    ...
    void set_to_steering_joystick()
    {
        delete steering_device_;
        steering_device_ = new SteeringJoystick;
    }
private:
    SteeringDevice * steering_device_;
};

class SelfDrivingCar: public Car
{};
```

Then you can do

```
SelfDrivingCar tesla;
... drive with steering wheel ...
tesla.set_to_steering_joystick();
... drive with steering joystick ...
```

But wait a minute ... at least with respect to the steering device, do you see that inheritance is not even necessary!!! Of course if you don't want the program to be too dynamic you can always fix the steering device during constructor call. So suppose all self-driving cars now use joysticks:

```
class Car
{
public:
    Car(int steering_type=0)
    {
        if (steering_type == 0)
            steering_device_ =
```

```
        new SteeringWheel;
    else:
        steering_device_ =
            new SteeringJoystick;
    }
private:
    SteeringDevice * steering_device_;
};

class SelfDrivingCar: public Car
{
public:
    SelfDrivingCar()
        : Car(1)
        {}
};
```

But of course in this case again inheritance seems to be useless again since you can do

```
Car car(1);
```

to get a car with joystick. Therefore there must be other factors specific to self driving cars not present in cars to warrant a class just for self-driving cars.