

C++ PROGRAMMING

DR. YIHSIANG LIOW (JULY 11, 2025)

Contents

80. Inheritance

OBJECTIVES

- Understand inheritance
- Understand how to call the constructor of the superclass
- Understand how the destructor of the superclass is called
- Understand how a method can hide a method of the same signature in the superclass
- Understand how a member variable can hide a member variable in the superclass
- Understand how to call a hidden method in the superclass
- Understand how names are found by the compiler in an inheritance hierarchy
- Understand the difference between composition and inheritance
- Understand how to choose between composition and inheritance
- Understand multiple inheritance
- Understand protected members
- Understand public, protected, and private inheritance

The problem

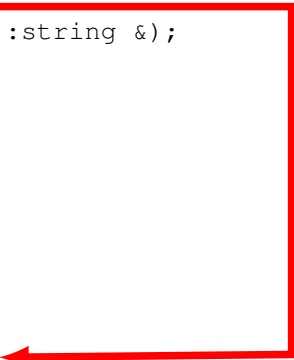
Instead of working with classes (and their objects) alone, we can work with a hierarchy of classes. Why do we want to do this? To improve “re-use”.

For instance: Suppose you have an `Employee` class and you want to create a `Manager` class. A manager **is an** employee – everything in the `Employee` class should be in the `Manager` class. For instance an `Employee` might have a last name, `Manager` should have a last name too.

You basically want the `Manager` class to have all the member variables and methods of the `Employee` class. Of course you can copy-and-paste the code from the `Employee` class into the `Manager` class:

```
class Employee
{
public:
    ...
    std::string get_firstname();
    void set_firstname(const std::string &);
private:
    std::string firstname;
    ...
};

class Manager
{
public:
    ...
    std::string get_firstname();
    void set_firstname(const std::string &);
    // other stuff not from Employee class
    ...
private:
    std::string firstname;
    // other stuff not from Employee class
    ...
};
```



Copy-and-paste??

Of course you would expect the `Manager` class to contain more things than the `Employee` class. For instance in the company where the above software is written, a manager might have a secretary while non-managers do not.

We don't like to duplicate code. If you need to modify the code from

the `Employee` class, then you also need to modify the same code in the `Manager` class. And if code in the `Employee` class is used in 10 other classes, the maintenance work on code would be horrific.

What should we do?

Inheritance

The solution is to use **inheritance**!!!

The general idea is very simple: Basically C++ allows you to define a class so that it will have all the “features” of another.

That’s all. BUT ... there is **a lot** of details beyond the general idea of inheritance. So **pay attention**.

Here is the basic syntax to get a class to inherit “features” from another class:

```
class Employee
{ ... };
class Manager: public Employee
{ ... };
```

This is called **public inheritance**, i.e., the `Manager` class **inherits** everything (i.e. member variables and methods) of the `Employee` class in the sense that the `Manager` class will have all the features (member variables and methods) from the `Employee` class.

Here is some basic terminology that you must know:

- `Employee` is the **parent class** or the **superclass** and `Manager` is the **child class** or the **subclass**.
- `Manager` is **derived** from `Employee` or `Manager` is a **derived subclass** of `Employee`.

I’m going to use the following very simple example to illustrate important principles when working with inheritance.

- The parent class `P` has the integer member `x_` together with `get_x` and `set_x` methods.
- The child class `C` has an integer member `y_` together with `get_y` and `set_y` methods.

I don’ t have to tell you ... run it.

```
#include <iostream>

class P
{
```

I’m putting the methods in one line so that we don’t have to page up and down.

```
public:
    P() : x_(1) {}
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C : public P
{
public:
    C() : y_(2) {}
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
                << "c.y_ =" << c.get_y() << '\n';
    return 0;
}
```

Notice that the `c` object of class `C` inherits `x_`, `get_x`, and `set_x` from `P`. That's why you can execute `c.get_x()` even though class `C` does not have a `get_x()` method.

Make sure you keep the above code handy since I'll be using this again and again in this set of notes. I'm going to call the above program our "Basic Example".

Exercise -1.0.1. Get rid of the inheritance clause in our Basic Example:

```
...
class P
{
    ...
};
class C : public P
{
    ...
};
```

and compile your code again. See the error? Now you cannot execute `c.get_x()`. READ the error message carefully. So we see that object `c` does seem to have the `get_x` method! (Put the `: public P` back into the code after you're done with this exercise.)

Exercise -1.0.2. Does `c` have the `set_x` method? How would you check that?

Exercise -1.0.3. Does `c` have the `x_` member variable? How would you check that? (Make `x_` public and print `c.x_` in `main()`.)

So `C` seems to have a “copy” of `P`'s declaration in it. Is that so? There are lots of other questions:

- What about the private section of `P`? Can `C` define methods to access the private members of `P`? If inheritance is a message to the compiler to “copy-and-paste” code from `P` to `C`, then surely methods in `C` can access the private members of `P`. Is that so?
- Also, what if `C` uses some features of `P` but not all – and in fact some features need to be modified. What can we do?
- What about constructor? Clearly `P : P()` constructor was called since in the above experiment `x_` is clearly 1. Can we call a parent constructor explicitly so that we can initialize the `x_`?
- What about object destruction?
- Etc., etc., etc.!!!

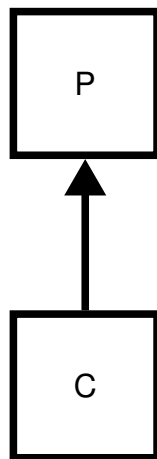
Exercise -1.0.4. The inheritance stack can be as tall as you like. Create a class `G` (“grandparent”) with integer member `z_` and with `get_z` and `set_z` method. Let the default constructor of `G` initialize `z_` to 42. Let class `P` inherit class `G`. In `main()`, print the value of `c.z_`.

Inheritance is not copy-and-paste of code!

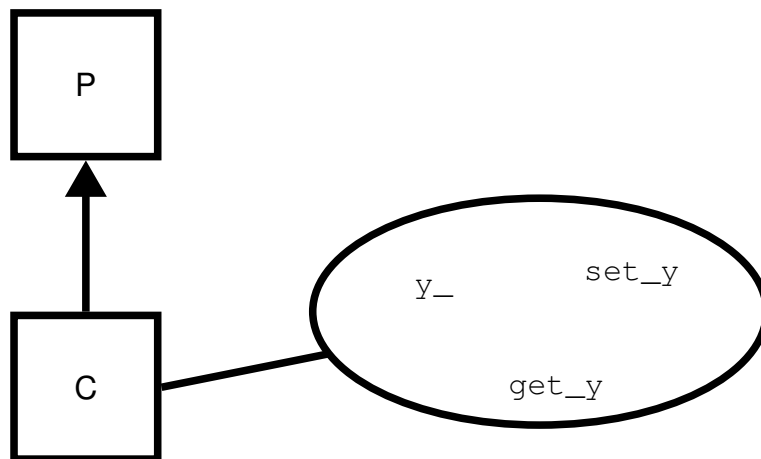
It's really important to understand that inheritance is not a copy-and-paste of code from the parent class to the child class. Inheritance is not like your “`#include`” business.

Here's how you should visualize objects created from a class that's a child class. Look at our Basic Example code again. We have a `c` object declared with class `C` which inherits class `P`. The object `c` has member `y_`, `get_y`, and `set_y` since the type of `c` is class `C`. From the experiments in the previous section, we see that `c` also has `gx_`, `get_x`, and `set_x` from its parent `P`.

Here's a picture to keep in mind. First here's a diagram of our classes:

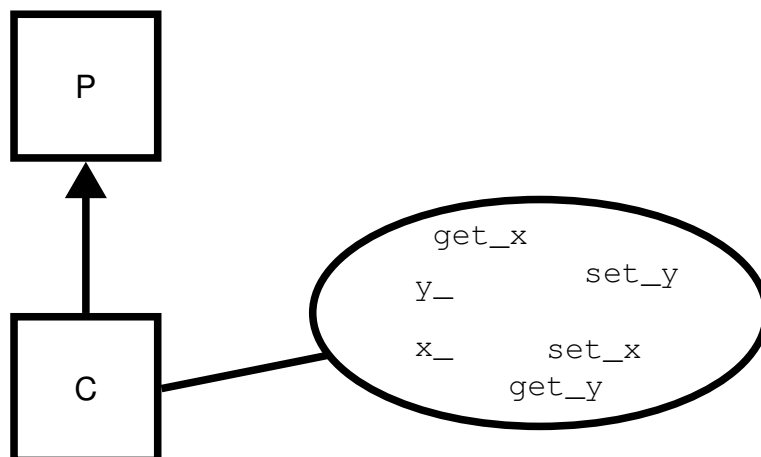


Each box represents the class. The arrow describes the inheritance relationship between the two classes. (This is the so-called class diagram from UML. I'm simplifying it a little – see previous notes for details on what to draw inside the boxes.) We created an object `c` from class `C`. So informally I'll draw it this way:



We also know that through inheritance, `c` also has `x_`, `get_x`, and `set_x`. So where's `c.x_`, `c.get_x`, and `c.set_x`?

Now if you think of `c.x_`, `c.get_x`, and `c.set_x` in the same bag of things like this:



then you would be surprised by the following experiment: Add an integer member variable to class `C`:

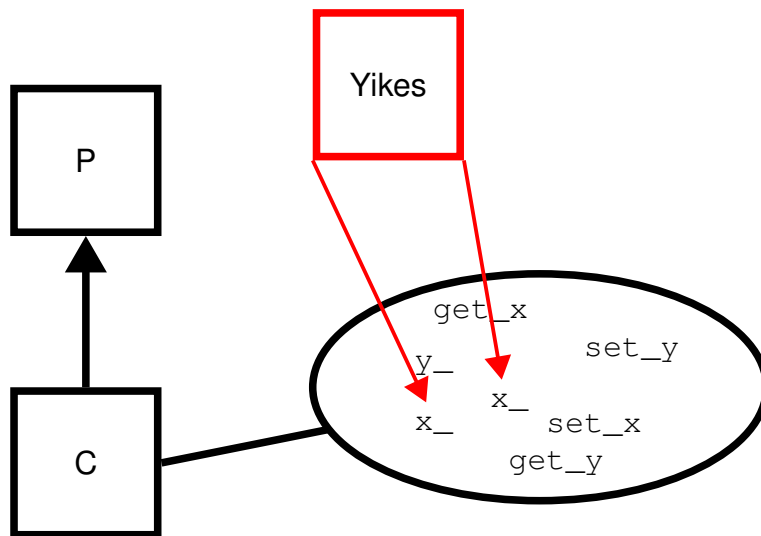
```
#include <iostream>

class P
{
public:
    P() : x_(1) {}
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C : public P
{
public:
    C() : y_(2) {}
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
               << "c.y_ =" << c.get_y() << '\n';
    return 0;
}
```

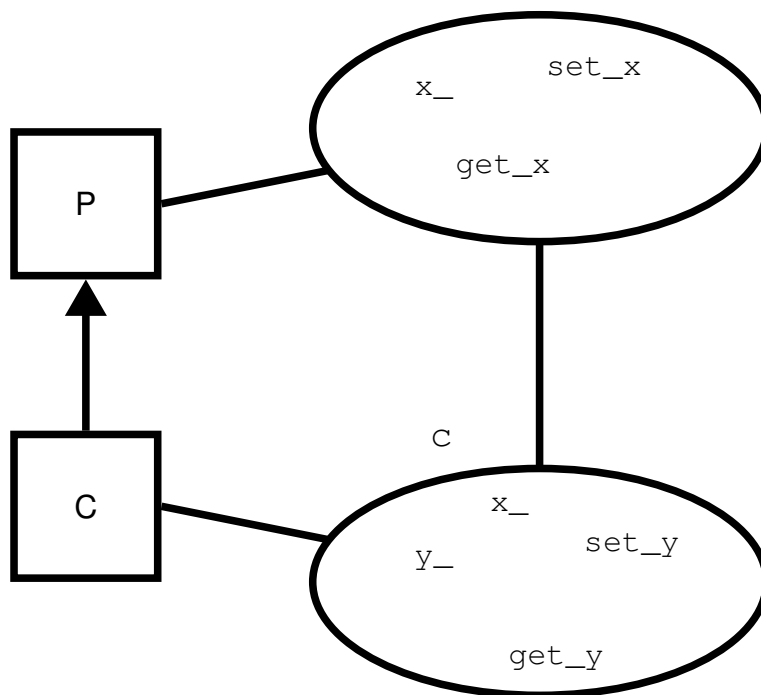
The program does compile. But this means that `c_` now has two `x_'s`!!!



When you call upon `c.x_` which one is it??? Why does the compiler allow this?

Exercise -1.0.5. Can you think of an experiment that can tell you which `x_` is used if you access `c.x_` in `main()`?

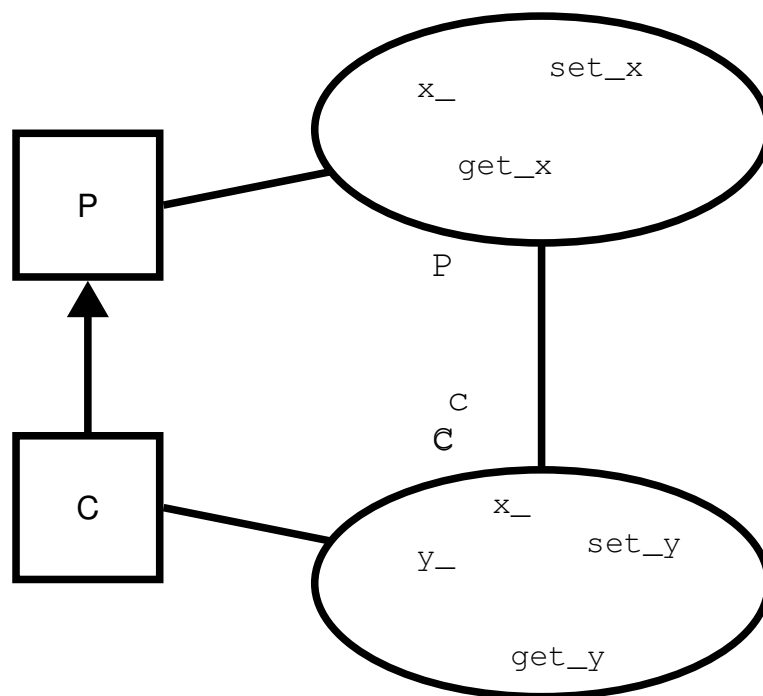
You really want to think of the members (member variables as well as methods) as being organized into two different scopes:



The members of `c` that are due to inheritance is at the top while the members declared in class `C` are at the bottom. In particular the `x_` declared in class `C` is at the bottom.

The name `c.x_` refers to the `x_` at the bottom, i.e., the `x_` due to the class `C`. Of course now you have a problem: what if in `main()`, you want to access the `x_` at the top???

In fact to be very precise, each member of `c` belongs to one of the two bags (bottom and top) and you can think of the two bags as scopes. You can in fact specify the scopes when you access a member of `c`. First of all the scopes have names: they are `C` and `P`, i.e., the names of the classes:



You can actually address a particular member by prepending it with `C::` or `P::` if you want to be specific. Try the following:

```
#include < iostream>

class P
{
public:
    P() : x_(1) {}
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C : public P
{
public:
    C() : y_(2) {}
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
               << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
               << "c.y_ =" << c.C::get_y() << '\n';
    return 0;
}
```

We also have two `x_`'s in `c`. Let's temporarily make them public and then access them:

```

#include < iostream>
class P
{
public:
    P() : x_(1) {}
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
public:
    int x_;
};

class C : public P
{
public:
    C() : y_(2) {}
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
public:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
               << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
               << "c.y_ =" << c.C::get_y() << '\n';
    std::cout << "c.x_ (in C) =" << c.P::x_ << ", "
               << "c.x_ (in P) =" << c.C::x_ << '\n';

    return 0;
}

```

NOTE: Since `c.C::x_` is not initialized, your C++ compiler will give you a warning. In that case, just do this (duh):

```
...  
  
class C : public P  
{  
public:  
    C() : ~EMPHASIZE@~redtext@x_(3),$$ y_(2) {}  
    ...  
~EMPHASIZE@~redtext@public:$$  
    int x_;  
    int y_;  
};  
  
...
```

Note that in `main()`, `c.x_` by default refers to the `x_` in `C` and not the `x_` of `P`. Sometimes we say that the `x_` in `C` **hides** the `x_` in `P`.

After the above experiments, change the “`public:`” back to “`private:`” and make the appropriate changes.

In general, here’s how your C++ compiler hunts down a name (member variable or method) or an object. Say your code has `c.x_`.

- The compiler starts with the class used to declare the object. If there’s an `x_` in that class, that’s the one used.
- If the class used to declare `c` does not have `x_`, your compiler will look at the parent class. If the parent has an `x_`, then that’s the one used.
- If the parent class does not have `x_`, and if the parent class inheritance from another class (that would be a grandparent class of the class of `c`), then the compiler will look for `x_` in the grandparent class of `C`.
- Etc.
- If it’s not found going up the public inheritance hierarchy, then your C++ compiler will yell at you.

If a class scope was specified, say you call upon `c.P::x_`, then the search for `x_` starts at class `P` proceeds upward through the inheritance hierarchy until it’s found.

Note that the search for a name goes **up** the inheritance hierarchy. Add the following to our Basic Example:


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

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

int main()
{
    ...
    c.a();
    return 0;
}
```

Exercise -1.0.6. What is the output? Or is there an error? The following is obtained by adding code to our Basic Example:

```
...

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

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

int main()
{
    ...
    c.c();
    return 0;
}
```

Private members of the parent class

For public inheritance, a child cannot access private member variables or private methods of parent. Let's do an experiment to verify this. Here's our Basic Example:

```
#include <iostream>

class P
{
public:
    P() : x_(1) {}
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C: public P
{
public:
    C() : x_(3), y_(2) {}
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
               << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
               << "c.y_ =" << c.C::get_y() << '\n';
    return 0;
}
```

Now let's create a method in C to access a private member of P:

```
...
class C : public P
{
public:
    ...
    void m() { P::x_ = 42; }
private:
    int x_;
    int y_;
};
...
```

You will get an error when you compile your program – make sure you read the error message. This is the same for private method in the parent:

```
...
class P
{
    ...
private:
    ~EMPHASIZE@~redtext@void m() {}$$
    int x_;
};

class C: public P
{
public:
    ...
    ~EMPHASIZE@~redtext@void n() { m(); }$$
    ...
};
...
```

Constructor

Now add some print statements in the parent and child constructor:

```
#include <iostream>

class P
{
public:
    P() : x_(1) { std::cout << "P::P()\n"; }
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C: public P
{
public:
    C() : x_(3), y_(2) { std::cout << "C::C()"; }
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
               << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
               << "c.y_ =" << c.C::get_y() << '\n';
    return 0;
}
```

Run it. You'll see that the parent constructor is indeed executed. This means that the child constructor actually called the parent's constructor.

Note that the **parent constructor** was **called before** the **child constructor**. Why is that? Because during child construction, you may use data already constructed in the parent. You cannot do the opposite.

Now what if you want to choose how to initialize the parent part of the `c` object? How do you explicitly call the parent's constructor? Let's add a new constructor in `P`, one that accepts an integer value and get the `C`'s constructor to call `P::P(int)`:

```
#include <iostream>

class P
{
public:
    P() : x_(1) { std::cout << "P::P()\n"; }
    P(int x) : x_(x) { std::cout << "P::P(int)\n"; }
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C: public P
{
public:
    C() : P(42), x_(3), y_(2)
        { std::cout << "C::C()"; }
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
               << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
               << "c.y_ =" << c.C::get_y() << '\n';
    return 0;
}
```

Call parent's
constructor
explicitly in the
initializer list

Exercise -1.0.7. You are given this incomplete code. Add to the constructor of `Car` and `Motorcycle` so that a `Car` object is initialized to have 4 wheels and a `Motorcycle` object is initialized to have 2 wheels.

```
class Vehicle
{
public:
    Vehicle(int numWheels)
        : numWheels_(numWheels)
    {}
    int numWheels() const { return numWheels_; }
private:
    int numWheels_;
};

class Car: public Vehicle
{
public:
    Car(bool keyless_entry)
        : keyless_entry_(keyless_entry)
    {}
private:
    bool keyless_entry_;
};

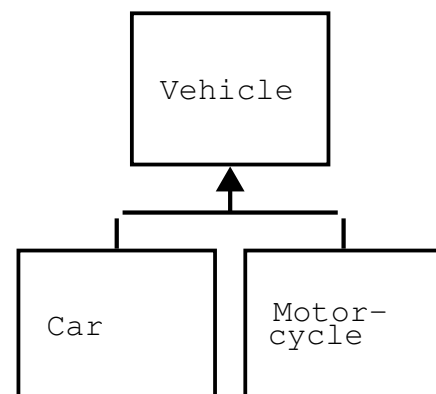
class Motorcycle: public Vehicle
{
public:
    Motorcycle()
        :
    {}
};

int main()
{
    Car aCar;           // aCar has 4 wheels
    Motorcycle aMotorcycle; // aMotorcycle has 2
                        // wheels

    std::cout << "aCar has "
               << aCar.numWheels() << " wheels\n";
    std::cout << "aMotorcycle has "
               << aMotorcycle.numWheels()
               << " wheels\n";

    return 0;
}
```

Exercise -1.0.8. This is an incomplete piece of code:



```
class GameObject
{
private:
    bool is_alive_;
    int x_;
    int y_;
};

class Laser: public GameObject
{
};

class Alien: public GameObject
{
};
```

Add code so that during the initialization of a `Laser` object, the `is_alive_` is set to `false` and during the initialization of an `Alien` object, the `is_alive_` is set to `true`.

Destructor

Now let's analyze what happens when a child object is destroyed. Of course the child object will call its destructor. What about the parent's destructor?

Run the following:

```
#include <iostream>

class P
{
public:
    P() : x_(1) { std::cout << "P::P()\n"; }
    ~P() { std::cout << "P::~~P()\n"; }
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C: public P
{
public:
    C() : x_(3), y_(2) { std::cout << "C::C()"; }
    ~C() { std::cout << "C::~~C()\n"; }
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
              << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
              << "c.y_ =" << c.C::get_y() << '\n';
    return 0;
}
```

Note that the **child destructor** was called **before** the **parent destructor**. So:

- Constructor: Parent's constructor called before child's constructor

- Destructor: Child's destructor called before parent's destructor

Assignment operator

If `c` is a `C` object and `p` is a `P` object, and `C` is derived from `P`, can we do the following:

```
c = p; ???  
p = c; ???
```

```

#include <iostream>

class P
{
public:
    P() : x_(1) { std::cout << "P::P()\n"; }
    P(int x) : x_(x) { std::cout << "P::P(int)\n"; }
    ~P() { std::cout << "P::~~P()\n"; }
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C: public P
{
public:
    C() : P(42), x_(3), y_(2)
    { std::cout << "C::C() "; }
    ~C() { std::cout << "C::~~C()\n"; }
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
              << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
              << "c.y_ =" << c.C::get_y() << '\n';

    P p;

    // Which one works?
    c = p;
    p = c;

    return 0;
}

```

So if `C` is derived from `P`, `c` is a `C` object and `p` is a `P` object, then `p = c;` is valid. Why?

Think about it. A `C` object has more “features”, so you can copy those features to a `P` object.

For instance suppose a child class object `c` has member variables

c.w, c.x, c.y, c.x and a parent class object p has member variables p.w, p.x, then when you perform

```
p = c;
```

this would happen:

```
p.w = c.w;  
p.x = c.x;
```

But if you attempt to do

```
c = p;
```

then it's not clear what should be done for the last two assignments:

```
c.w = p.w;  
c.x = p.x;  
c.y = ????  
c.z = ????
```

Class Reuse: “has-a” and “is-a”

Do not confuse inheritance and composition. They re-use old classes in different ways. Suppose `P` is a class.

There are two ways to reuse `P`:

- Using inheritance: `C` inherits from `P`
- Using composition: `C` contains a member that is of type `P`

Look at the following example carefully. Make sure you understand what is happening.

```
#include <iostream>

class P
{
public:
    P() : x_(1) { std::cout << "P::P()\n"; }
    P(int x) : x_(x) { std::cout << "P::P(int)\n"; }
    ~P() { std::cout << "P::~~P()\n"; }
    int get_x() const { return x_; }
    void set_x(int x) { x_ = x; }
private:
    int x_;
};

class C: public P
{
public:
    C() : P(42), x_(3), y_(2)
    { std::cout << "C::C()"; }
    ~C() { std::cout << "C::~~C()\n"; }
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    int x_;
    int y_;
};
```

```

...
class D
{
public:
    D() : p_(42), x_(3), y_(2)
    { std::cout << "D::D()"; }
    D() { std::cout << "D:: D()\n"; }
    int get_y() const { return y_; }
    void set_y(int y) { y_ = y; }
private:
    P p_;
    int x_;
    int y_;
};

```

D has a P object



```

int main()
{
    C c;
    std::cout << "c.x_ =" << c.get_x() << ", "
              << "c.y_ =" << c.get_y() << '\n';
    std::cout << "c.x_ =" << c.P::get_x() << ", "
              << "c.y_ =" << c.C::get_y() << '\n';

    P p;
    p = c;

    !textbfD d;

    return 0;
}

```

Study the above carefully. The `c` object has its own `x_` and `y_` and the `get_y` and `set_y` methods. It also has (through inheritance) its `P::x_` and method `P::get_x` and `P::set_x`. The object `d` also has `x_` and `y_` and the `get_x` and `set_x` and it also has `p_.x_` and `p_.get_x` and `p_.set_x`. In some sense both `c` and `d` have the same “features”.

`c` has `P`’s features by inheritance while `d` has `P`’s features through composition.

If what you want in a class can be found in another class, should you use composition or inheritance?

Look at “is-a” and “has-a” relationships between concepts.

Suppose you have a `Vehicle` class. You want to build a `Car` class. Think about the relationship between the concept of a car and a vehicle. Which one sounds right:

- A car “is a” vehicle or a vehicle “is a” car
- A car “has a” vehicle or a vehicle “has a” car

Clearly this one sounds better.

- A car “is a” vehicle

This tells you that you should probably use inheritance:

```
class Vehicle {...};  
class Car: public Vehicle {...};
```

Suppose you have an `Engine` class. You want to build a `Car` class. What is the relationship between an engine and a car? Which one sounds right:

- An engine “is a” car or an engine “is a” car
- An engine car “has a” vehicle or a vehicle “has an” engine

Clearly this is the best:

- A vehicle “has an” engine

This tells you that you should probably use composition:

```
class Engine {...};  
class Car {  
...  
private:  
    Engine anEngine;  
    ...  
};
```

For composition, you want to create methods for methods in the instance members (delegation):


```
class Engine {
public:
    void start();
};

class Car {
public:
    void start() { anEngine.start(); }
private:
    Engine anEngine;
};
```

Users can then do this:

```
Car honda_civic;
honda_civic.start();
// i.e., honda_civic.start() actually
// executes honda_civic.anEngine.start()
// i.e., honda_civic delegates the work of
// "start" to it's engine
```

Otherwise you would need this:

```
class Engine {
public:
    void start();
};

class Car {
public:
    Engine & get_engine() { return anEngine; }
private:
    Engine anEngine;
};
```

And users would have to do this:

```
Car honda_civic;
honda_civic.get_engine().start();
```

This is **not** so good. Why? Because you want to **hide lower level details (engines)** and let users of your `Car` class focus on higher level details (cars) which in this case is to “start a car” and not “start the engine of a car”.

In terms of re-use of code:

- Derivation/inheritance is a form of white-box re-use
- Composition is a form of black-box re-use

So in summary make sure you remember this:

- “is a” – inheritance
- “has a” – composition

Exercise -1.0.9. You want to write the Geometry Wars 2D game. You want different shapes with different colors. To get a feel for the problem, you want to focus on circles and squares. Circles have colors. Squares have colors. In fact ... every shape has a color. Therefore create a `Shape` class containing color, i.e., R, G, B (`int` instance variables). A circle has a radius. (HINT: Not all shapes have radius – squares do not have radii. The location of a circle is determined by the center of the circle. A square has a width and height which are the same in value. The location of the square is determined by the coordinates of the top-left corner. A rectangle is similar to the square except that the width and height need not be the same. Write relevant classes so that if you execute this:

```
// white circle at (2,5) of radius 3
Circle circle(2, 5, 3, 255, 255, 255);

// red square with top-left corner at (6, 7) with
// sides of length 8.
Square square(6, 7, 8, 255, 0, 0);

// green rectangle with top-left corner at (9, 10)
// with width 11 and height 12.
Rect rect(9, 10, 11, 12, 0, 255, 0);
```

Type conversion

Add the following to our Basic Example:

```
#include <iostream>

class P
{
    ...
};

class C: public P
{
    ...
};

int main()
{
    C c;
    ...
    P p;
    ...

    C c0(p); // This does not work!!!
    P p0(c); // This works

    return 0;
}
```

This tells you that if `C` is a subclass of `P`, then automatically, `P` has a constructor of the form

```
class P
{
    ...
    P(const C &);
    ...
};
```

The default behavior of this constructor is to copy relevant values from a `C` object to the `P` object that is being initialized.

This acts as a type conversion operator.

The idea is very similar to the assignment operator between objects of `C` and objects of `P`. And the reason that

```
C c0(p);
```

does NOT work is similar.

Multiple inheritance

A class can be the child class of more than one parent class.

Suppose we have a `Vehicle` class and a `Ship` class. Then `AmphibiousVehicle` is a child of both `Vehicle` and `Ship` class.

Syntax for multiple inheritance using public inheritance is simply:

```
class P0 {};
class P1 {};
class C: public P0, public P1 {};
```

Some programming languages do not allow multiple inheritance. Why?

Well, suppose class `C0` is derived from `P0`, `P1` and both `P0`, `P1` have `f()`. If `c` is a `C` object, then `c.f()` is ambiguous. **MAKE SURE** you run the following:

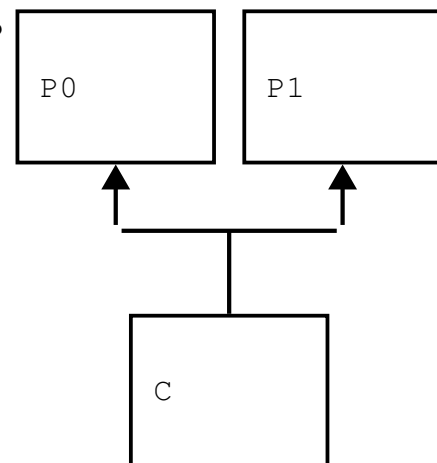
```
#include <iostream>

class P0
{
public:
    void f() {}
};

class P1
{
public:
    void f() {}
};

class C: public P0, public P1
{};

int main()
{
    C c;
    c.f(); // which one!!!!
    return 0;
}
```



Your C++ compiler will probably yell at you and say you have an ambiguous invocation. This is of course the same for ambiguous mem-

ber variables:

```
#include <iostream>

class P0
{
public:
    int x_;
};

class P1
{
public:
    int x_;
};

class C: public P0, public P1
{};

int main()
{
    C c;
    c.x_; // which one!?!?!
    return 0;
}
```

However note that if you have two members in both parents with the same name but you never call upon that name, your C++ will **not** complain. For instance

```
#include <iostream>

class P0
{
public:
    int x_;
};

class P1
{
public:
    int x_;
};

class C: public P0, public P1
{};

int main()
{
    C c;
    return 0;
}
```

will compile. If `P0` has a method that works with `P0::x_` and `P1` has a method that works with `P1::x_`, that wouldn't be a problem:

```
#include <iostream>

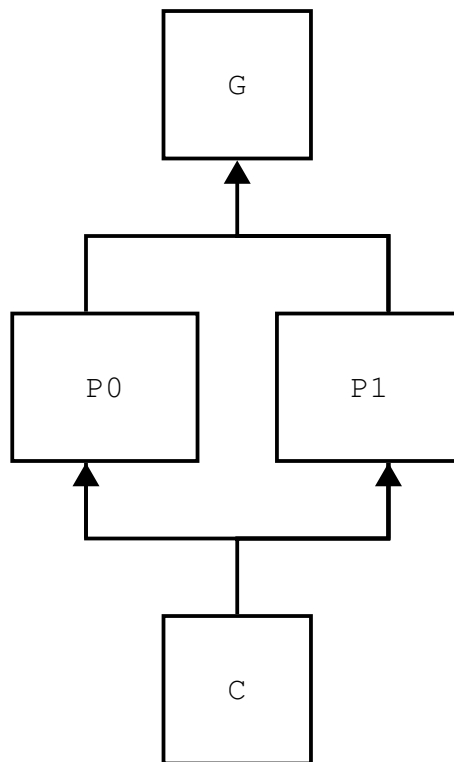
class P0
{
public:
    void f0() { std::cout << x_ << '\n'; }
    int x_;
};

class P1
{
public:
    void f1() { std::cout << x_ << '\n'; }
    int x_;
};

class C: public P0, public P1
{};

int main()
{
    C c;
    c.f0();
    c.f1();
    return 0;
}
```

Another situation where multiple inheritance might cause a problem is when, further up, `P0` and `P1` inherits from a common class:



```
#include <iostream>

class G
{
};

class P0: public G
{
};

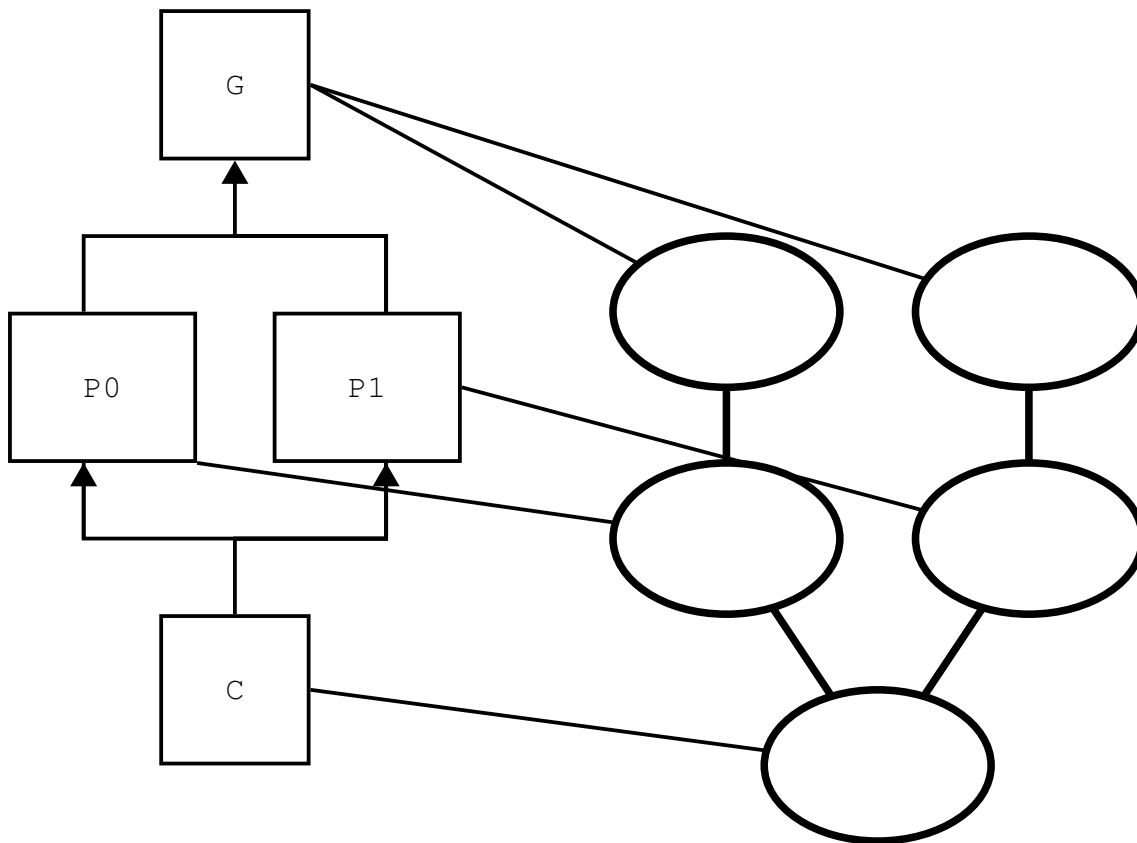
class P1: public G
{
};

class C: public P0, public P1
{
};

int main()
{
    C c;
    return 0;
}
```

The important thing to realize is that the object constructions starting with `C` is done one class at a time going up – there are **two paths** in this case. The two paths don't really “synchronize” with

each other: the creation of objects along the two inheritance paths is independent. This means that there are **two** objects created with class G for `c!!!`



To verify, first do this:

```
#include <iostream>

class G
{
public:
    int x_;
};

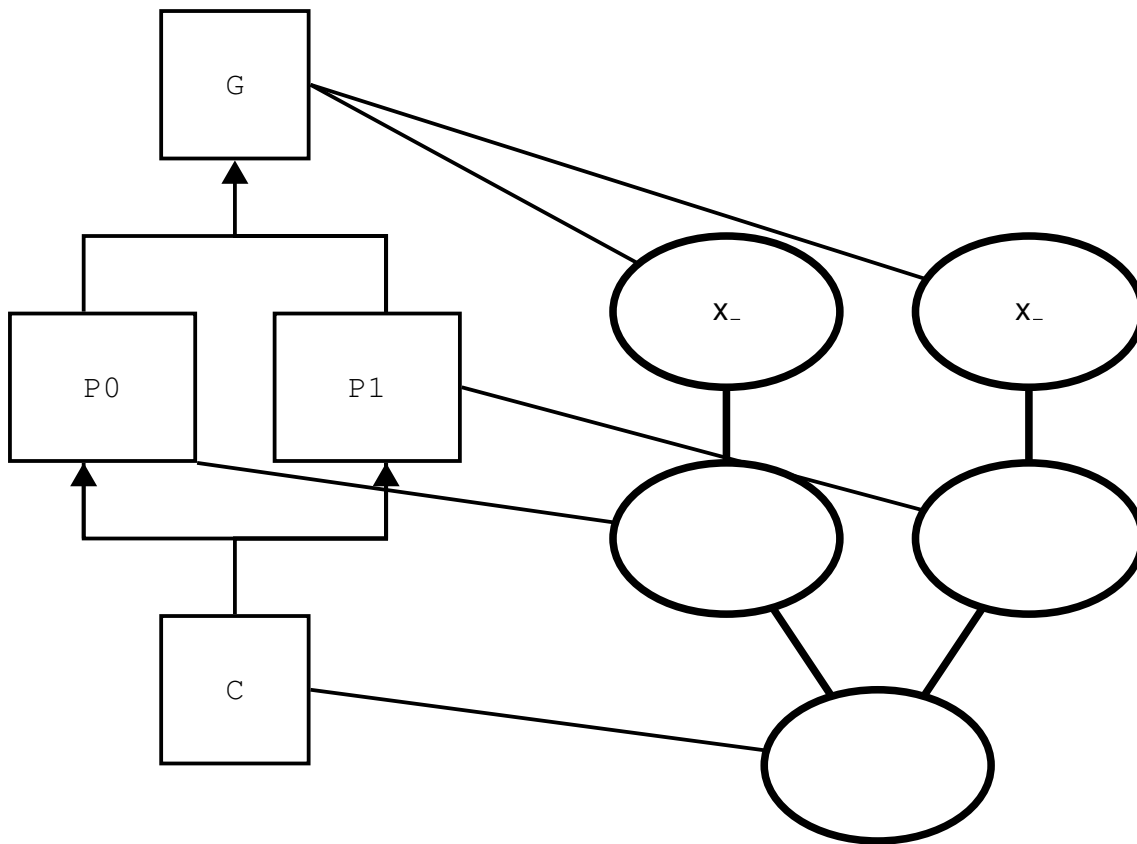
class P0: public G
{};

class P1: public G
{};

class C: public P0, public P1
{};

int main()
{
    C c;
    std::cout << &(c.P0::x_) << '\n';
    std::cout << &(c.P1::x_) << '\n';
    return 0;
}
```

The object `c` has two `x_`'s. We print the address of the two `x_`'s, which of course are different:



```
#include <iostream>

class G
{
public:
    int x_;
};

class P0: public G
{};

class P1: public G
{};

class C: public P0, public P1
{};

int main()
{
    C c;
    std::cout << &(c.P0::x_) << '\n';
    std::cout << &(c.P1::x_) << '\n';
    return 0;
}
```

Suppose you have a game of shapes. The Shape class describes (with its subclasses) how to draw the shape. You have another class PointMass that computes how the shapes moves as a physical point mass. You can have:

```
class PhysicalShape: public Shape, PointMass
{...};
```

This separates out the (motion) physics of object as a point mass from the drawing of the object.

- If there's a draw method: it's probably from the Shape class.
- If there's a move method: it's probably from the PointMass class

Public inheritance: protected member from the parent class

Recall that our child class cannot access private members in the parent class. (Remember that we're doing public inheritance.)

A **protected member** (either a member variable or method) is accessible by a subclass. Run the following:

```
class P
{
public: int x;
protected: int y_;
private: int z_;
};

class C: public P
{
public:
void f() { y_++; } // OK
};

int main()
{
    C c;
    c.y_++; // WRONG!!! Will not compile!!!
    return 0;
}
```

Public, protected, and private

So far you have seen public inheritance:

```
class P
{ ... };
class C: public P
{ ... };
```

There are two other types of inheritance: **protected and private inheritance**. Suppose you have a class `P`. Now define the following child classes:

```
class CPublic: public P {};
class CProtected: protected P {};
class CPrivate: private P {};
```

- The private members of `P` stay private in `CPublic`, `CProtected`, `CPrivate`.
- `CPublic`: public of `P` are public, protected of `P` are protected
- `CProtected`: public and protected of `P` are protected
- `CPrivate`: public, protected, and private of `P` are private.

```
class P
{
public: int x;
protected: int y_;
private: int z_;
};

class C: public P
{
public:
void f() { y_++; } // OK
};

int main()
{
    C c;
    c.x_++; // OK
    return 0;
}
```

```
class P
{
public: int x_;
protected: int y_;
private: int z_;
};

class C: protected P
{
public:
    void f() { y_++; } // OK
};

int main()
{
    C c;
    c.x_++; // WRONG!!! Public x in P becomes
            // protected in C so that only subclass
            // can access x
    return 0;
}
```



```
class P
{
public: int x_;
protected: int y_;
private: int z_;
};

class C: protected P

{
public:

void f() { y_++; } // OK

};

class C0: public C

{
public:

void f() { x_++; } // OK. Public x becomes
                // protected in C which
                // is accessible in C0

};

int main()
{
    return 0;
}
```

In summary, suppose `C` is a protected subclass of `P`, and `x_` is a public member of `P`.

- `x_` is accessible in `C`. (Any method in `C` can access member `x_`)
- `x_` becomes protected in `C`. Functions outside of `C` cannot access `obj.x_` if `obj` is an object of `C`.

Note that `p` is an object of `P`, the `p.x_` is still public.

```
class P
{
public: int x_;
protected: int y_;
private: int z_;
};

class C: private P // public x_ in P becomes
                  // private in C.
{
public:
    void f() { y_++; } // OK
};

class C0: public C
{
public:
    void f() { x_++; } // WRONG!!! x_ is now private
};

int main()
{
    return 0;
}
```

You can have as many layers of inheritance as you like:

```
class GameObject { ... };

class Weapon: public GameObject { ... };
class Laser: public Weapon { ... };
class Rocket: public Weapon { ... };

class Spaceship: public GameObject { ... };
class AlienSoldierShip: public Spaceship { ... };
class AlienCommanderShip: public Spaceship { ... };
```

Obviously create a new class only when there are member variables and/or methods which cannot be combined.

For instance if the following

```
Class AlienWorth10Points {...};
class AlienWorth20Points {...};
```

have exactly the same member variables and methods except that when objects of these types are destroyed in a gameplay, the points

earned are different, then you should have combined the two as...

```
class Alien
{
...
    int points;
};
```

Template inheritance

Now I want to show you how to do template inheritance. Try the following exercise:

Exercise -1.0.10. Write a class `C< T >` to be a subclass of `P< T >`. The class `P< T >` has a member variable `x` and method `m()`. The class `C< T >` will have a member variable named `y` and method `n()`. In `C< T >::n()` and `P< T >::m()` try to access the `x` defined in `P< T >`. For simplicity make all members public.

Here's the obvious first attempt at the above exercise. Run it.

```

#include <iostream>

template < typename T >
class P
{
public:
    void m()
    {
        x = 42;
        std::cout << "P::m() ... " << x << '\n';
    }
    T x;
};

template < typename T >
class C: public P< T >
{
public:
    void n()
    {
        std::cout << "C::n() ... \n";
        x = 43;
        std::cout << x << '\n';
        std::cout << y << '\n';
    }
    T y;
};

int main()
{
    C< int > c;
    c.n();
    c.m();
    return 0;
}

```

But there's an error! Here's the error message (if you are using g++):

```

main.cpp: In member function 'void C< T>::n()':
main.cpp:22:9: error: 'x'{} was not declared in this
scope
22 |         x = 43;
   |         ^

```

Written this way, `C< T >` is not able to access the `x` member variable of `P< T >`. Try this:

```
// ... as above ...
template < typename T >
class C: public P< T >
{
public:
    void n()
    {
        std::cout << "C::n() ... \n";
        this->x = 43;
        std::cout << this->x << '\n';
        std::cout << y << '\n';
    }
    T y;
};

// ... as above ...
```