Inheritance

Inheritance is a way to form new classes using classes that have already been identified.

> Inheritance is the basis of all objected oriented programming.

Let's consider the **Robot** class:

```
class Robot
{
    public:
        void setX(int newX);
        int getX();
        void setY(int newY);
        int getY();
    private:
        int m_x, m_y;
};
```

Now let's consider the **Shielded Robot** class:

```
class ShieldedRobot
{
    public:
        void setX(int newX);
        int getX();
        void setY(int newY);
        int getY();
        int getShield();
        void setShield(int s);
    private:
        int m_x, m_y, m_shield;
};
```

Notice the similarities. Both classes have x and y coordinates and provide the same set of methods to get and set the values of x and y.

A Shielded Robot *is a kind of* Robot!
It shares all of the same methods and data as a Robot. It just has some additional methods/data.

C++ lets us define a new class and have it "inherit" all of the methods/data of an existing, related class.

Inheritance is a technique that enables us to define a "subclass" (Shielded Robot) and have it "inherit" all of the functions and data of a "superclass".

> ## BTW, a subclass cannot touch the private member variables of its superclass!!!!

First, you define the superclass and implement all of its member functions.
Then, you define your subclass, explicitly basing it on the superclass.
Finally, you add *new* variables and member functions as needed.

So, assuming that class Robot's interface stays the same, with this implementation:

```
//base class
class Robot
{
     public:
        void setX(int newX)
        {
            m_x = newX;
        }
        int getX();
        {
            return m_x;
        }
        void setY(int newY);
        {
            m_y = newY;
        }
        int getY();
        {
            return m_y;
```

```
        }
    private:
        int m_x, m_y;
};
```

class Shielded Robot now looks like this, without repeating class Robot's member variables and methods:

```
//derived class
class Shielded Robot : public Robot
{
    public:
        //ShieldedRobot can do everything a Robot does, plus:
        int getShield()
        {
            return m_shield;
        }
        void setShield(int s)
        {
            m_shield = s;
        }
    private:
        //a Shielded Robot has x and y, plus
        int m_shield;
};
```

In C++, you can inherit more than once:

```
//base class
class Person
{
    public:
        string getName(void);
        ...
```

```
    private:
        string m_sName;
        int m_nAge;
};
```

A **Student** is a kind of **Person**.

```
//derived class of Person and base class of CompSciStudent
class Student : public Person
{
    public:
        //new stuff:
        int GetStudentID();
        ...
    private:
        //new stuffL
        int m_studentID;
        ...
};
```

A **CompSciStudent** is a kind of **Student**.

```
//derived class
class CompSciStudent : public Student
{
    public:
        //new stuff:
        void saySomethingSmart();
    private:
        //new stuff:
        string m_smartIdea;
};
```

A CompSciStudent object can say smart things, has a student ID, and a name!

Three Uses of Inheritance:

- Reuse
- Extension
- Specialization

Reuse

Every **public** method in the base class is automatically reused/exposed in the derived class.
They may be used normally by the rest of your program, and your derived class can use them too!

```
//base class
class Person
{
    public:
        string getName()
        {
            return m_name;
        }
        void goToBathroom()
        {
            cout << "splat!" << endl;
        }
};
```

```
//derived class
class Whiner : public Person
{
    public:
        void complain()
        {
            cout << "I hate homework!";
        }
};
```

```
int main()
{
    Whiner joe;
    joe.goToBathroom();
    joe.complain();
}
```

Program prints:
splat!
I hate homework!

```
//base class
class Robot
{
    public:
        Robot(void);
        int getX();
        int getY();
    private:
        void chargeBattery();
    private:
        int m_x, m_y;
};
```

```
//derived class
class ShieldedRobot : public Robot
{
    public:
        ShieldedRobot(void)
        {
```

```
            m_shield = 1;
            chargeBattery(); //THIS IS ILLEGAL!!!!
        }
        int getShield();
    private:
        int m_shield;
};
```

The above program will not work because only public members can be exposed/reused in the derived classes. Private members in the base class are hidden from the derived classes.

If you would like your derived class to be able to reuse one or more private member functions, but you don't want the rest of your program to use them, make them `protected` instead of private in the base class. This lets your derived class (and its derived classes) reuse these member functions from the base class.

```
//base class
class Robot
{
    public:
        Robot(void);
        int getX();
        int getY();
    protected:
        void chargeBattery();
    private:
        int m_x, m_y; //never make your member variables protected or public
};
```

```
//derived class
class ShieldedRobot : public Robot
{
    public:
```

```
        ShieldedRobot(void)
        {
            m_shield = 1;
            chargeBattery(); //THIS IS OK NOW!!!!
        }
        int getShield();
    private:
        int m_shield;
};
```

```
int main()
{
    ShieldedRobot stan;
    stan.chargeBattery(); //still fails!
}
```

Reuse Summary

If I define a `public` function in a base class B:

- Any function in class B may access it.
- Any function in all classes derived from B may access it.
- All classes/functions related to B may access it.

If I define a `private` member variable/function in a base class B:

- Any function in class B may access it.
- No functions in classes derived from B may access it.
- No classes/functions unrelated to B may access it.

If I define a `protected` member function in a base class B:

- Any function in class B may access it.
- Any functions in all classes derived from B may access it.
- No classes/functions unrelated to B may access it.

Extension

Extension is when you **add new behaviors** (member functions) **or data** to a derived class that were not present in a base class.

```
//base class
class Person
{
    public:
        string getName()
        {
            return m_name;
        }
        void goToBathroom()
        {
            cout << "Splat!" << endl;
        }
};
```

```
//derived class
class Whiner : public Person
{
    public:
        void complain() //new method
        {
            cout << "I hate " << whatIHate;
        }
    private:
        string whatIHate;
};
```

```
int main()
{
    Whiner joe;
```

```
    joe.complain();
}
```

However, while all public extensions may be used normally by the rest of your program, they're unknown to your base class!

> Your base class only knows about itself - it knows nothing about classes derived from it!

```
//base class
class Person
{
    public:
        string getName()
        {
            return m_name;
        }
        void goToBathroom()
        {
            if(iAmConstipated)
                complain(); //ERROR
        }
};
//derived class
class Whiner : public Person
{
    public:
        void complain() //new method
        {
            cout << "I hate " << whatIHate;
        }
    private:
```

```
        string whatIHate;
};
```

class Person cannot access class Whiner's `complain()` method.

Specialization

Specialization is when you **redefine an existing behavior** (from the base class) with a new behavior (in a derived class).
In addition to **adding entirely new functions and variables** to a derived class, you can also **override or specialize existing functions** from the base class in your derived class.

> If you override existing functions from the base class in your derived class, you should **always** insert the `virtual` keyword in front of **both** the original and replacement functions.

```
class Student
{
    public:
        virtual void WhatDoISay()
        {
            cout << "Go bruins!" << endl;
        }
        ...
};
```

```
class NerdyStudent : public Student
{
    public:
        virtual void WhatDoISay()
        {
            cout << "I love circuits!" << endl;
        }
        ...
```

```
};
```

```
int main()
{
    Student carey; //lmao when the CS profs are so extra
    NerdyStudent davidS;

    carey.WhatDoISay();
    davidS.WhatDoISay();
}
```

Output:
Go bruins!
I love circuits!

> If you define your member functions OUTSIDE your class, you must only use the `virtual` keyword within your class definition.

Use `virtual` here within your class definition:

```
class Student
{
    public:
        virtual void WhatDoISay();
        ...
};
class NerdyStudent : public Student
{
    public:
        virtual void WhatDoIsay();
        ...
};
```

Don't write `virtual` here:

```
void Student::WhatDoISay()
{
    cout << "hello!";
}
void NerdyStudent::WhatDoISay()
{
    cout << "I love circuits!";
}
```

You only want to use the `virtual` keyword for functions you intend to override in your subclasses. Example:

```
//base class
class Robot
{
     public:
        void setX(int newX) { m_x = newX; }
        int getX() { return m_x; }
        virtual void talk()
        {
            cout << "Buzz. Click. Beep." << endl;
        }
    private:
        int m_x, m_y;
};
class ComedianRobot : public Robot
{
    public:
        //inherits getX() and getY()
        virtual void talk()
        {
            cout << "Two robots walk into a bar..." << endl;
```

```
        }
    private:
        ...
};
```

If you redefine a function in the derived class, then the redefined version hides the base version of your function (but only when using your derived class :))

```
int main()
{
    Robot r;
    ComedianRobot c;
    r.talk();
    c.talk();
}
```

Output:
Buzz. Click. Beep.
Two robots walk into a bar...

Reuse of Hidden Base-class Methods

```
class Student
{
    public:
        virtual void cheer()
        {
            cout << "Go bruins!" << endl;
        }
        void goToBathroom()
        {
            cout << "Splat!" << endl;
        }
    ...
};
```

```
class NerdyStudent : public Student
{
    public:
        virtual void cheer()
        {
            cout << "go algorithms!" << endl;
        }
        void getExcitedAboutCS()
        {
            Student::cheer();
        }
}
```

> Your derived class will, by default, always use the most derived version of a specialized method.

If you want to call the base class's version of a method that's been redefined in the derived class, you can do so by using the `baseclass::method()` syntax.

Sometimes a method in your derived class will want to rely upon the overriden version in the base class:

```
class Student
{
    public:
        Student()
        {
            myFavorite = "alcohol";
        }
        virtual string WhatILike()
        {
            return myFavorite;
        }
```

```
    private:
        string myFavorite;
};
```

```
class NerdyStudent : public Student
{
    public:
        virtual string whatILike()
        {
            string fav = Student::whatILike(); //First you call the base-version of the method
            fav += " bunsen burners"; //Then you modify any result you get back and return it
            return fav;
        }
};
```

Inheritance and Construction

> C++ always constructs the base class first and the derived part second.

```
//base class
class Robot
{
    public:
        Robot(void)
        //call m_bat's constructor
        {
            m_x = m_y = 0;
        }
        ...
    private:
```

```
        int m_x, m_y;
        Battery m_bat;
};
```

```
//subclass
class ShieldedRobot : public Robot
{
    public:
        ShieldedRobot(void)
        //call Robot's constructor
        //call m_sg's constructor
        {
            m_shieldStrength = 1;
        }
    private:
        int m_shieldStrength;
        ShieldGenerator m_sg;
};
```

Any time you define a derived object, C++ first implicitly calls your base constructor, and then it constructs your derived object's member variables. Lastly, it runs the body of your derived constructor.

Inheritance and Destruction

> C++ destructs the derived part first, then the base part second.

```
//base class
class Robot
{
    public:
        ~Robot();
```

```
        {
            m_bat.discharge();
        }
        //call m_bat's destructor
        ...
    private:
        int m_x, m_y;
        Battery m_bat;
};
```

```
//subclass
class ShieldedRobot : public Robot
{
    public:
        ~ShieldedRobot()
        {
            m_sg.turnGeneratorOff();
        }
        //call m_sg's destructor
        //call Robot's destructor
        ...
    private:
        int m_shieldStrength;
        ShieldGenerator m_sg;
};
```

Review of virtual functions:

```
class Car
{
    virtual void accelerate()
```

```
    {
        useFuel();
    }
    virtual void useFuel()
    {
        cout << "use fuel";
    }
}
class Batmobile : public Car
{
    public:
        virtual void accelerate()
        {
            cout << "Vroom";
            Car::accelerate();
        }
        virtual void useFuel()
        {
            cout << "Bat fuel";
        }
}
```

```
Car c;
c.accelerate(); //will print out "use fuel"
Batmobile b;
b.accelerate(); //will print out "Vroom" and then "Bat fuel"
//when we call car's accelerate, the class will find the most derived version of useFuel() that
 it knows about, hence it calls Batmobile's useFuel()
```

a class will always use the most derived virtual function that it knows about

```
class Car
{
    virtual void accelerate()
    {
        Car::useFuel(); //now no matter what the useFuel() function looks like in the Batmobile
                        //class, Car's accelerate() method will call Car's useFuel() method
    }
    virtual void useFuel()
    {
        cout << "use fuel";
    }
}
class Batmobile : public Car
{
    public:
        virtual void accelerate()
        {
            cout << "Vroom";
            Car::accelerate();
        }
        virtual void useFuel()
        {
            cout << "Bat fuel";
        }
}
```

```
class SuperBM : public Batmobile
{
    virtual void useFuel();
}
Batmobile b;
```

```
b.accelerate();

//will print out "Vroom" and then "Bat fuel" even if SuperBM defines a new useFuel() method

//Batmobile doesn't know about SuperBM
```

Initializer Lists

A subclass cannot touch the privates of its base class, so it must use an initializer list. You have to use an initializer list if the nearest base class requires parameters.

```
class Animal //base class

{

    public:

        Animal(int lbs)

        {

            m_lbs = lbs;

        }

        void what_do_i_weigh(void)

        {

            cout << m_lbs << " lbs!\n";

        }

    private:

        int m_lbs;

};


class Duck : public Animal

{

    public:

        Duck(int lbs) : Animal(lbs)

        {

            m_feathers = 99;

        }

        void who_am_i()

        {
```

```
            cout << "A duck!";
        }
    private:
        int m_feathers;
};
```

Note: You won't be tested on assignment operators.

Inheritance Review:

- Reuse
  - Write code once in a base class and reuse the same code
- Extension
  - Add new behaviors that were not present in a base class
- Specialization
  - Redefine an existing behavior with a new behavior