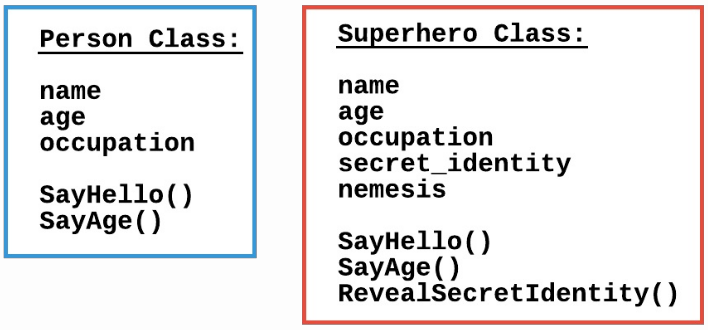
Learning Objectives: Base & Derived Classes

- **Define the terms inheritance, base class, and derived class**
- **Explain the relationship between the base class and the derived class**
- **Create a derived class from a given base class**
- **Understand how access modifiers work with inheritance types**
- **Compare base and derived classes**
- **Define Liskov's Substitution Principle**

What is Inheritance?

Defining Inheritance

Imagine you want to create two C++ classes, `Person` and `Superhero`. These respective classes might look something like this:



.guides/img/inheritance/CppNoInheritance

There are some similarities between the `Person` class and the `Superhero` class. If the `Person` class already exists, it would be helpful to "borrow" from the `Person` class so you only have to create the new attributes and functions for the `Superhero` class. This situation describes inheritance — one class copies the attributes and functions from another class.

Inheritance Syntax

In the IDE on the left, the `Person` class is already defined. To create the `Superhero` class that inherits from the `Person` class, add the following code at the end of the class definitions. Notice how the `Superhero` class definition contains a colon `:` followed by `public` and then `Person`. This is how you indicate to C++ that the `Superhero` class inherits from the `Person` class. You can also say that `Person` is the **base class** and `Superhero` is the **derived class**. A base class in C++ is also referred to as a **superclass** or **parent** class while a derived class is also referred to as a **subclass** or **child** class. All of these terms can be used interchangeably.

```
//add class definitions below this line

class Superhero : public Person {

};

//add class definitions above this line
```

Now declare an instance of the `Superhero` class and print the value of the `name` and `age` attributes using their getter functions.

```
  //add code below this line

  Superhero s;
  cout << s.GetName() << endl;
  cout << s.GetAge() << endl;

  //add code above this line
```

▼ **What does the output of the program above mean?**

In C++, reducing memory usage is important and initializing values to variables requires memory. Uninitialized variables do not get assigned specified values automatically. Thus, when printing the value of uninitialized variables, you might get random and unexpected output. The output is considered to be **junk data** that are left over at the variables' memory location.

challenge

Try these variations:

- Call `SetName` on `s` with `"Peter Parker"` as the argument.
- Call `SetOccupation` on `s` with `"Student"` as the argument.
- Print and call `GetOccupation` on `s`.
- Call `SayHello` on `s`.

▼ **Solution**

```
//add code below this line

Superhero s;
s.SetName("Peter Parker");
s.SetOccupation("Student");
cout << s.GetOccupation() << endl;
s.SayHello();

//add code above this line
```

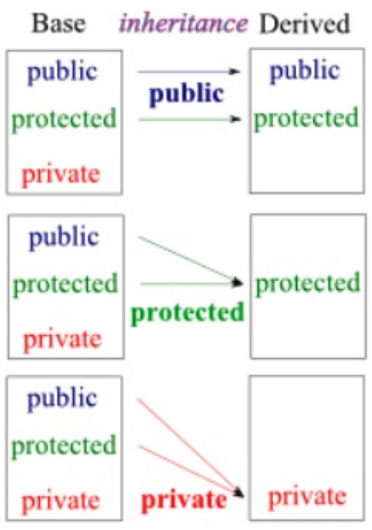
Accessibility of Inheritance

C++ places some rules about how inheritance works. Depending on the **access modifier** of the specified superclass, the subclass may or may not inherit certain class functions or attributes. Here is a list showcasing each access modifier and its effect on the subclass.

- “public inheritance makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.”
- “protected inheritance makes the public and protected members of the base class protected in the derived class.”
- “private inheritance makes the public and protected members of the base class private in the derived class.”

Source: https://www.programiz.com/cpp-programming/public-protected-private-inheritance

The list above can be represented using the chart below:



.guides/img/inheritance/InheritanceChart

Source: https://www.bogotobogo.com/cplusplus/private_inheritance.php

Notice how in each case, the derived class never inherits any private members from the base class. Public inheritance causes the derived class to inherit the members as is. Protected inheritance causes the derived class to inherit all public and protected members as protected only. And private inheritance causes all inherited members to be private only.

You can see how restrictive *protected* and *private* inheritance is, which is why they are rarely used. For this module, we will be using mainly *public* inheritance when creating derived classes.

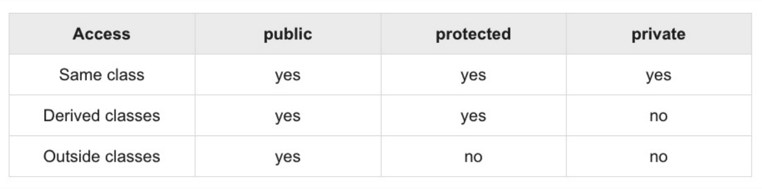
Effects of Inheritance Types 1

Access Modifiers Review

Before we can look at the effects of inheritance types, we will need to review some access modifier vocabulary:

- `private` - private members of a class can only be accessed by other members within the **same** class.
- `protected` - protected members of a class can be accessed by other members within the **same** class or by a **derived** class.
- `public` - public members of a class can be accessed by other members within the **same** class, by a **derived** class, or by an **external** or outside class.

The table below showcases these access modifiers' effects.



Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

.guides/img/inheritance/AccessModifierTable

Source: https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

Accessing Class Members

In the text editor, the `Base` class has already been defined. Within this base class, you'll see that there is a **public** function, a **protected** function, and a **private** function. Add the following derived class to your code:

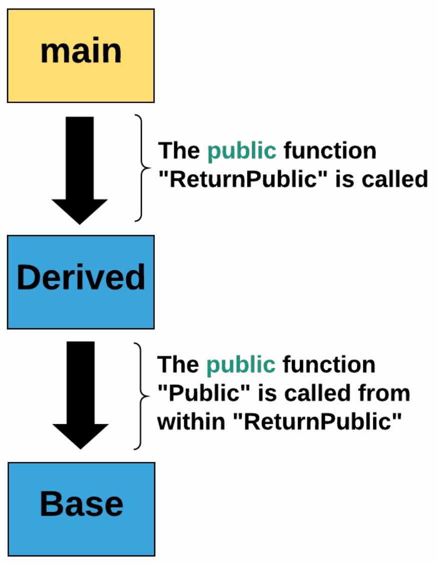
```
//add class definitions below this line

class Derived : public Base {
  public:
    void ReturnPublic(string s) {
      Public(s_derived); //public function inherited from Base
    }

  private:
    string s_derived;
};

//add class definitions above this line
```

The derived class `Derived` has one public function and one private attribute and it **publicly** inherits all public and protected members of the base class `Base`. This means that it can call and use any functions or attributes of `Base` with the exception of anything that is private.

Next, add the following code into the `main` function:

```
  //add code below this line

  string s_main;
  Derived dc;
  dc.ReturnPublic(s_main);

  //add code above this line
```

In `main`, a string `s_main` and a `Derived` object `dc` are created. Then `ReturnPublic()` is called on `dc` with `s_main` as the parameter. `ReturnPublic` is a `public` member function of the `Derived` class which takes a string as a parameter and calls the `Public` member function of the `Base` class on `s_derived`. The entire process sounds very complicated but can be explained visually in the flowchart below.


main
The public function "ReturnPublic" is called
Derived
The public function "Public" is called from within "ReturnPublic"
Base

.guides/img/inheritance/FlowChart1

The reason why `main` is able to call `ReturnPublic` is due to the fact that `ReturnPublic` is a public member function within `Derived`.

challenge

Try this variation:

- Revise `Derived` to look like:

```
class Derived : public Base {
  protected:
    void ReturnPublic(string s) {
      Public(s_derived);
    }

  private:
    string s_derived;
};
```

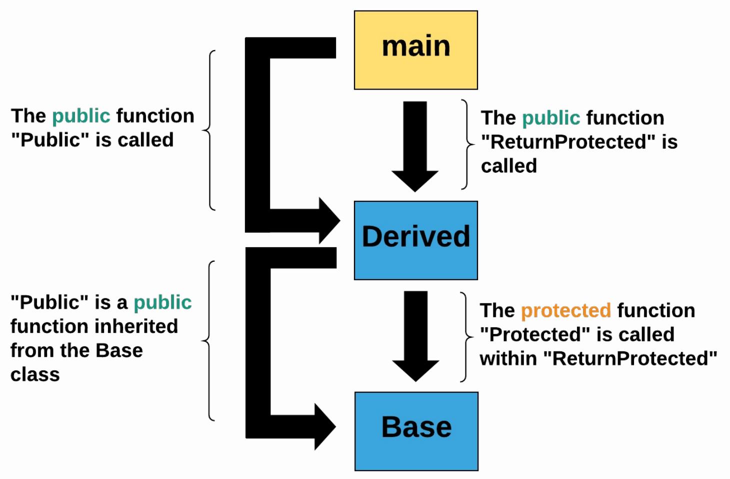
When the `ReturnPublic` member function of `Derived` is **protected**, `main` is no longer able to access it. Remember, external classes can only access **public** members of other classes, unless it is a derived class. Derived classes can access protected members in addition to public ones.

Next, let’s change `Derived` and `main` to look like this:

```
//add class definitions below this line

class Derived : public Base {
  public:
    void ReturnPublic(string s) {
      Public(s_derived); //public function inherited from Base
    }

    void ReturnProtected(string s) {
      Protected(s_derived); //protected function inherited from
        Base
    }

  private:
    string s_derived;
};

//add class definitions above this line
```

```
  //add code below this line

  string s_main;
  Derived dc;
  dc.ReturnProtected(s_main);
  dc.Public(s_main);

  //add code above this line
```

Notice how `main` can call `Public` and also `ReturnProtected` from `Derived` because those functions are public. Additionally, `Derived` can call the protected function `Protected` in `ReturnProtected` from `Base` because `Derived` is a derived class of `Base`.


main
The public function "Public" is called
The public function "ReturnProtected" is called
Derived
"Public" is a public function inherited from the Base class
The protected function "Protected" is called within "ReturnProtected"
Base

.guides/img/inheritance/FlowChart2

Effects of Inheritance Types 2

How Inheritance Works

Now that you're more comfortable with access modifiers, let's take a look at how inheritance types affect derived classes. Make sure your `Derived` and `main` code look like what's below. **Note** that `Derived` now inherits from `Base` through `protected` inheritance instead of `public`.

```
//add class definitions below this line

class Derived : protected Base {
  public:
    void ReturnPublic(string s) {
      Public(s_derived);
    }

    void ReturnProtected(string s) {
      Protected(s_derived);
    }

  private:
    string s_derived;
};

//add class definitions above this line
```

```
  //add code below this line

  string s_main;
  Derived dc;
  dc.ReturnProtected(s_main);
  dc.Public(s_main);

  //add code above this line
```

You'll notice that an error is produced saying that `'Base' is not an accessible base of 'Derived'`. This occurs because when a derived class inherits from a base class through **protected** inheritance, all public and protected members of the base class become `protected` in the derived class. This means that the `Public` function within `Derived` was inherited as a protected function and is therefore no longer accessible within `main`.

On the other hand, `ReturnProtected` is still a **public** function within `Derived`, which allows `main` to access it even though it calls the protected function `Protected` from `Base`. Comment out the command `dc.Public(s_main);` and run the code again to see the result.

Though not explicitly shown, `Derived` can be represented like this:

```
class Derived : protected Base {
  public:
    void ReturnPublic(string s) {
      Public(s_derived);
    }

    void ReturnProtected(string s) {
      Protected(s_derived);
    }

  /*protected:
    void Public(string s) { (inherited as protected from Base)
      s = "public";
      cout << s << endl;
    }

    void Protected(string s) { (inherited as protected from
        Base)
      s = "protected";
      cout << s << endl;
    }*/

  private:
    string s_derived;
};
```

If you ever get confused with inheritance types, you can always imaginatively rewrite all of the functions that were inherited and label them appropriately to help you visualize the derived class better. Here are some more examples:

```cpp
class Base {
  public:
    int x;

  protected:
    int y;

  private:
    int z;
};

class Derived : public Base {
  //public:
    //int x;

  //protected:
    //int y;
};
```

```cpp
class Base {
  public:
    int x;

  protected:
    int y;

  private:
    int z;
};

class Derived : protected Base {
  //protected:
    //int x;
    //int y;
};
```

```cpp
class Base {
  public:
    int x;

  protected:
    int y;

  private:
    int z;
};

class Derived : private Base {
  //private:
    //int x;
    //int y;
};
```

Note that **private** members are *never* inherited. Also, the examples are provided to help you visualize what the derived classes look like after inheritance. If you were to actually add the member attributes into `Derived` without comments, those attributes will count as being part of the `Derived` class instead of being **inherited** from the `Base` class.

Inheriting the Constructor

How are Constructors Inherited?

Unlike functions and attributes, the constructor that is inherited by the derived class needs to get linked or associated with that of the base class. To connect the derived constructor to the base constructor, follow this syntax:

- Derived class name
- Parentheses `()`
- Parameters with specified data types within parentheses
- Colon `:`
- Base class name
- Parentheses `()`
- Parameters of derived class as arguments in parentheses
- Any additional commands in curly braces `{}` or leave them empty

Sample Code:
`Constructor2(Param p1, Param p2) : Constructor1(p1, p2) {}`

Add the following code to the class definitions field:

```
//add class definitions below this line

class Superhero : public Person {
  public:
    Superhero(string n2, int a2) : Person(n2, a2) {}
};

//add class definitions above this line
```

And the following to `main`:

```
  //add code below this line

  Superhero s("Spider-Man", 16);
  s.ReturnPerson();

  //add code above this line
```

By associating the derived constructor with the base constructor, `C++` is able to pass the parameters specified in the derived constructor as arguments of the base constructor. In the code above, the arguments `Spider-Man` and `16` are passed to the `Superhero` constructor and then transferred over to the `Person` constructor where they get assigned to `name` and `age` respectively. Then, the `ReturnPerson` function is used to print `name` which is now `Spider-Man` and `age` which is now `16`.

Comparing Base & Derived Classes

Determining a Derived Class's Base Class

How do you determine if a derived class actually belongs to a base class? One common way to determine this is to use the `is_base_of<Base, Derived>::value` function. Just substitute `Base` with the name of the base class and `Derived` with the name of the derived class.

```
//add code below this line

cout << boolalpha;
cout << "B is derived from A: " << is_base_of<A, B>::value <<
      endl;
cout << "C is derived from B: " << is_base_of<B, C>::value <<
      endl;
cout << "A is derived from C: " << is_base_of<C, A>::value <<
      endl;

//add code above this line
```

Here is an example:

```
//add code below this line

cout << boolalpha;
cout << "Superhero is derived from Person: " <<
      is_base_of<Person, Superhero>::value << endl;
cout << "Animal is derived from Superhero: " <<
      is_base_of<Superhero, Animal>::value << endl;
cout << "Person is derived from Animal: " <<
      is_base_of<Animal, Person>::value << endl;

//add code above this line
```

challenge

Try this variation:

- Add the following to `main`:

```
cout << "Person is derived from Superhero: ";
cout << is_base_of<Superhero, Person>::value << endl;
```

You’ll notice that the function returns `true` if the derived class inherits from the base class and `false` when that is not the case.

Determining an Object’s Base Class

Unfortunately, C++ has no built-in function to determine if an object is from a class that inherits from another class. Instead, use `typeid(<object_name>).name()` to try and extract the object’s type, and then use the `is_base_of<Base, Derived>::value` to see if that object’s class is derived from another specified class. Replace `<object_name>` with the name of the object.

Remove all existing code in `main` and add the following:

```
//add code below this line

Superhero s;
cout << "s is of type: " << typeid(s).name() << endl;

//add code above this line
```

Your output may look something like `s is of type: 9Superhero`. The `9` is just a number that is produced by the compiler, which can be ignored. Once you determine the object’s class, you can then compare that class to another class to see if it is a derived class.

```
//add code below this line

Superhero s;
cout << "s is of type: " << typeid(s).name() << endl;
cout << boolalpha;
cout << "Superhero is derived from Person: ";
cout << is_base_of<Person, Superhero>::value << endl;

//add code above this line
```

The functions provided above can help you determine an object’s class and compare it with another **known** class. There is unfortunately no function to determine an object’s base class directly.

Substitution Principle

Substitution Principle

When one class inherits from another, C++ considers them to be related. They may contain different data types, but C++ allows a derived class to be used in place of the base class. This is called **Liskov's Substitution Principle**. In the text editor you'll notice that `Superhero` inherits from `Person`, but `Animal` does not. All classes have the `Greeting` function which prints a statement specific to the class.

```cpp
//add class definitions below this line

class Person {
  public:
    void Greeting() {
      cout << "I'm a Person" << endl;
    }
};

class Superhero : public Person {
  public:
    void Greeting() {
      cout << "I'm a Superhero" << endl;
    }
};

class Animal {
  public:
    void Greeting() {
      cout << "I'm an Animal" << endl;
    }
};

//add class definitions above this line
```

According to the Substitution Principle, an object of `Superhero` can be used in a situation that expects an object of `Person`. Add the `Substitution` function below which explicitly requires a parameter of a `Person` object.

```
//add function definitions below this line

void Substitution(Person p) {
  p.Greeting();
}

//add function definitions above this line
```

Instantiate an object of `Superhero` and pass it to the `Substitution` function. Even though the object `s` has the wrong data type, the code should still work due to the Substitution Principle. Because `Superhero` is derived from `Person`, object `s` can be used in place of an object of type `Person`. Run the code to verify the output.

```
  //add code below this line

  Superhero s;
  Substitution(s);

  //add code above this line
```

challenge

Try this variation:

- Revise `main` to:

```
  Animal a;
  Substitution(a);
```

▼ **Why did this produce an error?**
The `Animal` class is not derived from `Person`, therefore the Substitution Principle no longer applies.

The Substitution Principle is a One-Way Relationship

Let's revise the `Substitution` function and `main` to look like below:

```
//add function definitions below this line

void Substitution(Superhero s) {
  s.Greeting();
}

//add function definitions above this line
```

```
  //add code below this line

  Person p;
  Substitution(p);

  //add code above this line
```

The code above produces an error stating that the object `p` cannot be converted to a `Superhero` object. Kind of like how a square is considered to be a rectangle, but a rectangle is not considered to be a square, inheritance works the same way. A `Superhero` object is considered to be a `Person` object, but a `Person` object is not considered to be `Superhero` object.

Inheritance Can Be Extended

Inheritance can be **extended**, which means that a derived class can inherit attributes of another derived class. Revise the class definitions to look like below:

```
//add class definitions below this line

class Person {
  public:
    void Greeting() {
      cout << "I'm a Person" << endl;
    }
};

class Hero : public Person {
  public:
    void Greeting() {
      cout << "I'm a Hero" << endl;
    }
};

class Superhero: public Hero {
  public:
    void Greeting() {
      cout << "I'm a Superhero" << endl;
    }
};

//add class definitions above this line
```

You’ll notice that we have an intermediate class called `Hero` which is a derived class of `Person`. The `Superhero` class is then derived from the `Hero` class. When there are multiple levels of inheritance, the immediate upper class is considered to be a **direct** base class and all higher classes are considered to be **indirect** base classes. This means that `Hero` is a direct base class of `Superhero` while `Person` is an indirect base class of `Superhero`. `Superhero` is still considered to be a derived class of both `Hero` and `Person`.

See what happens when you you revise the `Substitution` function and `main` to look like below:

```
//add function definitions below this line

void Substitution(Person p) {
  p.Greeting();
}

//add function definitions above this line
```

```
//add code below this line

Superhero s;
Substitution(s);
Hero h;
Substitution(h);

//add code above this line
```

Whether the object is of class `Superhero` or `Hero`, the `Substitution` function still works because both are derived classes of `Person`.