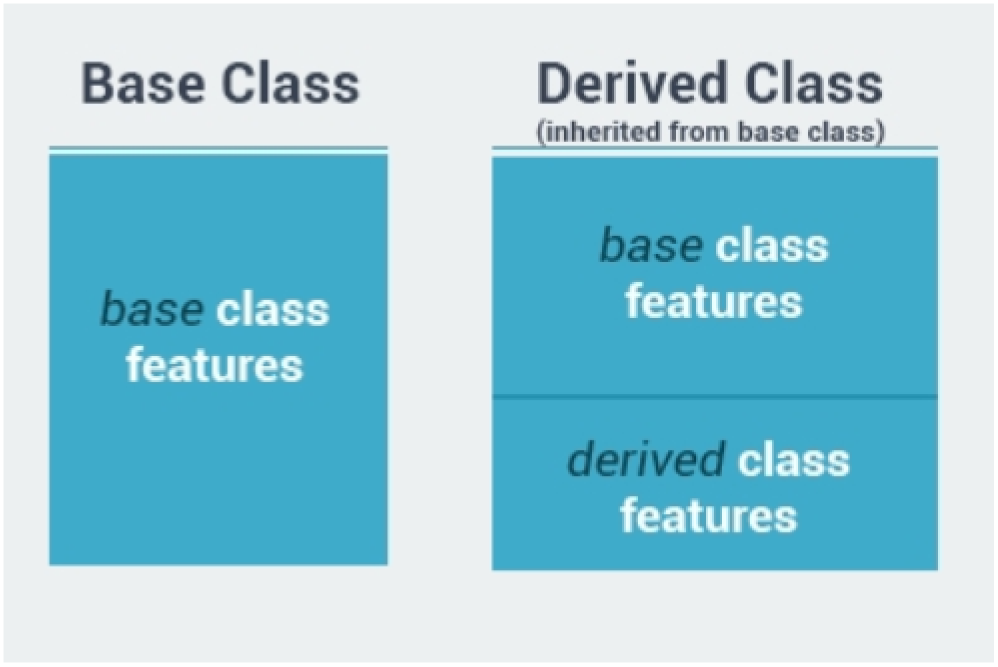
Inheritance & Polymorphism

Inheritance

Inheritance is one of the most important concepts of object-oriented programming.
Inheritance allows us to define a class based on another class. This facilitates greater ease in creating and maintaining an application.

The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class. For example, the **Daughter** class (derived) can be inherited from the **Mother** class (base).
The derived class inherits all feature from the base class, and can have its own additional features.



The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal, hence dog IS-A animal as well.

Inheritance

To demonstrate inheritance, let's create a **Mother** class and a **Daughter** class:

```
class Mother
{
 public:
  Mother() {};
  void sayHi() {
    cout << "Hi";
  }
};
```

```
class Daughter
{
 public:
  Daughter() {};
};
```

The Mother class has a public method called **sayHi()**.

> The next step is to **inherit** (derive) the Daughter from the Mother.

Inheritance

This syntax derives the **Daughter** class from the **Mother** class.

```
class Daughter : public Mother
{
 public:
  Daughter() {};
};
```

The Base class is specified using a **colon** and an **access specifier**: **public** means, that all public members of the base class are public in the derived class.

> In other words, all public members of the **Mother** class become public members of the **Daughter** class.

Inheritance

As all public members of the Mother class become public members for the Daughter class, we can create an object of type Daughter and call the **sayHi()** function of the Mother class for that object:

```
#include <iostream>
using namespace std;

class Mother
{
 public:
  Mother() {};
  void sayHi() {
   cout << "Hi";
  }
};

class Daughter: public Mother
{
 public:
  Daughter() {};
};

int main() {
  Daughter d;
  d.sayHi();
}
//Outputs "Hi"
```

Try It Yourself

A derived class inherits all base class methods with the following exceptions:
- Constructors, destructors
- Overloaded operators
- The friend functions

> A class can be derived from multiple classes by specifying the base classes in a **comma-separated** list. For example: **class Daughter: public Mother, public Father**

Access Specifiers

Up to this point, we have worked exclusively with **public** and **private** access specifiers.
Public members may be accessed from anywhere outside of the class, while access to private members is limited to their class and friend functions.

> As we've seen previously, it's a good practice to use public methods to access private class variables.

Protected

There is one more access specifier - **protected**.
A **protected** member variable or function is very similar to a private member, with one difference - it can be accessed in the derived classes.

```
class Mother {
 public:
  void sayHi() {
   cout << var;
  }

 private:
  int var=0;

 protected:
  int someVar;
};
```

> Now **someVar** can be accessed by any class that is **derived** from the Mother class.

Type of Inheritance

Access specifiers are also used to specify the **type of** inheritance.
Remember, we used **public** to inherit the Daughter class:

```
class Daughter: public Mother
```

private and **protected** access specifiers can also be used here.

Public Inheritance: public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Protected Inheritance: public and protected members of the base class become protected members of the derived class.

Private Inheritance: public and protected members of the base class become private members of the derived class.

Public inheritance is the most commonly used inheritance type.
If no access specifier is used when inheriting classes, the type becomes **private** by default.

Inheritance

When inheriting classes, the base class' constructor and destructor are not inherited.
However, they are being called when an object of the derived class is created or deleted.

To further explain this behavior, let's create a sample class that includes a constructor and a destructor:

```
class Mother {
 public:
 Mother()
 {
  cout <<"Mother ctor"<<endl;
 }
 ~Mother()
 {
  cout <<"Mother dtor"<<endl;
 }
};
```

Creating an object in main results in the following output:

```
int main() {
  Mother m;
}
/* Outputs
Mother ctor
Mother dtor
*/
```

Try It Yourself

The object is created and then deleted, when the program finishes to run.

Inheritance

Next, let's create a **Daughter** class, with its own constructor and destructor, and make it a derived class of the **Mother**:

```
class Daughter: public Mother {
public:
 Daughter()
 {
```

```
 cout <<"Daughter ctor"<<endl;
}
~Daughter()
{
 cout <<"Daughter dtor"<<endl;
}
};
```

Create a Daughter class, with its own constructor and destructor.

Inheritance

Now, what happens when we create a **Daughter** object?

```
int main() {
  Daughter m;
}

/*Outputs
Mother ctor
Daughter ctor
Daughter dtor
Mother dtor
*/
```

Try It Yourself

Note that the base class' constructor is called first, and the derived class' constructor is called next.
When the object is destroyed, the derived class's destructor is called, and then the base class' destructor is called.

You can think of it as the following: The derived class needs its base class in order to work - that is why the base class is set up first.

Summary

Constructors
The base class constructor is called first.

Destructors
The derived class destructor is called first, and then the base class destructor gets called.

This sequence makes it possible to specify initialization and de-initialization scenarios for your derived classes.

Polymorphism

The word **polymorphism** means "having many forms".
Typically, polymorphism occurs when there is a hierarchy of classes and they are related by **inheritance**.

C++ polymorphism means that a call to a member function will cause a **different** implementation to be executed depending on the **type** of object that invokes the function.

> Simply, polymorphism means that a single function can have a number of different implementations.

Polymorphism

Polymorphism can be demonstrated more clearly using an example:
Suppose you want to make a simple game, which includes different enemies: monsters, ninjas, etc. All enemies have one function in common: an **attack** function. However, they each attack in a different way. In this situation, polymorphism allows for calling the same **attack** function on different objects, but resulting in different behaviors.

The first step is to create the **Enemy** class.

```
class Enemy {
 protected:
  int attackPower;
 public:
  void setAttackPower(int a){
   attackPower = a;
  }
};
```

> Our Enemy class has a public method called **setAttackPower**, which sets the protected **attackPower** member variable.

Polymorphism

Our second step is to create classes for two different types of enemies, **Ninjas** and **Monsters**. Both of these new classes inherit from the **Enemy** class, so each has an attack power. At the same time, each has a specific **attack** function.

```
class Ninja: public Enemy {
 public:
  void attack() {
   cout << "Ninja! - "<<attackPower<<endl;
  }
};

class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster! - "<<attackPower<<endl;
  }
};
```

As you can see, their individual **attack** functions differ.
Now we can create our **Ninja** and **Monster** objects in main.

```
int main() {
 Ninja n;
 Monster m;
}
```

Ninja and **Monster** inherit from **Enemy**, so all **Ninja** and **Monster** objects are **Enemy** objects. This allows us to do the following:

```
Enemy *e1 = &n;
Enemy *e2 = &m;
```

We've now created two pointers of type **Enemy**, pointing them to the **Ninja** and **Monster** objects.

Polymorphism

Now, we can call the corresponding functions:

```
int main() {
  Ninja n;
  Monster m;
  Enemy *e1 = &n;
  Enemy *e2 = &m;

 e1->setAttackPower(20);
 e2->setAttackPower(80);

 n.attack();
 m.attack();
}

/* Output:
Ninja! - 20
Monster! - 80
*/
```

Try It Yourself

We would have achieved the same result by calling the functions directly on the objects. However, it's faster and more efficient to use pointers.
Also, the pointer demonstrates, that you can use the **Enemy** pointer without actually knowing that it contains an object of the subclass.

Tap **Continue** to learn more!

Virtual Functions

The previous example demonstrates the use of base class pointers to the derived classes. Why is that useful? Continuing on with our game example, we want every Enemy to have an **attack()** function.
To be able to call the corresponding attack() function for each of the derived classes using Enemy pointers, we need to declare the base class function as **virtual**.
Defining a virtual function in the base class, with a corresponding version in a derived class, allows polymorphism to use Enemy pointers to call the derived classes' functions.
Every derived class will override the attack() function and have a separate implementation:

```
class Enemy {
 public:
  virtual void attack() {
  }
};

class Ninja: public Enemy {
 public:
  void attack() {
   cout << "Ninja!"<<endl;
  }
};
```

```
class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster!"<<endl;
  }
};
```

A virtual function is a base class function that is declared using the keyword **virtual**.

Virtual Functions

Now, we can use **Enemy** pointers to call the **attack()** function.

```
int main() {
  Ninja n;
  Monster m;
  Enemy *e1 = &n;
  Enemy *e2 = &m;

  e1->attack();
  e2->attack();
}

/* Output:
Ninja!
Monster!
*/
```

Try It Yourself

As the attack() function is declared virtual, it works like a template, telling that the derived class might have an **attack()** function of its own.

Virtual Functions

Our game example serves to demonstrate the concept of polymorphism; we are using **Enemy** pointers to call the same **attack()** function, and generating different results.

```
e1->attack();
e2->attack();
```

If a function in the base class is **virtual**, the function's implementation in the derived class is called according to the actual type of the object referred to, regardless of the declared type of the pointer.

A class that declares or inherits a virtual function is called a **polymorphic** class.

Virtual Functions

Virtual functions can also have their implementation in the base class:

```
class Enemy {
 public:
  virtual void attack() {
   cout << "Enemy!"<<endl;
  }
};

class Ninja: public Enemy {
 public:
  void attack() {
   cout << "Ninja!"<<endl;
  }
};

class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster!"<<endl;
  }
};
```

Now, when you create an **Enemy** pointer, and call the **attack()** function, the compiler will call the function, which corresponds to the object's type, to which the pointer points:

```
int main() {
 Ninja n;
 Monster m;
 Enemy e;

 Enemy *e1 = &n;
 Enemy *e2 = &m;
 Enemy *e3 = &e;

 e1->attack();
 // Outputs "Ninja!"

 e2->attack();
 // Outputs "Monster!"

 e3->attack();
 // Outputs "Enemy!"
}
```

Try It Yourself

This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Pure Virtual Function

In some situations you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.
The virtual member functions without definition are known as **pure virtual functions**. They basically specify that the derived classes define that function on their own.
The syntax is to replace their definition by =0 (an equal sign and a zero):

```
class Enemy {
 public:
  virtual void attack() = 0;
};
```

The = 0 tells the compiler that the function has no body.

Pure Virtual Functions

A **pure** virtual function basically defines, that the derived classes will have that function defined on their own.
Every derived class inheriting from a class with a pure virtual function **must** override that function.

> If the pure virtual function is not overridden in the derived class, the code fails to compile and results in an error when you try to instantiate an object of the derived class.

Pure Virtual Functions

The pure virtual function in the **Enemy** class must be overridden in its derived classes.

```
class Enemy {
 public:
  virtual void attack() = 0;
};

class Ninja: public Enemy {
 public:
  void attack() {
   cout << "Ninja!"<<endl;
  }
};

class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster!"<<endl;
  }
};
```

Try It Yourself

> Tap **Try It Yourself** to play around with the code!

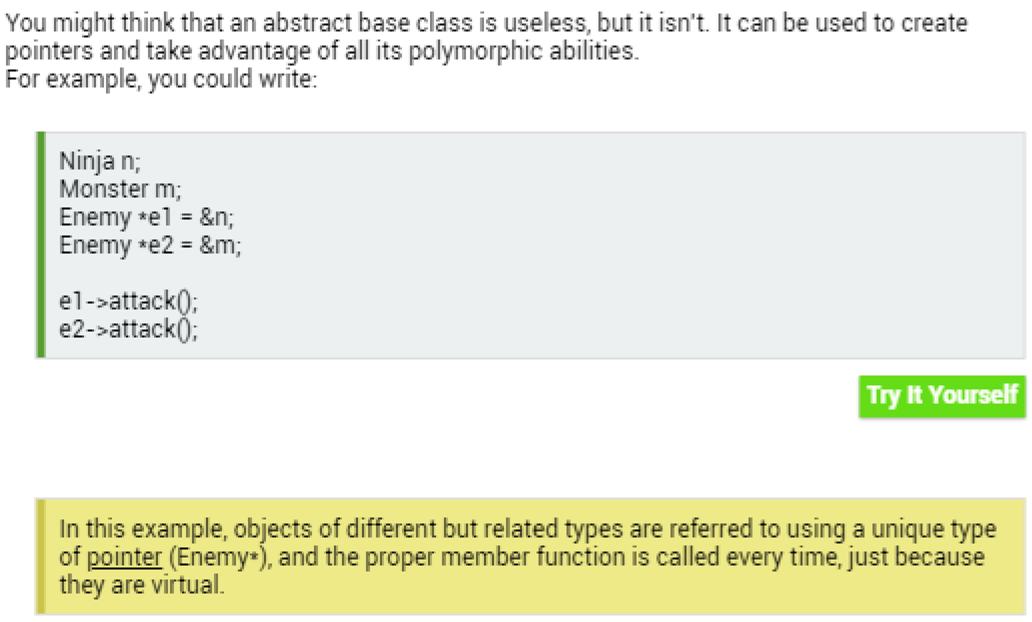
Abstract Classes

You **cannot** create objects of the base class with a pure virtual function.
Running the following code will return an error:

```
Enemy e; // Error
```

Try It Yourself

These classes are called **abstract**. They are classes that can only be used as base classes, and thus are allowed to have pure virtual functions.



You might think that an abstract base class is useless, but it isn't. It can be used to create pointers and take advantage of all its polymorphic abilities.
For example, you could write:

```
Ninja n;
Monster m;
Enemy *e1 = &n;
Enemy *e2 = &m;

e1->attack();
e2->attack();
```

In this example, objects of different but related types are referred to using a unique type of pointer (Enemy*), and the proper member function is called every time, just because they are virtual.

End.