Object-Oriented Programming
Inheritance

Inheritance

- Class Inheritance allows us to reuse, extend, and change the behavior of a class.
- New classes are created using an existing class.
- The new class is called a **“derived”** class or **“child”** class.
- The existing class is called the **“base”** class or **“parent”** class.
- The derived class inherits all the properties of the base.

Inheritance

class <derived_class_name> : <access-specifier> <base_class_name>

{

}

Modes of Inheritance

- **Public inheritance**: the public members of the base will be public in the derived and protected members of the base will be protected in the derived.
- **Protected inheritance**: both public and protected members of the base will become protected in the derived.
- **Private inheritance**: both public and protected members of the base will become private in the derived.

Modes of Inheritance

```
class A
{
public:
    int x;

protected:
    int y;

private:
    int z;
};
```

```
class B : public A
{
    //x is public
    //y is protected
    //z is not accessible from B
};

class C : protected A
{
    //x is protected
    //y is protected
    //z is not accessible from C
};

class D : private A //private is the default
{
    //x is private
    //y is private
    //z is not accessible from D
};
```

Object-Oriented Programming
Constructors

Inheritance and Constructors

- When making a constructor for the derived class, it *must* call a base class constructor.
- Since you can define any number of constructors, you get to choose which constructor to call.

Inheritance and Constructors

- To call a base class constructor from the derived constructor, add it to the member initialization list.
- EX:

```
class Car
{
public:
     Car() :
          mMake("Ford"),
          mModel("A"),
          mModelYear(1908)
     {   }

     Car(int year, string mk, string mdl) :
          mModelYear(year),
          mMake(make),
          mModel(model)
     {}
```

```
class FlyingCar : public Car
{
public:
     FlyingCar()//calls the default ctor of the base
     {}

     FlyingCar(int maxAlt, int yr, string mk, string mdl)
          : Car(yr, make, model) //calls the base ctor
     {}
```

Object-Oriented Programming
Protected

Protected

- The derived class can directly access all of the public and protected members of the base class.
- Private members of the base class, even though they are part of the derived class, cannot be directly accessed by the code in the derived class.

Object-Oriented Programming
Polymorphism

Polymorphism

- Polymorphism means many shapes/forms.
- Two kinds of polymorphism:
  - Compile-time polymorphism
  - Runtime polymorphism

Object-Oriented Programming
Compile-Time Polymorphism

Polymorphism
Compile-time polymorphism

- Two kinds of Compile-time polymorphism:
  - Method (or function) overloading
  - Operator overloading

Polymorphism
Method Overloading

- **Method Overloading** – multiple methods with the same name but different parameters.
  - Achieved by…
    - Changing the number of parameters
    - Changing the types of the parameters

Polymorphism
Method Overloading

```
int add(int n1, int n2);
int add(int n1, int n2, int n3);
float add(float f1, float f2);
double add(double d1, double d2);
```

Polymorphism
Operator Overloading

- **Operator Overloading** – adding different behavior to the operators (+, -, etc).
- Operator Overloading in C++ - GeeksforGeeks

Polymorphism
Operator Overloading

```
class Account
{
private:
    double mBalance;
public:
    Account operator+(Account const& obj)
    {
        Account result;
        result.mBalance = mBalance + obj.mBalance;
        return result;
    }
};
```

Object-Oriented Programming
Runtime Polymorphism

Polymorphism
Runtime polymorphism

- Runtime polymorphism is achieved by **method (or function) overriding**.
- The function call is **resolved at runtime**, not during compilation.

- Method overriding happens when a derived class has a definition for a base class method. The base class method is said to be overridden.

Polymorphism
Runtime polymorphism

- A **virtual method** is a base class method using the **keyword** virtual.
- The method call is determined at runtime.

```
class base
{
public:
    virtual void print()
    {
        printf("Hello base.");
    }
};
```

```
class derived : public base
{
public:
    //overrides the base version
    void print()
    {
        printf("Hello derived.");
    }
};
```

Polymorphism
Runtime polymorphism

- You can optionally add **override** to the derived method for extra compile time checking. It indicates that you intend to override a base method with this method. If the signature doesn't match exactly, then a build error occurs.

```
class base
{
public:
    virtual void print()
    {
        printf("Hello base.");
    }
};
```

```
class derived : public base
{
public:
     //overrides the base version
    void print() override
    {
        printf("Hello derived.");
    }
};
```

Object-Oriented Programming
Unique Pointers

Unique Pointers

- A unique pointer (**std::unique_ptr**) owns and manages another object through a pointer. It disposes of the object when it goes out of scope.
- Use `#include <memory>`

Making Unique Pointers

- Use the **make_unique** method to generate a unique pointer.
- It creates and returns a unique_ptr to an object of the specified type, which is constructed by using the specified arguments.
- You pass arguments to it that match the arguments needed to call the **constructor on the object**.

Making Unique Pointers

- Constructor of the derived class:

```
derived(std::string str, int num) : base(num), mStr(str)  { }
```

- Since the constructor has 2 parameters, we must call make_unique with 2 arguments for those parameters.

```
std::unique_ptr<derived> pDerived = std::make_unique<derived>("Gotham", 5);
```

Object-Oriented Programming
Upcasting

Upcasting

- Upcasting is when you point to derived object through a base type variable.
- This is safe because the compiler knows the hierarchy.
- A derived type can always be casted to a base type.

Upcasting

```
std::unique_ptr<base> pBase = std::make_unique<derived>("Gotham", 5);
pBase->print(); //will call the derived version since it's overridden
```

Because the print method is virtually overridden, it will call the derived version of print, not the base version of print.

Upcasting

If you already have a unique_ptr to a derived type, then you can use the std::move method to upcast.

```
//get a unique ptr to a derived type
std::unique_ptr<derived> pDerived = std::make_unique<derived>("Gotham", 5);


//move the unique ptr to a base type
std::unique_ptr<base> pBase = std::move(pDerived);
```