# Lecture 15 Inheritance

#CS106L

- **Recap**
- **Namespaces**
- **Inheritance**

---

const before parameter vs const after function

---

- to be an interface, a class must consist only of pure virtual functions
- to implement an interface, a class must define all of those virtual functions
- if a class has at least one pure virtual function, then it's called an abstract class(interfaces are a subset of abstract class)
- abstract class cann't be instantiated
- no virtual members-instead, if a member has the same name as an inherited member, it hides it

```
struct A{

    int a;

};

struct B : public A{

    double a;         //hides A::a
```

```
};
```

---

## Q1: `namespace`?

In C++, a namespace is a container that holds a set of identifiers (such as variable names, function names, and class names) that can be used to organize and group related code. Namespaces are intended to prevent naming conflicts between different libraries, frameworks, and modules, by providing a way to group related code together and give it a unique name.

Here's an example of how to define a namespace in C++:

```
namespace myNamespace {

    int myVariable = 5;

    void myFunction() {

        // code here

    }

    class MyClass {

        // code here

    };

}
```

In this example, the `myNamespace` namespace contains an integer variable `myVariable`, a function `myFunction()`, and a class `MyClass`. To access these identifiers from outside of the namespace, you can use the `::` operator to qualify the identifier with the namespace name. For example:

```
        int x = myNamespace::myVariable;

        myNamespace::myFunction();

        myNamespace::MyClass myObject;
```

A namespace can also include other namespaces, and you can use the `using` keyword to make the identifiers in a namespace available without the need to qualify them with the namespace name. For example:

```
using namespace myNamespace;

        int x = myVariable;

        myFunction();

        MyClass myObject;
```

It's worth noting that, Namespaces can also be defined inside of a function, class or another namespace, which is called an anonymous namespace. An anonymous namespace can only be used in the file it's defined in and it's intended to provide a way to group related code together without giving it a unique name.

```
namespace {

    int myVariable = 5;

    void myFunction() {

        // code here

    }

    class MyClass {

        // code here

    };

}
```

## Q2: `inheritance`?

In C++, inheritance is a way of creating a new class that is based on an existing class, and it allows the new class to inherit the properties and methods of the existing class, while also adding new properties and methods of its own.

Here's an example of how to use inheritance in C++:

```
class Base {

public:

    int x;

    void print() {

        std::cout << "x = " << x << std::endl;

    }

};



class Derived: public Base {

public:

    int y;

    void print() {

        std::cout << "x = " << x << ", y = " << y << std::endl;

    }

};
```

In this example, the `Derived` class inherits from the `Base` class using the `: public Base` syntax. This means that the `Derived` class inherits all of the public properties and methods of the `Base` class, including the `x` variable and the `print()` method. The `Derived` class can also define its own properties and methods, such as the `y` variable and its own implementation of the `print()` method.

Inheritance can be useful in situations where you want to reuse code and avoid duplicating it. It allows you to create a new class that inherits the properties and methods of an existing class, and then add new properties and methods of its own.

## Q3: `virtual`?

In C++, the `virtual` keyword is used in the context of polymorphism, to indicate that a member function or a class destructor is virtual. A virtual function is a member function of a class that can be overridden by derived classes, allowing them to provide their own implementation. A virtual function is called through a base class pointer or reference, and it will execute the most-derived version of the function that is accessible through the pointer or reference at runtime.

Here's an example of how to use virtual functions in C++:

```
class Shape {

public:

    virtual void draw() = 0;

};



class Circle: public Shape {

public:

    void draw() override {

        std::cout << "Drawing a circle" << std::endl;

    }

};



class Square: public Shape {

public:

    void draw() override {

        std::cout << "Drawing a square" << std::endl;

    }

};
```

In this example, the `Shape` class has a pure virtual function `draw()`, indicated by the `= 0` at the end of the function declaration. This means that the `Shape` class is an abstract class, and it cannot be instantiated. The `Circle` and `Square` classes inherit from the `Shape` class and provide their own implementation of the `draw()` function using the `override` keyword.

Now, if we have a pointer or reference of the Shape class, we can refer to any derived class of Shape and call the draw method, the corresponding class' implementation will be called.

```
        Shape * shape1 = new Circle();

        Shape * shape2 = new Square();

        shape1->draw(); // will print "Drawing a circle"

        shape2->draw(); // will print "Drawing a square"
```

The `virtual` keyword also can be used to indicate that a class destructor is virtual. This means that when an object of a class that has a virtual destructor is deleted through a base class pointer or reference, the correct derived class destructor will be called. This is important to avoid resource leaks, as the derived class may have allocated resources that need to be freed in its destructor.