Lecture 16 Inheritance and Template Classes

#CS106L

- **Finishing Inheritance**
- **Template Classes**
- **Concepts Revisited**

1.

- base class: the class that gets inherited from
- derived class: the class that inherits from the base class
- if you intend to make your class inheritable(i.e. if your class has any virtual functions), make your destructor virtual!

```
        virtual ~Base () {}
```

otherwise will almost definitely have memory leaks.

2. private: can only be accessed by this class

   protected: can be accessed by this class or derived classes

   public: can be accessed by anyone

3. templates(static polymorphism) vs derived classes(dynamic polymorphism)

- The main difference between templates and derived classes is that templates are used to create generic types and functions that can work

with multiple types of data, while derived classes are used to create new classes that inherit the properties and methods of existing classes.

prefer templates when:

- runtime efficiency is most important
- not common base can be defined

prefer derived classes when:

- compile-time efficiency is most important
- intend to hide implementations
- don't want code bloat

Q1:pure virtual function and no pure virtual function?

In C++, pure virtual functions are functions that are declared with the `= 0` syntax, indicating that they are pure virtual functions and do not have an implementation. A class that contains pure virtual functions is called an abstract class, and it cannot be instantiated. Instead, it must be inherited by derived classes, which must provide their own implementation of the pure virtual functions.

On the other hand, non-pure virtual functions are functions that have an implementation and can be overridden by derived classes. Derived classes can choose to override the non-pure virtual function to provide their own implementation or to use the base class implementation.

Here's an example of pure virtual functions and non-pure virtual functions:

```
class Shape {

public:

    virtual void draw() = 0; // pure virtual function
```

```
    virtual void print() { std::cout << "Shape" << std::endl; } // non-pure virtual function

};



class Circle: public Shape {

public:

    void draw() override { std::cout << "Drawing a Circle" << std::endl; }

    void print() override { std::cout << "Circle" << std::endl; }

};
```

In this example, the `Shape` class has a pure virtual function `draw()`, and a non-pure virtual function `print()`, the derived class `Circle` must provide an implementation for the pure virtual function `draw()` but it can choose whether to override the non-pure virtual function `print()` or not.

Q2:polymorphism?

In object-oriented programming, polymorphism is the ability of an object or a function to take on multiple forms. There are two main types of polymorphism in C++:

1. Compile-time polymorphism (also known as static polymorphism or overloading): This type of polymorphism occurs when a single function or operator can take multiple forms based on the number or types of its arguments. This is achieved through function overloading, where multiple functions have the same name but different parameter lists, or operator overloading, where operators have different meanings depending on the types of their operands.
2. Run-time polymorphism (also known as dynamic polymorphism or overriding): This type of polymorphism occurs when a single function can take multiple forms based on the type of the object on which it is called. This is achieved through virtual functions, where the most-derived version of a function that is accessible through a base class pointer or reference is called at runtime.

Q3:class template?

A class template in C++ is a blueprint for creating classes that can work with multiple types of data. It allows you to define a class that can be instantiated with different types of data, without having to write separate code for each type.

A class template is defined with the `template` keyword followed by a template parameter list enclosed in angle brackets (<>). The template parameter list specifies the types that can be used to instantiate the class. Here's an example of a simple class template:

```
template <typename T>

class MyClass {

public:

    MyClass(T value) : _value(value) {}

    void setValue(T value) { _value = value; }

    T getValue() const { return _value; }



private:

    T _value;

};
```

In this example, the class `MyClass` is a template class that takes one template parameter `T`. This template parameter can be replaced by any type when the class is instantiated.

Class templates can also have multiple template parameters, which allows you to create more complex classes that can work with multiple types of data. Here's an example of a class template with multiple template parameters:

```
template <typename T, typename U>

class MyPair {
```

```
public:

    MyPair(T first, U second) : _first(first), _second(second) {}

    T getFirst() const { return _first; }

    U getSecond() const { return _second; }



private:

    T _first;

    U _second;

};
```

Q4: `static_cast`?

In C++, the `static_cast` operator is used to perform a type conversion from one type to another. It is used to convert a value from one type to another, where the target type is specified in the cast expression. The `static_cast` operator performs compile-time type checking and generates an error if the conversion is not safe.

Here's an example of how to use `static_cast`:

```
        int a = 10;

        float b = static_cast<float>(a);
```

`static_cast` can also be used to cast pointer and references to classes up or down a hierarchy. For example:

```
class Shape {};

class Circle : public Shape {};

Circle c;

Shape &s = static_cast<Shape&>(c);
```

In this example, `c` is an instance of `Circle`, which is derived from `Shape`, so it is safe to convert a reference of `Circle` to `Shape`.

It can also be used to cast pointers and references to void*

```
Circle c;

void* v = static_cast<void*>(&c);
```

It's important to note that `static_cast` does not perform any runtime type checking and does not invoke any constructors or destructors. It does not check if the conversion is safe and it will not cause a compile error if the conversion is not safe. Therefore, it is important to use `static_cast` with caution and only when you are sure that the conversion is safe.

Q5:references or pointers with inheritance?

In C++, references and pointers can be used with inheritance to allow an object of a derived class to be accessed through a reference or pointer of the base class. This is called polymorphism, which allows a single function or operator to take multiple forms based on the type of the object on which it is called.

Here's an example of how to use references with inheritance:

```
class Shape {

public:

    virtual void draw() = 0;

};



class Circle: public Shape {

public:

    void draw() override {

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

    }
```

```
};



void drawShape(Shape& shape) {

    shape.draw();

}



int main(){

    Circle c;

    drawShape(c);

}
```

In this example, the `drawShape` function takes a reference of `Shape`, it doesn't care which type of object it is, it only needs to know that the object has a draw method. This allows the function to work with any derived class of `Shape`, such as `Circle`.

Here's an example of how to use pointers with inheritance:

```
class Shape {

public:

    virtual void draw() = 0;

};



class Circle: public Shape {

public:

    void draw() override {

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

```
    }
};


void drawShape(Shape* shape) {
    shape->draw();
}


int main(){
    Circle c;
    drawShape(&c);
}
```

In this example, the `drawShape` function takes a pointer of `Shape`, it doesn't care which type of object it is, it only needs to know that the object has a draw method. This allows the function to work with any derived class of `Shape`, such as `Circle`.