19CSE201 :Advanced Programming

Lecture 16
Some Special functions & Classes in C++

By
Ritwik M
Assistant Professor(SrGr)
Dept. Of Computer Science & Engg
Amrita Vishwa Vidyapeetham - Coimbatore

Based on material from Stroustrup B. Programming: principles and practice using C++. Second edition, Addison Wesley; 2014 and multiple sources on the internet

A Quick Recap

- Basics of C++
- Pointers in C++
- Memory Management in C++
- OOP Concepts in C++
    - Classes
    - Objects
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism
- Examples
- Exercises

Some Special Functions and Classes

- Friend Function
- Virtual Function
- Interfaces
    - Pure Virtual Functions
    - Abstract Class

Friend Function

- A friend function is a non - member function of a class.
- A friend function of a class is defined outside that class' scope, but it has the right to access all private and protected members of the class.
- A friend can be a function, member function, or a class in which case the entire class and all of its members are friends.
- Syntax

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

Friend Function - Example

```
class Distance {
    private:
     int meter;
     // friend function
     friend int addFive(Distance);
    public:
     Distance() : meter(0) {}

};
```

```
// friend function definition
   int addFive(Distance d) {
//accessing private members from the friend
  //function
    d.meter += 5;
    return d.meter;
}
```

```
//Main function
int main() {
   Distance D;
   cout << "Distance ="<<addFive(D)
   return 0;
}
```

Some Special Functions and Classes

- Friend Function ✓
- Virtual Function
- Interfaces
  - Pure Virtual Functions
  - Abstract Class

Virtual Function

- A virtual function or virtual method is an inheritable and overridable function or method for which dynamic dispatch is facilitated.
- In short, a virtual function defines a target function to be executed, but the target might not be known at compile time.
- A virtual function is a member function that is declared as virtual within a base class and redefined by a derived class.
- Syntax
  - `virtual dataType functionName()`
- When a class containing virtual function is inherited, the derived class can redefine (Override) the virtual function to suit its own unique needs.
- Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.
- The method name and type signature should be same for both base and derived version of function.

Virtual Function - Example

```
class base
{
    public:
        virtual void show()
        {
            cout<<"from base class"<<endl;
        }
};
class derived : public base
{
        public:
        void show()
        {
            cout<<"from derived class"<<endl;
        }
};
```

- Using the virtual keyword with the base class version of show function, the late binding takes place, and the derived version of the function is called since the base pointer points a derived type of object
- To invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class

```
//Main function
int main()
{
    base*ptr;
    derived d;
    ptr=&d;
    ptr->show();
    return 0;
}
```

A Practical (real-world) Application

- Consider an employee management software for an organization.
- Let the code has a simple base class Employee, the class contains virtual functions like raiseSalary(), transfer(), promote(), etc. Different types of employees like Manager, Engineer, etc. may have their own implementations of the virtual functions present in base class Employee.
- In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don't need to worry about them because if raiseSalary() is present for a specific employee type, only that function would be called.

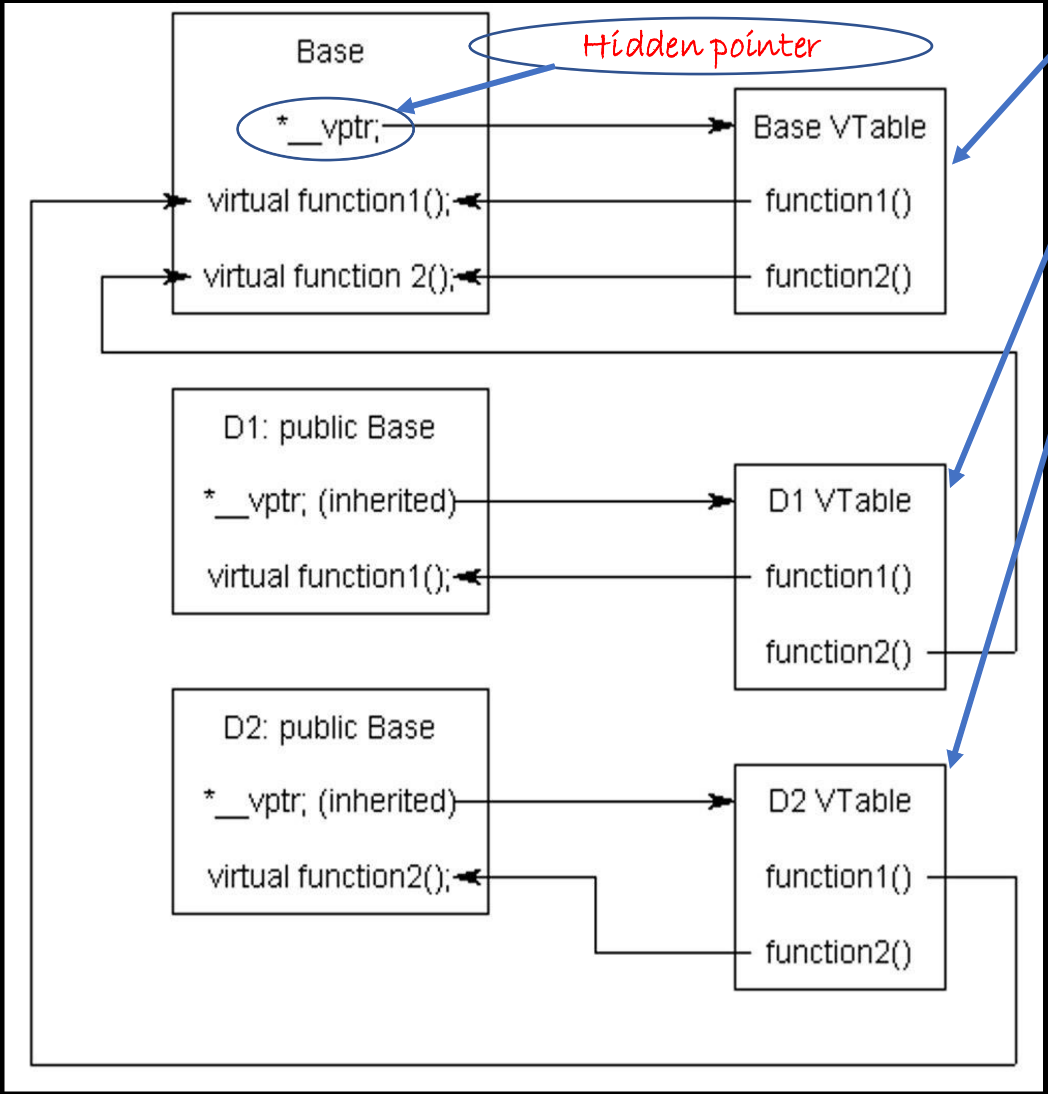
Virtual Functions Working: The virtual table

- To implement virtual functions, C++ uses a special form of late binding known as the virtual table.
- The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner
- Description
  - First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table.
    - This table is simply a static array that the compiler sets up at compile time
    - A virtual table contains one entry for each virtual function that can be called by objects of the class.
    - Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.
  - The compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class
    - Unlike the *this pointer, which is actually a function parameter used by the compiler to resolve self-references, *__vptr is a real pointer.
    - Consequently, it makes each class object allocated bigger by the size of one pointer.
    - It also means that *__vptr is inherited by derived classes, which is important.

```
class Base
{
public:
    virtual void function1()
        {cout<<"Base1";}
    virtual void function2()
        {cout<<"Base2";}
};
class D1: public Base
{
public:
 virtual void
  function1()  {cout<<"derived
  1";}
};
class D2: public Base
{
public:
    virtual void function2()
  {cout<<"derived2";}
};
```

The virtual table - Example

3 Classes – 3 Tables



```
//Main function
int main()
{
 D1 d1;
 Base *dPtr = &d1;
 DPtr -> function 1;
//what if dPtr pointed
//to a Base object instead
//of a D1 object?
 Base b;
 Base *bPtr = &b;
 bPtr->function1();
 return 0;
}
```

The virtual table - Example Explained

- When a class object is created, `*__vptr` is set to point to the virtual table for that class.
  - For example, when an object of type Base is created, `*__vptr` is set to point to the virtual table for Base.
    - When objects of type D1 or D2 are constructed, `*__vptr` is set to point to the virtual table for D1 or D2 respectively.
  - Because there are only two virtual functions here, each virtual table will have two entries (one for `function1()` and one for `function2()`).
    - Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.
- The virtual table for Base objects is simple.
    - An object of type Base can only access the members of Base.
    - Base has no access to D1 or D2 functions.
    - Consequently, the entry for `function1` points to `Base::function1()` and the entry for `function2` points to `Base::function2()`.
- The virtual table for D1 is slightly more complex.
    - An object of type D1 can access members of both D1 and Base.
    - However, `D1` has overridden `function1()`, making `D1::function1()` more derived than `Base::function1()`.
    - Consequently, the entry for `function1` points to `D1::function1()`. `D1` hasn't overridden `function2()`, so the entry for `function2` will point to `Base::function2()`.
  - The virtual table for D2 is similar to D1, except the entry for `function1` points to `Base::function1()`, and the entry for `function2` points to `D2::function2()`

Rules for Virtual Functions

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- We cannot have a virtual constructor, but we can have a virtual destructor
- The return type of the virtual function must be consistent throughout all of its implementing classes

More about Virtual Functions

- Calling a virtual function is slower than calling a non-virtual function
  - First, we have to use the `*__vptr` to get to the appropriate virtual table.
  - Second, we have to index the virtual table to find the correct function to call.
  - Third, we call the function.
    - As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call.
    - However, with modern computers, this added time is usually insignificant.

Some Special Functions and Classes

- Friend Function ✓
- Virtual Function ✓
- Interfaces
  - Pure Virtual Functions
  - Abstract Class

Interfaces in C++

- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The C++ interfaces are implemented using abstract classes
- A class is made abstract by declaring at least one of its functions as a pure virtual function.
- NOTE: These abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

Pure Virtual Function

- A pure virtual function (abstract function) is a virtual function in C++ which does not need a function definition and only declaration is required.
  - It is declared by assigning 0 in the declaration.
  - Eg: `virtual void display() = 0;`
- This must be overridden in a derived class

Pure Virtual Function - Example

```
class Base
{ //Adding a pure virtual function to a class implies that it is up to the derived
  classes to implement this function.
public:
    const char* sayHi() const { return "Hi"; } // a normal non-virtual function
    virtual const char* getName() const { return "Base"; } // a normal virtual function
    virtual int getValue() const = 0; // a pure virtual function
    int doSomething() = 0; // Compile error: can not set non-virtual functions to 0
};
```

- Using a pure virtual function has two main consequences:
  - First, any class with one or more pure virtual functions becomes an abstract base class
    - which means that it can not be instantiated!
  - Second, any derived class must define a body for this function,
    - or that derived class will be considered an abstract base class as well

Some Special Functions and Classes

- Friend Function ✓
- Virtual Function ✓
- Interfaces
  - Pure Virtual Functions ✓
  - Abstract Class

Abstract Class

- The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit.
- Abstract classes cannot be used to instantiate objects and serves only as an interface for its sub classes.
- Attempting to instantiate an object of an abstract class causes a compilation error.
- Abstract Classes contain at least one Pure Virtual function in it.
- Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

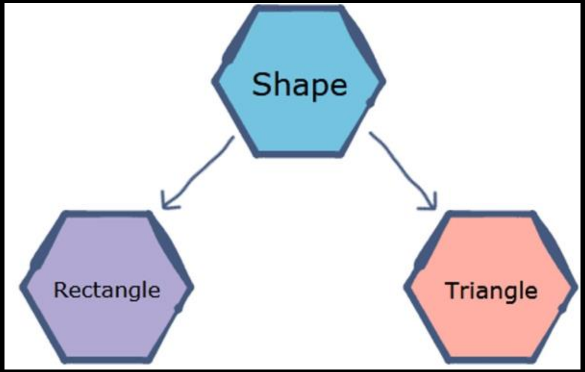
Characteristics of an Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Abstract Class - Example

```
class Shape {
   public:
     // Pure virtual function declared
      virtual int Area() = 0;
      // Function to set width.
      void setWidth(int w) {
         width = w;
      }
      // Function to set height.
      void setHeight(int h) {
         height = h;
      }
   protected:
      int width;
      int height;
};
```

```
//Main function
int main()
{
 Rectangle R;  Triangle T; R.setWidth(5);
R.setHeight(10); T.setWidth(20);  T.setHeight(8);
cout << "The area of the rectangle is: " << R.Area();
cout << "The area of the triangle is: " << T.Area()
}
```



```
//A rectangle is a shape & it inherits shape.
class Rectangle: public Shape {
   public:
// The implementation for Area is specific to
// a rectangle.
      int Area() {
         return (width * height);
      }
};
```

```
// A triangle is a shape; it inherits shape.
class Triangle: public Shape {
   public:
//Triangle uses the same Area function but
//implements it to return area of a triangle.
      int Area() {
         return (width * height)/2;
      }
};
```

Some Special Functions and Classes

- Friend Function ✓
- Virtual Function ✓
- Interfaces
  - Pure Virtual Functions ✓
  - Abstract Class ✓

TASK!

- There is something known as a "Virtual Destructor".
- Find out what it is and write a short note (maximum 1 page) on its importance as well as how, where and when to use it.

Exercises

- Check out the Assignments Tab on AUMS

Quick Summary

- Basics of C++
- Pointers in C++
- Memory Management in C++
- OOP Concepts in C++
  - Classes
  - Objects
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- Friend Function
- Virtual Function
- Interfaces
- Abstract Class
- Examples
- Exercises

Up Next

A major test on everything we have covered in the course.