Pointers, Virtual Functions, Abstract classes

”Virtual functions, Abstract classes, Friend functions and classes, Static functions”

Fundamentals of OOPs

Shakirullah Waseeb
shakir.waseeb@gmail.com

Nangarhar University

November 8, 2017

Agenda

1. Virtual Functions
2. Late Binding
3. Abstract Classes
4. Friend Function
5. Static Function
6. Questions and Discussion

Virtual Functions

- **Virtual:** existing in appearance but not in reality
- **Virtual Function:** a program that appears to be calling a function of *one class* may in reality be calling a function of a *different class*
- Why are virtual functions?
  - suppose a particular operation on objects of different classes
  - calculating net salary for objects of different types of employees (Fixed Term, Project Based, Hour Based etc)
  - drawing different types of objects (triangle, rectangle, circle etc)
- This is called **polymorphism**; *means different forms* which requires some conditions to that must be met
  - all the different classes (e.g. fixed-term, project-based, and hour-based) must be descended from a single base class (in this case employee)
  - the calculate_net_salary() function must be declared virtual

Ground-Up the Basics

- Accessing member functions (not-virtual) with pointers

Example

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

class A {
     public:
          void  disp() {
               cout « "I am base class";
          }
};

class D1: public A {
     public:
          void  disp() {
               cout « " I am derived class D1";
          }
};

class D2: public A {
     public:
          void  disp() {
               cout « " I am derived class D2";
          }
};
```

```
int main() {
  A*  ptrA;
  D1 der1;
  D2 der2;
  ptrA = &der1;
  ptrA − > disp();

  ptrA = &der2;
  ptrA − > disp();
};
```

Output

```
   I am base class
I am base class
```

Ground-Up the Basics

- Accessing member functions (with-virtual) with pointers

Example

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

class A {
    public:
        virtual void  disp() {
            cout « "I am base class”;
        }
};

class D1: public A {
    public:
        void  disp() {
            cout « "I am derived class D1”;
        }
};

class D2: public A {
    public:
        void  disp() {
            cout « "I am derived class D2”;
        }
};
```

```
int main() {
   A*  ptrA;
   D1 der1;
   D2 der2;
   ptrA = &der1;
   ptrA − > disp();

   ptrA = &der2;
   ptrA − > disp();
};
```

Output

```
I am derived class D1
I am derived class D2
```

Late Binding

- The compiler always call the base class function in case of non-virtual function example (disp() in base class) via **ptrA**
- While in case of virtual function in base class example (*virtual* disp() in base class) the compiler doesn't know what class the contents of **ptrA** may contain (D1 or D2)
- Which version of the disp() function should be called?
- Known upon running of the program, and known what class is pointed to by **ptrA** the appropriate version of disp() will be called
- This is known is *late binding* or *dynamic binding*
- Choosing at compilation time is called *early binding* or *static binding*

Abstract Classes

- When a function is defined in the base class with following syntax
  virtual void disp()=0;
- Then it is known as **pure virtual function**
- When any pure virtual function appears in a class, then it is know as **abstract class**; which means this class is no more available to be instantiated
- We can't create any object of an abstract class

Friend Function

- Assignment!

Static Function

- Assignment!

Your Turn: Time to hear from you!

[1]

[1] https://fensafitters.files.wordpress.com/2013/07/3d095.jpg

References

Robert Lafore
Object-Oriented Programming in C++, 4th Edition .
2002.

Piyush Kumar
Object oriented Programming (Using C++)
http://www.compgeom.com/ piyush/teach/3330