Chapter 7: Polymorphism

Department of Computer Science and Engineering
Kathmandu University

Instructor: Rajani Chulyadyo, PhD

Contents

Virtual function

Virtual means existing in appearance but not in reality.

When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.

A virtual function is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class.

Why virtual function?

Suppose you have **a number of objects of different classes** but you want to put them all in an array and **perform a particular operation on them using the same function call.**

For example, suppose you have Rectangle objects and Triangle objects (from the previous example) in an array and you want to get their area.

With the help of virtual functions, you will be able to do the following

```
Polygon* polygons[n];
// Initialize the objects
// ...
for (Polygon *p : polygons) {
        std::cout << "The area is " << p->area() << std::endl;
}
```

Pure virtual functions

- A pure virtual function (or abstract function) a special kind of virtual function that has no body at all!
- A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
- To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

```
class Polygon
{
public:
   Polygon() {}
   virtual double area() = 0;
};
```

Pure virtual functions

A pure virtual function indicates that “it is up to the derived classes to implement this function”.

A pure virtual function is useful

- when we have a function that we want to put in the base class, but only the derived classes know what it should return.
- we want our base class to provide a default implementation for a function, but still force any derived classes to provide their own implementation.

Pure virtual functions, Abstract base classes

Using a pure virtual function has two main consequences:

1. Any class with one or more pure virtual functions becomes an **abstract base class**, which means that it **cannot be instantiated!**
2. Any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

Interface class

An interface class is a class that has no member variables, and where all of the functions are pure virtual.

Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Interface class: Example

Suppose we want to implement a Last-In-First-Out (LIFO) list, called Stack, with three functions: push (adding an element), pop (removing the last inserted element, and top (peeking the last inserted element without removing it).

It can be implemented by storing the data contiguously (in an array) or non-contiguously (in a linked list - not covered here).

We want the client code to be least affected when we change the details of the implementation (i.e., changing the contiguous storage to non-contiguous one).

Interface class: Example

In this case, we can define an interface class that defines the behaviours / functionalities of the LIFO list.

```
class Stack {
public:
   virtual ~Stack() {}

   virtual bool push(const int element) = 0;
   virtual bool pop(int &element) = 0;
   virtual bool top(int &element) const = 0;
};
```

Interface class: Example

The details are then implemented in a derived class.

```
class ArrayStack : public Stack
{
private:
   int *data;
   int topIndex;
   int size;
public:
   ArrayStack(int size);

   virtual bool push(const int element);
   virtual bool pop(int &element);
   virtual bool top(int &element) const;
};
```

```
ArrayStack::ArrayStack(int size)
   : size(size), topIndex(-1),
     data(new int[size]) {}

bool ArrayStack::push(const int element){
   if (topIndex < size - 1){
       topIndex++;
       data[topIndex] = element;
       return true;
   } else {
       return false;
   }
}
```

Interface class: Example

```
bool ArrayStack::top(int &element) const{
    if (topIndex < 0) {
        return false;
    } else {
        element = data[topIndex];
        return true;
    }
}
bool ArrayStack::pop(int &element) {
    if (top(element)) {
        topIndex--;
        return true;
    } else {
        return false;
    }
}
```

```
int main()
{
    Stack *s = new ArrayStack(10);
    s->push(10);
    s->push(9);

    int element;

    s->top(element);
    std::cout << "Top element is " << element;

    s->pop(element);
    std::cout << "Popped element is "
              << element;
}
```

Virtual destructors

We should always make your destructors virtual if we are dealing with inheritance.

If a base class pointer that is pointing to a derived object is deleted, only the base class part of the derived object is deleted unless the base class destructor is not marked as virtual, thereby leading to memory leakage.

Dynamic casting

A base pointer can point to any of the derived object, but the reverse is not true.

To convert base-class pointers into derived-class pointers, we use a casting operator named **dynamic_cast**.

Dynamic casting is needed when we need access to something that is derived-class specific (e.g. a function that only exists in the derived class).

Dynamic cast: Example

```
class Polygon
{
public:
   virtual double area() const
   {
       return 0;
   }
};
```

```
class Rectangle : public Polygon{
protected:
   double length, width;
public:
   Rectangle(double length = 0, double width = 0)
       : length(length), width(width) {}

   double area() const { return length * width; }

   // An additional method not available in Polygon
   double perimeter() const{
       return 2 * (length + width);
   }
};
```

Dynamic cast: Example

```
int main()
{
   Polygon *p = new Rectangle(20, 10);
   std::cout << p->area() << std::endl;

   // Print the perimeter of the rectangle?
   // p->perimeter() won't work

   // So, first convert p to Rectangle pointer, and then call perimeter()
   Rectangle *r = dynamic_cast<Rectangle *>(p);
   std::cout << "Perimeter = " << r->perimeter() << std::endl;
}
```

Lab 6

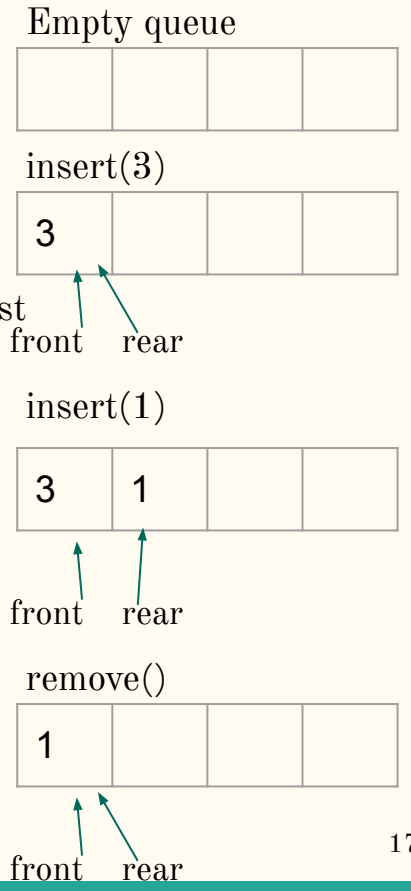
Question 1

We need to implement a First-In-First-Out (FIFO) list, called Queue, with four functions: insert (adding an element), remove (removing the first inserted element), front (peeking the first inserted element without removing it) and rear (peeking the last inserted element without removing it).

Like Stack, it can be implemented by storing the data contiguously or non-contiguously.

Create an interface class IQueue with the functionalities mentioned above. Create a class ArrayQueue that inherits IQueue and stores the data elements in an array.

The insert() method must throw an exception if the queue is full. Similarly, remove(), front() and rear() must throw an exception if the queue empty.



References

1. https://www.learncpp.com/cpp-tutorial/122-virtual-functions/
2. https://www.learncpp.com/cpp-tutorial/126-pure-virtual-functions-abstract-base-classes-and-interface-classes/