

Chapter 7: Polymorphism

Department of Computer Science and Engineering
Kathmandu University

Instructor: Rajani Chulyadyo, PhD

Contents

- Introduction
- Pointers to objects
- Pointers to derived classes
- Virtual functions
- Pure virtual functions



Polymorphism

- Poly = many
- Morph = forms
- Polymorphism: Ability to take more than one form. Processing objects differently based on their data type.
- Examples:
 - Function overloading
 - Operator overloading
 - Constructor overloading

Types of polymorphism

- **Compile-time polymorphism** refers to forms of polymorphism that are resolved by the compiler.
 - These include function overload resolution, as well as template resolution.
- **Runtime polymorphism** refers to forms of polymorphism that are resolved at runtime.
 - This includes virtual function resolution.

Binding

When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language. Each line of machine language is given its own unique sequential address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into addresses.

Direct function calls can be resolved using a process known as **early binding** whereas *indirect function calls* (e.g., calling a function via a function pointer) can be resolved using a process known as **late binding**.

Early binding (Static binding)

If the compiler knows at the compile-time which function is called, it is called **early binding** or **static binding**.

When the compiler encounters a direct function call in the program, it replaces that function call with a machine language instruction to go to that function.

Late binding (Dynamic binding)

If a compiler does not know at compile-time which functions to call up until the run-time, it is called **late binding** or **dynamic binding**.

Late binding occurs when we make implicit or indirect function calls (e.g., using function pointers or virtual functions) in our program.


Pointer (Recall)

- A variable that stores the memory address as its value.
- Is created with the * operator

```
int a = 5;
```

```
int* ptr = &a; // Assign the address of a to the pointer
```

Address-of operator



a			
	5		
3241	3245	3249	

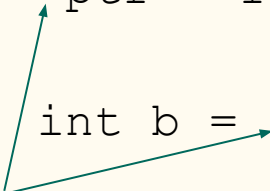
ptr			
	3245		

Pointer (Recall)

- Pointers are said to "point to" the variable whose address they store.
- Pointers can be used to directly access the variable they point to using the dereference operator (*).

```
int a = 5;
int* ptr = &a; // ptr points to a
*ptr = 10;      // a's value will now be 10
int b = *ptr;    // A new variable b will be created.
                // a's value will be copied to b
```

Dereference operator



Pointer (Recall)

Pointer initialization

```
int var;  
int* ptr = &var;  
  
int a;  
int* ptr1;    // It is an uninitialized pointer  
ptr1 = &var;  // Now it is initialized to point to var  
  
int* ptr2 = nullptr; // A null pointer points to nowhere  
  
int* ptr3 = 0;      // Null pointer.
```

Pointer (Recall)

Pointers and dynamic memory allocation

Dynamic memory is allocated using operator `new` followed by a data type specifier.

```
int* ptr = new int(2);
```

It creates and initializes objects with dynamic storage duration, i.e., objects whose lifetime is not limited by the scope in which they were created.

It returns a pointer to the beginning of the new block of memory allocated.

Pointer (Recall)

Pointers and dynamic memory allocation

- Once the memory allocated using operator `new` is no longer needed, it can be freed using operator `delete` so that the memory becomes available.

```
delete ptr;
```

Pointers (Recall)

Pointer to an array

```
int a[3] = {3, 4, 5 };  
  
int *ptr = a; // Pointer to an array (points to the first element of a)  
  
  
int (*a)[5]; // Pointer to an array of five numbers  
  
int b[5] = { 1, 2, 3, 4, 5 };  
  
a = &b; // Points to the whole array b
```

Pointers (Recall)

Array of pointers

```
int *var_name[array_size];
```

Pointers (Recall)

Pointers to pointers

```
int **var;
```

Usage: arrays of pointers, two-dimensional dynamically allocated arrays

Pointers (Recall)

Pointer to a function

- A function pointer is a variable that holds the address of a function.
- Syntax for creating a function pointer:

```
// fcnPtr1 is a pointer to a function that takes no arguments and  
returns  
// an integer  
int (*fcnPtr1)();
```

```
// fcnPtr2 is a pointer to a function that takes an argument of type  
// double and returns an integer  
int (*fcnPtr2)(double);
```


Pointers (Recall)

Pointer to a function

```
#include <iostream>

int func1(int a) { return a/2; }
int func2(int a) { return a*2; }

int main() {
    int (*F)(int);
    F = &func1;
    std::cout << (*F)(100) << std::endl; // call func1 through F (explicit dereference)
    F = &func2;
    std::cout << F(100) << std::endl;    // call func2 through F (implicit dereference)
}
```

Pointers (Recall)

Pointer to a function

One of the most useful things to do with function pointers is *pass a function as an argument to another function*.

Functions used as arguments to another function are sometimes called **callback functions**.

This allows a C++ program to select a function dynamically at a run time.

Pointers (Recall)

Callback function

```
#include <utility> // for std::swap
#include <iostream>

bool asc(int a, int b) { return a > b; }
bool desc(int a, int b) { return a < b; }

void sortTwoNumbers(int &a, int &b,
    bool (*comparisonFcn)(int, int))
{
    if (comparisonFcn(a, b)) {
        std::swap(a, b);
    }
}
```

```
int main() {
    int a = 10, b = 2;

    sortTwoNumbers(a, b, asc);
    std::cout << "a = " << a <<
        ", b = " << b << "\n";

    sortTwoNumbers(a, b, desc);
    std::cout << "a = " << a <<
        ", b = " << b << "\n";
}
```

Pointers to objects

It is perfectly valid to create pointers that point to classes.

```
class Polygon
{
public:
    int num_sides;
    ...
};

int main()
{
    Polygon *p;
}
```

Pointers to objects

Use of the member selection operator (.) doesn't work if you have a pointer to an object.

To select a member from a pointer to an object, we can use the **member selection from pointer operator (->)**, which is equivalent to dereferencing the pointer to get the object and then selecting the member.

```
class Polygon
{
public:
    int num_sides;
    ...
};

int main()
{
    Polygon *p = new Polygon();
    // Equivalent to (*p).num_sides = 3
    p->num_sides = 3;
}
```

Pointers to derived class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class.

```
class Base
{...};

class Derived : public Base
{...};

int main()
{
    Base b1, *p;
    Derived d1;

    p = &b1; // valid
    p = &d1; // also valid
}
```

Pointers to derived class

What will be the output of this program?

```
#include <iostream>

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

class Triangle : public Polygon {
protected:
    double base, height;

public:
    Triangle(double base = 0, double height = 0)
        : base(base), height(height) {}
    double area() { return height * base / 2; }
};
```

```
int main()
{
    Polygon *p;
    Triangle t(20, 11.5);
    std::cout << t.area() << "\n";

    p = &t;
    std::cout << p->area() << "\n";
}
```

Pointers to derived class

Because `p` is a pointer to the base class (`Polygon`), it can only see the members of the `Polygon` class.

Even though `Triangle::area()` shadows (hides) `Polygon::area()` for `Triangle` objects, the `Polygon` pointer/reference cannot see `Triangle::area()`.

```
int main()
{
    Polygon *p;
    Triangle t(20, 11.5);
    std::cout << t.area() << "\n"; // 115

    p = &t;
    std::cout << p->area() << "\n"; // 0
}
```


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.

In order to declare a member function of a class as virtual, we must precede its declaration with the keyword `virtual`.

Virtual function

When `Polygon::area()` is virtual, `p->area()` will resolve to `Triangle::area()`.

```
#include <iostream>

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

class Triangle : public Polygon {
protected:
    double base, height;
public:
    Triangle(double base = 0, double height = 0)
        : base(base), height(height) {}
    virtual double area() {
        return height * base / 2;
    }
};
```

```
int main()
{
    Polygon *p;
    Triangle t(20, 11.5);
    std::cout << t.area() << "\n"; // 115

    p = &t;
    std::cout << p->area() << "\n"; // 115
}
```

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) is 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.

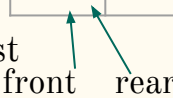
Create an interface class IQueue with the functionalities mentioned above. Create a class ArrayQueue that inherits IQueue and stores the data elements in an array.

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.

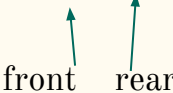
Empty queue



insert(3)



insert(1)



remove()



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/>
3. Lafore, R. Object Oriented Programming in C++. Sams Publishing.