Software Development

Part E - Polymorphism

Abstract Base Classes

Design polymorphic objects to amplify the reusability of code
Introduce pure virtual functions
Demonstrate a unit test of an interface

"Program to an interface, not an implementation"
(Gamma, Helm, Johnson, Vlissides, 1994).

Object-oriented languages use interfaces to define the single identifier to multiple meanings that polymorphism provides. Separating the interface from its various implementations promotes low coupling between the client code and an object's class hierarchy. The interface specifies what any object in the hierarchy offers to a client, while each implementation specifies how the interface provides what it has offered to its clients. This separation of concerns is central to software engineering. The interface effectively hides the hierarchy from its clients. We can upgrade the hierarchy by adding derived classes without having to change the client code. We can upgrade the client code without having to change the hierarchy.

C++ supports the distinction between an interface and its implementations through abstract and concrete classes. An abstract class is a base class that defines an interface, while a concrete class is a derived class that implements that interface. The abstract class identifies the member functions that the class hierarchy exposes to its clients and is the gateway to testing the derived classes in its own inheritance hierarchy. Each concrete class gives a specific meaning to the interface.

This chapter describes pure virtual functions, which are the principal components of an abstract base class. The chapter shows how to define an abstract class and use it with an array of pointers in client code. This chapter concludes with an example of a unit test on an abstract base class.

PURE VIRTUAL FUNCTION

The principal component of an abstract base class is a *pure virtual member function*. *Pure* refers to the lack of any implementation detail. That is, a pure virtual function is a signature without a definition. The client code only requires access to the signature.

Declaration

The declaration of a pure virtual function takes the form

```
virtual Type identifier(parameters) = 0;
```

The assignment to 0 identifies the virtual function as pure. A pure function must be a virtual member function.

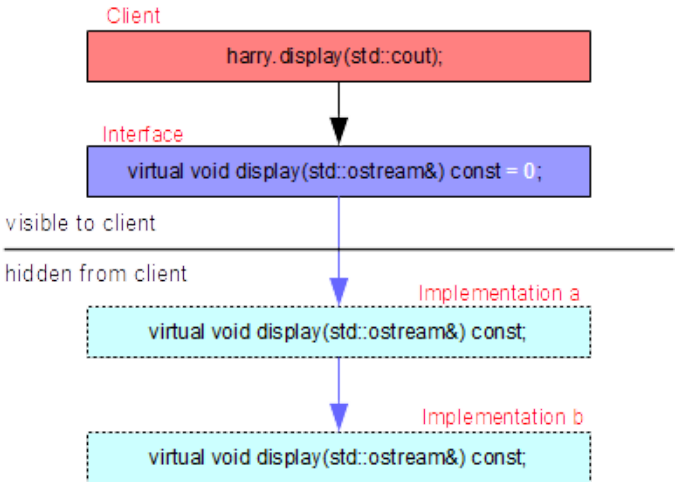
Example

We define the pure virtual function for the signature `display(std::ostream&) const` using

```
virtual void display(std::ostream&) const = 0;
```

Implementations

A pure virtual member function typically has multiple definitions within its hierarchy. Each definition has the same signature as the pure virtual function but a different meaning. That is, it provides the client with the implementation that suits a specific dynamic type.



Note that the separation between the client and the hierarchy's implementation is crisp. The client code does not need any access to the variety of implementations available within the hierarchy. The implementation code has no access to the client codes that use the hierarchy.

ABSTRACT CLASSES

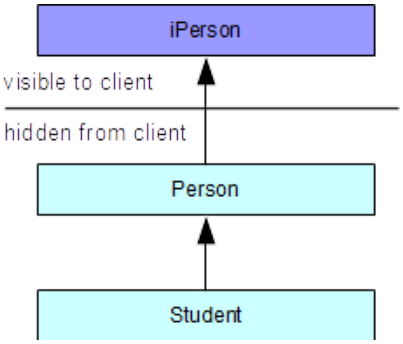
An abstract class is a class that contains or inherits a pure virtual function. Because the class provides no implementation(s) for its pure virtual function(s), the compiler cannot instantiate the class. Any attempt to create an instance of an abstract base class generates a compiler error.

Definition

The definition of any abstract base class contains or inherits at least one pure virtual member function. The class definition contains the declaration of the pure virtual function. We call an abstract base class without any data members a *pure interface*.

Example

Let us define an abstract base class named **`iPerson`** for our **`Person`** hierarchy and use this class to expose the hierarchy's **`display()`** function to any client code.



The **`iPerson.h`** header file defines our abstract class:

```
// Abstract Base Class for the Person Hierarchy
// iPerson.h

#include <iostream>

class iPerson {
  public:
    virtual void display(std::ostream&) const = 0;
};
```

We derive our **`Person`** hierarchy from this interface. The header file that defines our **`Person`** and **`Student`** class includes the header file that defines our abstract base class:

```
// Late Binding
```

```
// Student.h

#include <iostream>
#include "iPerson.h"
const int NC = 30;
const int NG = 20;

class Person : public iPerson {
    char name[NC+1];
  public:
    Person();
    Person(const char*);
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    float grade[NG];
    int ng;
  public:
    Student();
    Student(int);
    Student(const char*, int, const float*, int);
    void display(std::ostream&) const;
};
```

The class definitions for the **Person** and **Student** classes inform the compiler that each concrete class implements its own version of the **display()** member function.

ARRAY OF POINTERS

A systematic technique for accessing objects of different dynamic type within the same hierarchy is through an array of pointers of their static type. The executable code dereferences each pointer at run time based on its object's dynamic type.

Person Example

The following code demonstrates the use of an array of pointers to **Person** objects. The objects pointed to by the array elements may be of differing dynamic type, but are of the same static type. The **CreatePerson()** global function creates a **Person** object and returns its address.

Client Code

The following client code manages **Person** objects through the array of pointers **p**. The output generated for the input provided is listed on the right:

```
// Array of Pointers
// array_of_pointers.cpp

#include <iostream>
#include "iPerson.h"

const int NP = 5;

int main() {
    iPerson* p[NP];
    for (int i = 0; i < NP; i++)
        p[i] = nullptr;

    int n = 0;
    bool quit = false;
    do {
        iPerson* ptemp = CreatePerson();
        if (ptemp != nullptr) {
            p[n++] = ptemp;
        } else {
            quit = true;
        }
    } while(n < NP && !quit);

    for (int j = 0; j < n; j++) {
```

```
Type (0,1,2) : 1
Name: Jane Doe
Type (0,1,2) : 2
Name: Harry
Student Number : 1234
Number of Grades : 3
Grade 1 : 45.6
Grade 2 : 67.8
Grade 3 : 89.5
Type (0,1,2) : 0
Jane Doe
```

```
        p[j]->display(std::cout);
        std::cout << std::endl;
    }

    for (int j = 0; j < n; j++)
        delete p[j];
}
```

```
Harry 1234:
 45.60
 67.80
 89.50
```

Interface

The interface to a **Person** object includes the prototype for a global function that creates the object:

```
// Abstract Base Class for the Person Hierarchy
// iPerson.h

#include <iostream>

class iPerson {
  public:
    virtual void display(std::ostream&) const = 0;
};

iPerson* CreatePerson();
```

Concrete Class Definitions

The concrete class definitions specify the various implementations of the **iPerson** interface:

```
// Person and Student Concrete Classes
// Student.h

#include "iPerson.h"
const int NC = 30;
const int NG = 20;

class Person : public iPerson {
    char name[NC+1];
  public:
    Person();
    Person(const char*);
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    float grade[NG];
    int ng;
  public:
    Student();
    Student(int);
    Student(const char*, int, const float*, int);
    void display(std::ostream&) const;
};

iPerson* CreatePerson();
```

The prototype for **CreatePerson()** is repeated for documentation.

Implementations

Each concrete class that declares the **display()** member function in its definition defines its own version of the **display()** function. The implementation file also defines the global **CreatePerson()** function:

```
// Person Hierarchy - Implementation
// person.cpp

#include <iostream>
#include <cstring>
#include "Student.h"
```

```
Person::Person() {
    name[0] = '\0';
}

Person::Person(const char* nm) {
    std::strncpy(name, nm, NC);
    name[NC] = '\0';
}

void Person::display(std::ostream& os) const {
    os << name << ' ';
}

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int n) {
    *this = Student("", n, nullptr, 0);
}

Student::Student(const char* nm, int sn, const float* g, int ng_) : Person(nm) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::display(std::ostream& os) const {
    if (no > 0) {
        Person::display(os);
        os << no << ":\n";
        os.setf(std::ios::fixed);
        os.precision(2);
        for (int i = 0; i < ng; i++) {
            os.width(6);
            os << grade[i] << std::endl;
        }
        os.unsetf(std::ios::fixed);
        os.precision(6);
    } else {
        os << "no data available" << std::endl;
    }
}

iPerson* CreatePerson() {
    iPerson* p = nullptr;
    int type, no, ng;
    float grade[NG];
    char name[NC+1];
    bool repeat;

    do {
        std::cout << "Type (0,1,2) : ";
        std::cin >> type;
        std::cin.ignore();
        repeat = false;
        switch(type) {
            case 0:
                break;
            case 1:
                std::cout << "Name: ";
                std::cin.getline(name, NC+1);
```

```
                p = new Person(name);
                break;
            case 2:
                std::cout << "Name: ";
                std::cin.getline(name, NC+1);
                std::cout << "Student Number : ";
                std::cin >> no;
                std::cout << "Number of Grades : ";
                std::cin >> ng;
                if (ng > NG) ng = NG;
                for (int i = 0; i < ng; i++) {
                    std::cout << "Grade " << i + 1 << " : ";
                    std::cin >> grade[i];
                    std::cin.ignore();
                }
                p = new Student(name, no, grade, ng);
                break;
            default:
                repeat = true;
                std::cout << "Invalid type. Try again\n";
        }
    } while(repeat);

    return p;
}
```

Multiple definitions would be unnecessary if the definitions of `display()` were identical.

UNIT TESTS ON AN INTERFACE

Good programming practice suggests that we code unit tests for an interface rather than a specific implementation. This practice requires that the interface does not change during the life cycle of the software. With a constant interface we can perform unit tests at every upgrade throughout an object's lifecycle without changing the test code.

Example

To illustrate a unit test on the interface of a hierarchy, consider a module of `Sorter` classes. The number of implementations changes throughout the life cycle.

The `Sorter` module contains all of the implemented algorithms. The interface and the tester module remain unchanged. With every upgrade to the `Sorter` module, we execute the unit tester on the interface.

Sorter Module

Each `Sorter` class sorts in a different way. The interface to the hierarchy exposes the `sort()` and `name()` member functions of each class.

The header file for the interface contains:

```
// Sorter Interface
// iSorter.h

class iSorter {
  public:
    virtual void sort(float*, int) = 0;
    virtual const char* name() const = 0;
};

iSorter* CreateSorter(int);
int noOfSorters();
```

The header file for the `Sorter` concrete classes contains:

```
// Sorter Concrete Classes
// Sorter.h

#include "iSorter.h"
```

```
class SelectionSorter : public iSorter {
  public:
    void sort(float*, int);
    const char* name() const;
};

class BubbleSorter : public iSorter {
  public:
    void sort(float*, int);
    const char* name() const;
};

iSorter* CreateSorter(int);
int noOfSorters();
```

The implementation file for the **Sorter** module defines the **sort()** and **name()** member functions for the **SelectionSorter** class and the **BubbleSorter** class as well as the global **CreateSorter()** and **noOfSorters()** functions:

```
// Sorter Hierarchy - Implementation
// Sorter.cpp

#include "Sorter.h"

void SelectionSorter::sort(float* a, int n) {
    int i, j, max;
    float temp;

    for (i = 0; i < n - 1; i++) {
        max = i;
        for (j = i + 1; j < n; j++)
            if (a[max] > a[j])
                max = j;
        temp = a[max];
        a[max] = a[i];
        a[i] = temp;
    }
}

const char* SelectionSorter::name() const {
    return "Selection Sorter";
}

void BubbleSorter::sort(float* a, int n) {
    int i, j;
    float temp;

    for (i = n - 1; i > 0; i--) {
        for (j = 0; j < i; j++) {
            if (a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

const char* BubbleSorter::name() const {
    return "Bubble Sorter";
}

iSorter* CreateSorter(int i) {
    iSorter* sorter = nullptr;
    switch (i) {
        case 0:
            sorter = new SelectionSorter();
            break;
        case 1:
            sorter = new BubbleSorter();
            break;
    }
```

```
    return sorter;
}

int noOfSorters() {
    return 2;
}
```

Unit Tester

The unit tester generates the test data and reports the test results:

```
// Test Main for the iSorter Interface
// Test_Main.cpp

#include <iostream>
#include <ctime>
#include "iSorter.h"

void populate(float* a, int n) {
    srand(time(nullptr));
    float f = 1.0f / RAND_MAX;
    for (int i = 0; i < n; i++)
        a[i] = rand() * f;
}

void test(iSorter* sorter, float* a, int n) {
    sorter->sort(a, n);
    bool sorted = true;
    for (int i = 0; i < n - 1; i++)
        if (a[i] > a[i+1]) sorted = false;
    if (sorted)
        std::cout << sorter->name()
         << " is sorted" << std::endl;
    else
        std::cout << sorter->name()
         << " is not sorted" << std::endl;
}

int main() {
    int n;
    std::cout << "Enter no of elements : ";
    std::cin >> n;
    float* array = new float[n];

    for (int i = 0; i < noOfSorters(); i++) {
        iSorter* sorter = CreateSorter(i);
        populate(array, n);
        test(sorter, array, n);
        delete sorter;
    }

    delete [] array;
}
```

```
Enter no of elements : 200
Selection Sorter is sorted
Bubble Sorter is sorted
```

Note that we do not need to change this test code if we derive another class from the `iSorter` interface.

SUMMARY

- a pure virtual function is a member function declaration without an implementation
- an abstract base class contains or inherits at least one pure virtual function
- an interface is an abstract base class with no data members
- good programming practice performs unit tests on an interface rather than any specific implementation

EXERCISES

- Complete the Handout on Abstract Base Classes
- Complete the Workshop on Pure Virtual Functions

print this page

Top

Previous: Virtual Functions

Next: Templates

ICT | Home | Outline | Timeline

Notes | IPC Notes | MySeneca | Workshops | Assignments | Instructor

Designed by Chris Szalwinski

Copying From This Site

Last Modified: 05/20/2017 11:50