南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Lab 12

CONTENTS

- Learn **operator overloading**
- Learn **Friend functions**
- Learn how to overload the **<< operator** for output
- Learn the conversion of class
- Learn smart pointers

2 Knowledge Points

2.1 Operator overloading and Copy Constructor

2.2 Friend functions

2.3 Overloading the << operator for output

2.4 Conversion of class

2.5 Smart pointers

2.1 Operator Overloading

In C++, the overloading principle applies **not only to functions**, **but to operator**. Operators can be extended to work **not just with built-in types**, **but also classes**.

Overloaded operators are functions with special names: the keyword ***operator*** followed by the **symbol for the operator being defined**. Like any other function, an overloaded operator has a return type, a parameter list, and a body.

return_type operator op(argument-list)

op is the symbol for the operator being overloaded

An operator function must either be a member of a class or have at least one parameter of class type.

Copy Constructor

The **copy constructors** define what happens when an object is initialized from another object of the same type. It is used during initialization.

A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values. A copy constructor for a class normally has this prototype:

class_name (const class_name &);

It takes a constant reference to a class object as its argument.

When you do not define a copy constructor for a class, the compiler synthesizes one for you. Unlike the synthesized default constructor, a copy constructor is synthesized even if you define other constructors.

The default copy constructor performs a **member-to-member copy** of the non-static members (memberwise copying, also sometimes called ***shallow copying***). Each member is copied **by value**.

A **copy constructor** is invoked whenever a new object is created and initialized to an existing object of the same kind. The following four defining declarations invoke a copy constructor (Suppose the object c1 is already created.).

```
Complex c2 (c1);

Complex c3 = c1;

Complex c4 = Complex(c1);

Complex *pc = new Complex(c1);
```

This statement initializes a anonymous object to ***c1*** and assigns the address of the new object t the ***pc*** pointer.

A copy constructor is usually called in the following situations implicitly, so it should **not be explicit**:

1. When a class object is returned by value.
2. When an object is passed to a function as an argument and is passed by value.
3. When an object is constructed from another object of the same class.
4. When a temporary object is generated by the compiler.

using the + symbol to add two Complex objects

```cpp
//complex.h
#include <iostream>
#ifndef COMPLEX_H
#define COMPLEX_H


class Complex
{
private:
    double real;
    double imag;
public:
    Complex() : real(1), imag(1)
    {
        std::cout << "Default constructor is invoked.\n";
    }


    Complex(double re, double im) : real(re), imag(im)
    {
        std::cout << "Parameterized constructor is invoked.\n";
    }


    Complex(const Complex &); // prototype of the copy constructor


    Complex operator+(Complex rhs);


    ~Complex()
    {
        std::cout << "Destructor is invoked.\n";
    }
    void Show() const;
};
#endif
```

```cpp
//complex.cpp
#include <iostream>
#include "complex.h"

Complex::Complex(const Complex &c)
{
    real = c.real;
    imag = c.imag;
    std::cout << "Copy Constructor called." << std::endl;
}

Complex Complex::operator+(Complex rhs)
{
    this->real += rhs.real;
    this->imag += rhs.imag;
    return *this;
}

void Complex::Show() const
{
    std::cout << real << ((imag >= 0) ? "+" : "")
<< imag << "i" << std::endl;
}
```

Operator overloading works as a function

The types of return and parameter are both object not reference. Returning an object or passing by value of an object will invoke its copy constructor.

```cpp
// complexmain.cpp
#include <iostream>
#include "complex.h"

int main()
{
    Complex c1;
    Complex c2(1, 2);

    Complex c3 = c1 + c2;

    std::cout << "c1=";
    c1.Show();
    std::cout << "c2=";
    c2.Show();
    std::cout << "c3=";
    c3.Show();

    std::cout << "Done." << std::endl;

    return 0;
}
```

Operator overloading

The **left operand** is the invoking object, the **right operand** is the one parameter passed the argument.

Complex c = c1.operator+(c2) ;

Returning an object and passing by value of an object will invoke its copy constructor. Moreover, returning an object may create a temporary object, and then destroy it.

```
Default constructor is invoked.
Parameterized constructor is invoked.
Copy Constructor called.
Copy Constructor called.
Destructor is invoked.
c1=2+3i
c2=1+2i
c3=2+3i
Done.
Destructor is invoked.
Destructor is invoked.
Destructor is invoked.
```

The value of c1 is modified

Note: use passing by reference or returning reference whenever possible.

```cpp
Complex Complex::operator+(const Complex &rhs) const
{
    Complex result;
    result.real = real + rhs.real;
    result.imag = imag + rhs.imag;
    return result;
}
```

To avoid modifying the value of **this** object, a local object should be used.

You can return local object to the caller

```cpp
Complex& Complex::operator+(const Complex &rhs) const
{
    Complex result;

    result.real = real + rhs.real;
    result.imag = imag + rhs.imag;

    return result;
}
```

Do not return the reference of a local object, because when the function terminates, the reference would be a reference to a non-existent object.

```
complex.cpp: In member function ‘Complex& Complex::operator+(const Complex&) const’:
complex.cpp:23:12: warning: reference to local variable ‘result’ returned [-Wreturn-local-addr]
   23 |     return result;
      |            ^~~~~~
complex.cpp:20:13: note: declared here
   20 |     Complex result;
      |             ^~~~~~
```

Consider this case: compute the addition of a complex and a numeric number

```
Complex c3 = c1 + 2;
```

If an operator function is a member function, the first (left-hand) operand is the invoking object. So we can write another overloaded addition operator function with a double parameter as follows:

```
Complex operator+(double n) const;
```

The definition of the function is:

```
Complex Complex::operator+(double n) const
{
    Complex result;
    result.real = this->real + n;
    result.imag = this->imag;

    return result;
}
```

When a function returns an object, a temporary object will be created. It is invisible and does not appear in your source code. The temporary object is automatically destroyed when the function call terminates.

```
Complex Complex::operator+(double n) const
{
    double re = this->real + n;
    double im = this->imag;

    return Complex(re, im);
}
```

This return style is known as return constructor argument. By using this style instead of returning an object, the compiler can eliminate the cost of the temporary object. This even has a name: the ***return value optimization***.

How about the following case?

```
Complex c3 = 2 + c1;
```

The compiler can not find the correspond member function.

Conceptually, **2 + c1** should be the same as **c1 + 2** , but the first expression can not match any member function because 2 is not a Complex object.

Remember, **the left operand is the invoking object**, but 2 is not an object. So the compiler cannot replace the expression with a member function call.

In this case, only nonmember overloading operator function can be used. A nonmember function is not invoked by an object. But nonmember functions can’t directly access private data in a class. This time we use **friend function** to solve this problem.

2.2 Friend Function

If a function is defined as a **friend function** of a class, it has the same access privileges as a member function of the class. This means a friend function can access all the **private** and **protected** data of that class.

By using the keyword **friend** compiler knows the given function is a friend function.

Friend Function in C++

```
class ClassName
{
    .......
    // friend function declaration

    friend return_type functionName(parameter list);
};
```

The friend function prototype is preceded by keyword **friend,** and is declared in the class.

```
return_type functionName(parameter list)
{
    ....    /* private and protected data of
               ClassName can be accessed form
               this function because it is a
               friend function
            */
}
```

The function can be defined anywhere in the program like a normal C++ function. **The function definition does not use either the keyword friend or scope resolution operator.**

```cpp
//complex.h
#include <iostream>
#ifndef COMPLEX_H
#define COMPLEX_H


class Complex
{
private:
    double real;
    double imag;


public:
    Complex() : real(1), imag(1){}
    Complex(double re, double im) : real(re), imag(im){}

    Complex operator+(const Complex &rhs) const;
    Complex operator+(double n) const;


    void Show() const;

    friend Complex operator+(double n,Complex &rhs);
};


#endif
```

friend function declaration in Complex class definition

When defining a friend function, don't use the **Complex::** qualifier. Also you need not use the **friend** keyword in the definition.

```cpp
Complex operator+(double n, Complex &rhs)
{
    return Complex(n + rhs.real, rhs.imag);
}
```

or

```cpp
Complex operator+(double n, Complex &rhs)
{
    return rhs + n;
}
```

```cpp
// complexmain.cpp

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


int main()
{
    Complex c1;
    Complex c = 2 + c1;


    std::cout << 2;
    std::cout << " + ";
    c1.Show();


    std::cout << " = ";
    c.Show();
    std::cout << std::endl;


    std::cout << "Done." << std::endl;


    return 0;
}
```

With the nonmember overloaded operator function, the **left operand** of an operator expression corresponds to the first argument of the operator function, and the **right operand** corresponds to the second argument.

```cpp
Complex operator+(double n, Complex &rhs)
{
    return Complex(n + rhs.real, rhs.imag);
}
```

2.3 Overloading the << operator for output

One very useful feature of classes is that you can overload the **<< operator**, so that you can use it with **cout** to **display an object's contents**.

Suppose **a** is a **Complex object**, to display Complex values, we've been using:

a.Show();

```
void Complex::Show() const
{
    std::cout << real << ((imag >= 0) ? "+" : "") << imag << "i" << std::endl;
}
```

Can we use **cout << a;** to display Complex value?

The First Version of Overloading <<

If you use a **Complex** member function to overload **<<** , the **Complex** object would come first, the display's style is like **c << cout**; not cout << c;. So we choose to overload the operator by using a **friend function**:

```
friend void operator<<(std::ostream& os, Complex &rhs);
```

friend function declaration

```
void operator<<(std::ostream &os, Complex &rhs)
{
    os << rhs.real << ((rhs.imag >= 0) ? "+" : "") << rhs.imag << "i";
}
```

friend function definition

But the implementation doesn't allow you to combine the redefined **<< operator** with ones cout normally uses:

cout << a << "\n"; // can't do

The Second Version of Overloading <<

We revise the operator<<( ) function so that it returns a reference to an ostream object:

friend function declaration

```
friend std::ostream& operator<<(std::ostream& os, Complex &rhs);
```

```
std::ostream& operator<<(std::ostream &os, Complex &rhs)
{
    os << rhs.real << ((rhs.imag >= 0) ? "+" : "") << rhs.imag << "i";
    return os;
}
```

friend function definition

Ordinarily, the first parameter of an output operator is a reference to a nonconst ostream object. The ostream is nonconst because writing to the stream changes its state. The parameter is a reference because we cannot copy an ostream object.

The second parameter ordinarily should be a reference to const of the class type we want to print. The parameter is a reference to avoid copying the argument. It can be const because (ordinarily) printing an object does not change that object. To be consistent with other output operators, operator<< normally returns its ostream parameter.

Increment and decrement operators

Classes that define increment or decrement operators should define both the **prefix** and **postfix** versions. These operators usually should be defined as members because these operators change the state of the object.

```
return_type operator ++();

return_type operator  --();
```

Normal overloading cannot distinguish between the prefix and postfix operators. To solve this problem, the **postfix** versions take an **extra (unused) parameter of type int**. When we use a postfix operator, the compiler supplies 0 as the argument for this parameter.

```cpp
//rational.h
#pragma once
#include <iostream>

class Rational {
private:
    int numerator;
    int denominator;


public:
    Rational(int n = 0, int d = 1) : numerator(n), denominator(d) {}

    Rational& operator++()
    {
        this->numerator++;
        return *this;
    }

    Rational operator++(int)
    {
        Rational ret = *this;
        ++(*this);  // operator ++();
        return ret;
    }

    friend std::ostream& operator<<(std::ostream& os, const Rational& rhs)
    {
        os << rhs.numerator << "/" << rhs.denominator;
        return os;
    }
};
```

prefix version of operator++

postfix version of operator++

```cpp
//rational.cpp
#include <iostream>
#include "rational.h"

using namespace std;


int main() {
    Rational a = 10;
    Rational b(1, 2);


    cout << "a = " << a << ", ++a = " << ++a << endl;
    cout << "b = " << b << ", b++ = " << b++ << endl;


    return 0;
}
```

Result:

```
a = 10/1, ++a = 11/1
b = 1/2, b++ = 1/2
```

2.4 Conversion of class

2.4.1 Implicit Class-Type Conversions

Every constructor that can be called with **a single argument** defines an implicit conversion to a class type. Such constructors are sometimes referred to as ***converting constructors***.

```cpp
// circle.h
#include <iostream>

class Circle
{
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    Circle() : radius(1) {}

    double getArea() const;
    double getRadius() const;


    friend std::ostream &operator<<(std::ostream &os,
const Circle &c);
};
```

Converting constructor

```cpp
// circlemain.cpp
#include <iostream>
#include "circle.h"
int main()
{
    Circle r1;

    Circle r2 = 3;

    Circle r3(10);

    r3 = 4;

    std::cout << r1 << std::endl;
    std::cout << r2 << std::endl;
    std::cout << r3 << std::endl;
    return 0;
}
```

Convert int to Circle type

Convert int to Circle type

when we use the copy form of initialization (with an =), implicit conversions happens.

```
Radius = 1, Area = 3.14
Radius = 3, Area = 28.27
Radius = 4, Area = 50.27
```

```cpp
// rational.h
#pragma once
#include <iostream>

class Rational
{
private:
    int numerator;
    int denominator;

public:
    Rational(int n = 0, int d = 1) : numerator(n), denominator(d) {}

    int getN() const { return numerator; }
    int getD() const { return denominator; }

    friend std::ostream &operator<<(std::ostream &os, const Rational &rhs)
    {
        os << rhs.numerator << "/" << rhs.denominator;
        return os;
    }
};

const Rational operator*(const Rational &lhs, const Rational &rhs)
{
    return Rational(lhs.getN() * rhs.getN(), lhs.getD() * rhs.getD());
}
```

Constructor with default arguments works as a converting constructor.

We define the operator * as a normal function not a friend function of the Rational class.

```cpp
// rational.cpp
#include <iostream>
#include "rational.h"
using namespace std;

int main()
{
    Rational a = 10;
    Rational b(1, 2);

    Rational c = a * b;
    cout << "c = " << c << endl;

    Rational d = 2 * a;
    cout << "d = " << d << endl;

    Rational e = b * 3;
    cout << "e = " << e << endl;

    Rational f = 2 * 3;
    cout << "f = " << f << endl;

    return 0;
}
```

Convert int to Rational type

```
c = 10/2
d = 20/1
e = 3/2
f = 6/1
```

Use explicit to suppress the implicit conversion

We can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor as ***explicit***:

```
// rational.h
#pragma once
#include <iostream>

class Rational
{
private:
    int numerator;
    int denominator;
public:
    explicit Rational(int n = 0, int d = 1) : numerator(n), denominator(d)
{}
  ...
}
```

Turn off implicit conversion

```
int main()
{
        Rational a = 10;

        Rational a = (Rational)10;

        Rational a = static_cast<Rational>(10);
```

Can not do the implicit conversion

Use these two styles for explicit conversion

2.4.2 Conversion function

Conversion function is a member function with the name ***operator*** followed by a type specification, no return type, no arguments.

operator **typeName( );**

```cpp
class Rational
{
private:
    int numerator;
    int denominator;


public:
    Rational(int n = 0, int d = 1) :
numerator(n), denominator(d) {}


    int getN() const { return numerator; }
    int getD() const { return denominator; }


    operator double() const
    {
        return numerator / denominator;
    }
};
```

Conversion function

```cpp
Rational h(1, 2);
double g = 0.5 + h;
```

Convert Rational object **h** to **double** by conversion function

Declare a conversion operator as explicit for calling it explicitly

```cpp
explicit operator double() const
{
    return numerator / denominator;
}
```

```cpp
Rational h(1, 2);
double g = 0.5 + (double)h;
```

Caution: You should use implicit conversion functions with care. Often a function that can only be invoked explicitly is the best choice.

2.5 Smart Pointers

A **smart pointer** is a class object that acts like a regular pointer with the important feature that it automatically deletes the object to which it points. A smart pointer is a class template defined in the **std** namespace in the **<memory>** header file.

Each of these classes has an **explicit constructor** taking a pointer as an argument. Thus, there is no automatic type cast from a pointer to a smart pointer object.

```
int *p = new int(20);
std::unique_ptr<int> up = p;
```

Can not convert a regular pointer to a smart pointer implicitly.

```
std::unique_ptr<int> up = static_cast<std::unique_ptr<int>>(p);
```

Convert a regular pointer to a smart pointer explicitly.

2.5.1 Unique pointer

unique_ptr stores one pointer only. We can assign a different object by removing the current object from the pointer.

A unique_ptr does not share its pointer. It cannot be copied to another unique_ptr, passed by value to a function, or used in any C++ Standard Library algorithm that requires copies to be made. A unique_ptr can only be moved. This means that the ownership of the memory resource is transferred to another unique_ptr and the original unique_ptr no longer owns it.

A smart pointer is a class template that you declare on the stack, and initialize by using a raw pointer that points to a heap-allocated object. A unique pointer can be initialized with a pointer upon creation or with a raw pointer or by the **make_unique** helper function.

```cpp
#include <iostream>
#include <memory>
using std::cout, std::endl, std::string;


int main()
{
    int *p = new int(20);
    std::unique_ptr<int> up1(p);
    cout << "up1's content: " << *up1 << endl;


    std::unique_ptr<float> up2(new float(9.8f));
    cout << "up2's content: " << *up2 << endl;


    std::unique_ptr<string> up3(new string("Hello C++"));
    cout << "up3's content: " << *up3 << endl;


    std::unique_ptr<string> up4 = std::make_unique<string>("Hello World!");
    cout << "up4's content: " << *up4 << endl;


    std::unique_ptr<int[]> up5 = std::make_unique<int[]>(5);
    cout << "up5's contents: " << endl;
    for (int i = 0; i < 5; i++)
        cout << up5[i] << " ";
    cout << endl;


    double *pd = new double[3]{1, 2, 3};
    std::unique_ptr<double[]> up6(pd);
    cout << "up6's contents: " << endl;
    for (int i = 0; i < 3; i++)
        cout << up6[i] << " ";
    cout << endl;


    return 0;
}
```

Use **new** operator or **make_unique()** function to create unique_ptr. make_unique() is recommended.

```
up1's content: 20
up2's content: 9.8
up3's content: Hello C++
up4's content: Hello World!
up5's contents:
0 0 0 0 0
up6's contents:
1 2 3
```

You can also use a pointer to initialize a smart pointer unique_ptr

```cpp
std::unique_ptr<int> up7 = std::move(up1);
```

Use the **move** function to transfer the ownership from up1 to up7.
Is the assignment statement
unique_ptr<int> up7 = up1; OK? Why?

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

User-defined class

```cpp
class A
{
public:
    int a;
    A(int a) : a(a) { cout << "Constructor with data: " << a << endl;}
    ~A() { cout << "Destructor with data: " << a << endl; }
};

int main()
{
```

Initialize a unique_ptr with **new**

```cpp
    unique_ptr<A> up1(new A(1));
    cout << up1->a << endl;
```

Declare a unique_ptr and assign one later by **reset()**

```cpp
    unique_ptr<A> up2;
    up2.reset(new A(2));
    cout << up2->a << endl;
```

Initialize a unique_ptr by **make_unique**

```cpp
    unique_ptr<A> up3 = make_unique<A>(3);
    cout << up3->a << endl;
```

Initialize a unique_ptr with **a raw pointer**

```cpp
    A* pA = new A(4);
    unique_ptr<A> up4(pA);
    cout << up4->a << endl;

    return 0;
}
```

Result:

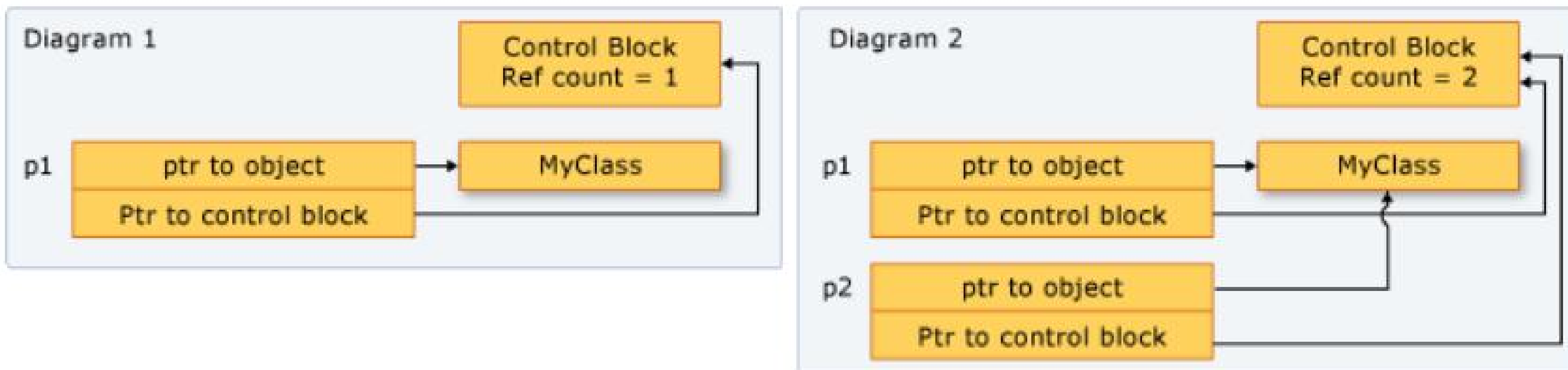
```
Constructor with data: 1
1
Constructor with data: 2
2
Constructor with data: 3
3
Constructor with data: 4
4
Destructor with data: 4
Destructor with data: 3
Destructor with data: 2
Destructor with data: 1
```

2.3.2 Shared pointer

The shared pointer is a reference counting smart pointer. By using ***shared_ptr*** more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** using ***use_count()* method.**

After you initialize a shared_ptr you can copy it, pass it by value in function arguments, and assign it to other shared_ptr instances. All the instances point to the same object, and share access to one "control block" that increments and decrements the reference count whenever a new shared_ptr is added, goes out of scope, or is reset. When the reference count reaches zero, the control block deletes the memory resource and itself.

The following illustration shows several shared_ptr instances that point to one memory location.



```cpp
int main()
{
    shared_ptr<A> sp1(new A(1));
    cout << "sp1->a = " << sp1->a << endl;

    shared_ptr<A> sp2 = make_shared<A>(2);
    cout << "sp2->a = " << sp2->a << endl;

    shared_ptr<A> sp3 = sp1;
    cout << "sp3->a = " << sp3->a << endl;

    cout << "the count of sp1: " << sp1.use_count() << endl;
    cout << "the count of sp2: " << sp2.use_count() << endl;
    cout << "the count of sp3: " << sp3.use_count() << endl;

    cout << "sp1 points to: " << sp1.get() << endl;
    cout << "sp2 points to: " << sp2.get() << endl;
    cout << "sp3 points to: " << sp3.get() << endl;

    sp2 = sp1;

    cout << "\nafter assign sp2 = sp1:" << endl;
    cout << "the count of sp1: " << sp1.use_count() << endl;
    cout << "the count of sp2: " << sp2.use_count() << endl;
    cout << "the count of sp3: " << sp3.use_count() << endl;

    cout << "sp1 points to: " << sp1.get() << endl;
    cout << "sp2 points to: " << sp2.get() << endl;
    cout << "sp3 points to: " << sp3.get() << endl;

    return 0;
}
```

Result:

```
Constructor with data: 1
sp1->a = 1
Constructor with data: 2
sp2->a = 2
sp3->a = 1
the count of sp1: 2
the count of sp2: 1
the count of sp3: 2
sp1 points to: 0x55c9dfc8deb0
sp2 points to: 0x55c9dfc8e310
sp3 points to: 0x55c9dfc8deb0
Destructor with data: 2

after assign sp2 = sp1:
the count of sp1: 3
the count of sp2: 3
the count of sp3: 3
sp1 points to: 0x55c9dfc8deb0
sp2 points to: 0x55c9dfc8deb0
sp3 points to: 0x55c9dfc8deb0
Destructor with data: 1
```

https://en.cppreference.com/w/cpp/memory/shared_ptr

```cpp
#include <iostream>
#include <string>
#include <memory>

int main()
{
    using namespace std;
    shared_ptr<string> films[5] =
    {
        shared_ptr<string>(new string("Fowl Balls")),
        shared_ptr<string>(new string("Duck Walks")),
        shared_ptr<string>(new string("Chicken Runs")),
        shared_ptr<string>(new string("Turkey Errors")),
        shared_ptr<string>(new string("Goose Eggs"))
    };


    shared_ptr<string> pwin;
    pwin = films[1]; // the counter of pwin and films[1]
is 2
    cout << "The nominees for best avian baseball film
are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";


    return 0;
}
```

An array of shared_ptr

Get the value of the object

Result:

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Chicken Runs
Turkey Errors
Goose Eggs
The winner is Duck Walks!
```

3 Exercises

1. Continue improving the Complex class and adding more operations for it, such as: **-, *, ~, ==, !=** etc. Make the following program run correctly.

```
#include <iostream>
#include "complex.h"
using namespace std;
int main()
{
    Complex a(3, 4);
    Complex b(2, 6);

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "~b = " << ~b << endl;
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "2 * a = " << 2 * a << endl;
    cout << "a * 2 = " << a * 2 << endl;

    Complex c = b;
    cout << "b == c? " << boolalpha << (b == c) << endl;
    cout << "b != c? " << (b != c) << endl;
    cout << "a == b? " << (a == b) << endl;

    Complex d;
    cout << "Enter a complex number (real part and imaginary part): ";
    cin >> d;
    cout << d << endl;
    return 0;
}
```

Note that you have to overload the **<< and >> operators**. Use const whenever warranted.

A sample runs might look like this:

Result:

```
a = 3+4i
b = 2+6i
~b = 2-6i
a + b = 5+10i
a - b = 1-2i
a * b = -18+26i
2 * a = 6+8i
a * 2 = 6+8i
b == c? true
b != c? false
a == b? false
Enter a complex number (real part and imaginary part): Enter real part: 3 -6
Enter imaginary part: 3-6i
```

2. Could the program be compiled successfully? Why? Modify the program until it passes the compilation. Then run the program. What will happen? Explain the result to the SA.

```
#include <iostream>
#include <memory>

using namespace std;

int main()
{
   double *p_reg = new double(5);
   shared_ptr<double> pd;
   pd = p_reg;
   cout << "*pd = " << *pd << endl;

   shared_ptr<double> pshared = p_reg;
   cout << "*pshred = " << *pshared << endl;

   string str("Hello World!");
   shared_ptr<string> pstr(&str);
   cout << "*pstr = " << *pstr << endl;

   return 0;
}
```