SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

Lab 11, dynamic memory in classes

廖琪梅，王大兴

Dynamic memory in classes

- Constructor, destructor, copy constructor and assignment operator
- Smart pointers

Four important member functions

To define a class containing **a pointer member**, you should think more carefully about four things: **constructor**, **destructor**, **copy constructor** and **assignment operator**.

In constructor, first, use **new** to allocate enough memory to hold the data where the pointer points to. Second, initialize the storage space with proper data.

In destructor, release the memory using **delete**.

With copy operations(**copy constructor** and **assignment operator**), we have two choices: one is hard copy(deep copy) and another is soft copy(shallow copy).

Hard copy

```
#pragma once

#include <iostream>
using namespace std;
class PtrHardcopy {
private:
    string* ps;
    int i;

public:
    PtrHardcopy(const string &s = string()):
        ps(new string(s)),i(0) {    }

    PtrHardcopy(const PtrHardcopy &p):
        ps(new string(*p.ps)), i(p.i) {    }

    PtrHardcopy& operator=(const PtrHardcopy&);

    ~PtrHardcopy() { delete ps; }

};
```

```
PtrHardcopy& PtrHardcopy::operator=(const PtrHardcopy& rhs)
{
    auto newp = new string(*rhs.ps);

    delete ps;

    ps = newp;
    i = rhs.i;

    return *this;
}
```

Assignment operators typically combine the actions of the destructor and the copy constructor. Like the destructor, assignment destroys the left-hand operand's resources. Like the copy constructor, assignment copies data from the right-hand operand.
Self-assignment(an object is assigned to itself) must be considered.

Constructor by initialization list, it dynamically allocates its own copy of that string and stores a pointer to that string in **ps**.

Copy constructor by initialization list, it also allocates its own, separate copy of the string.

Destructor frees the memory allocated in its constructors by executing delete on the pointer member, **ps**.

Soft copy

```cpp
#pragma once

#include <iostream>
using namespace std;
class PtrSoftcopy {
private:
    string* ps;
    int i;
    size_t* num;

public:
    PtrSoftcopy(const string& s = string()) :
        ps(new string(s)), i(0), num(new size_t(1)) {   }

    PtrSoftcopy(const PtrSoftcopy& p) :
        ps(p.ps), i(p.i), num(p.num) { ++*num;   }

    PtrSoftcopy& operator=(const PtrSoftcopy&);

    ~PtrSoftcopy()
    {
        if (-- * num == 0)
        {
            delete ps;
            delete num;
        }
    }
};
```

add a new data member named **num** that will keep track of how many objects share the same string.

The constructor that takes a string allocates this counter and initializes it to 1, indicating that there is one user of this object’s string member.

The copy constructor copies all three members from its given **PtrSoftcopy**. This constructor also increments the **num** member, indicating that there is another user for the string to which **ps** and **p.ps** point.

The destructor cannot unconditionally delete **ps**—there might be other objects pointing to that memory. Instead, the destructor decrements the reference count, indicating that one less object shares the string. If the counter goes to zero, then the destructor frees the memory to which both **ps** and **num** point.

```cpp
PtrSoftcopy& PtrSoftcopy::operator=(const PtrSoftcopy& rhs)
{
    ++*rhs.num;
    if (--*num == 0)
    {
        delete ps;
        delete num;
    }

    ps = rhs.ps;
    i = rhs.i;
    num = rhs.num;

    return *this;
}
```

The assignment operator must increment the counter of the right-hand operand and decrement the counter of the left-hand operand, deleting the memory used if appropriate. Also, as usual, the operator must handle self-assignment.

A copy constructor is usually called in the following situations:

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.

The following four definitions(constructing an object from another object) invoke a copy constructor:

```
Complex c1 (c2);

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.

Smart pointers

To make using dynamic memory easier (and safer), the C++ new library provides two smart pointer types(smart pointer template classes)**unique_ptr** and **shared_ptr** that manage dynamic objects.

A smart pointer acts like a regular pointer with the important exception 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.

Unique pointer

- The unique_ptr<> template holds a pointer to an object and deletes this object when the unique_ptr<> is deleted.
- Make sure that only exactly one copy of an object exists. Assignment operation of two unique_ptr<> is not allowed. However, it supports move semantics, where the pointer is moved from one unique_ptr<> to another.
- A unique pointer can be initialized with a pointer upon creation.

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

using namespace std;

int main()
{
    unique_ptr<int> up1(new int(9));
    cout << "up1's content:" << *up1 << endl;

    unique_ptr<string> up2(new string("Hello C++!"));
    cout << "up3's contents:" << *up2 << endl;

    unique_ptr<string> up3 = make_unique<string>("Hello world!");
    cout << "up3's contents:" << *up3 << endl;

    unique_ptr<int[]> up4 = make_unique<int[]>(5);
    cout << "up4's constents:" << endl;
    for (int i = 0; i < 5; i++)
        cout << up4[i] << "  ";
    cout << endl;

    float* p = new float[3]{ 1,2,3 };
    unique_ptr<float[]> up5(p);
    cout << "up5's constents:" << endl;
    for (int i = 0; i < 3; i++)
        cout << up5[i] << "  ";
    cout << endl;

    unique_ptr<int> up6 = move(up1);
    cout << "up6's content:" << *up6 << endl;

    return 0;
}
```

```
up1's content:9
up3's contents:Hello C++!
up3's contents:Hello world!
up4's constents:
0  0  0  0  0
up5's constents:
1  2  3
up6's content:9
```

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

You can use a pointer to initialize a unique_ptr

Use the **move** function to transfer the ownership from up1 to up6.
Is the assignment statement **unique_ptr<int> up6 = up1;** OK? Why?

Is there any memory leak problem in the program?
Need we use the statement **delete[] p;** to free the memory we allocated before?

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

using namespace std;

class A
{
private:
    int x;

public:
    A(int a) : x(a)
    {
        cout << "Constructor with data:" << x << endl;
    }
    ~A()
    {
        cout << "Destructor with data:" << x << endl;
    }
    int getA() const
    {
        return x;
    }
};
```

```cpp
int main()
{
    unique_ptr<A> up1(new A(1));
    cout << "up1's data:" << up1->getA() << endl;

    A* aptr = new A(2);
    unique_ptr<A> up2(aptr);
    cout << "up2's data:" << up2->getA() << endl;

    unique_ptr<A> up3 = make_unique<A>(3);
    cout << "up3's data:" << up3->getA() << endl;

    return 0;
}
```

You can create unique_ptr by user-define type.

```
Constructor with data:1
up1's data:1
Constructor with data:2
up2's data:2
Constructor with data:3
up3's data:3
Destructor with data:3
Destructor with data:2
Destructor with data:1
```

Shared pointer

- More than one pointer can point to one object.
- After you initialize a shared_ptr<>, you can copy it, pass it by value in function arguments, and assign it to other shard_ptr<> instances.
- The shared pointer maintains a Ref_count that is a reference counter.
- If the last pointer is released, the dynamic memory is released.
- We can know the value of Ref_count by using the **use_count()** function.

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

using namespace std;

class A
{
private:
    int x;

public:
    A(int a) : x(a)
    {
        cout << "Constructor with data:" << x << endl;
    }
    ~A()
    {
        cout << "Destructor with data:" << x << endl;
    }
    int getA() const
    {
        return x;
    }
};
```

```cpp
int main()
{
    shared_ptr<A> up1(new A(1));
    cout << "up1's data:" << up1->getA() << endl;

    A* aptr = new A(2);
    shared_ptr<A> up2(aptr);
    cout << "up2's data:" << up2->getA() << endl;

    shared_ptr<A> up3 = make_shared<A>(3);
    cout << "up3's data:" << up3->getA() << endl;

    shared_ptr<A> up4 = up3;
    cout <<  "After initializing,up4's data:" << up4->getA() << endl;

    up4 = up2;
    cout << "After assignment, up4's data:" << up4->getA() << endl;

    return 0;
}
```

Use **new** operator or **make_shared()** function to create shared_ptr. make_shared() is recommended.

```
Constructor with data:1
up1's data:1
Constructor with data:2
up2's data:2
Constructor with data:3
up3's data:3
After initializing,up4's data:3
After assignment, up4's data:2
Destructor with data:3
Destructor with data:2
Destructor with data:1
```

Does shared pointer always releases memory? Can we do this?

```cpp
class B;

class A {
public:
    shared_ptr<B> pb;
    A() { cout<<"Constructer A" <<endl; }
    ~A() { cout<<"Destructor A" <<endl; }
};

class B {
public:
    shared_ptr<A> pa;
    B() { cout<<"Constructer B" <<endl; }
    ~B() { cout<<"Destructor B" <<endl; }
};

int main() {

    shared_ptr<A> spa = make_shared<A>();
    shared_ptr<B> spb = make_shared<B>();

    spa->pb = spb;
    spb->pa = spa;

return 0;
}
```

Exercise:

1. 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;
    pd = shared_ptr<double>(p_reg);
    cout << "*pd = " << *pd << endl;

    shared_ptr<double> pshared = p_reg;
    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;
}
```

Exercise:

2.Create a class Matrix to describe a matrix. The element type is float. One member of the class is shared_ptr<float> for the matrix data.

The two matrices can share the same data through a copy constructor or a copy assignment.

The following code can run smoothly without memory problems.

```
class Matrix{...};
Matrix a(3,4);
Matrix b(3,4);
Matrix c = a + b;
Matrix d = a;
d = b;
```