C/C++ Program Design

LAB 12

CONTENTS

- More on constructor and destructor
- Learn how to define and use copy constructor
- Learn how to define and use assignment operator
- Returning objects
- Using pointers to objects

2 Knowledge Points

2.1 Constructor and Destructor

2.2 Copy Constructor

2.3 Assignment operator

2.4 Returning object

2.5 Using pointers to objects

2.1 Constructor and Destructor

If a class holds a pointer, how are the constructor and destructor different?

```
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:

    String(const char* cstr = 0);
```

```
String::String(const char* cstr)
{
    if (cstr) {
        m_data = new char[strlen(cstr) + 1];
        strcpy(m_data, cstr);
    }
    else {
        m_data = new char[1];
        *m_data = '\0';
    }
}
```

There is a **pointer-to-char** member in the class declaration, which means that the class declaration does not allocate storage space of the string itself. Instead, it uses **new** in the constructor to allocate space for the string. The constructor must **allocate enough memory** to hold the string, and then it must **copy the string** to that location.

```
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:

    String(const char* cstr = 0);

    ~String();
```

```
String::~String()
{
    delete[] m_data;
}
```

The destructor must **delete** the member points to memory allocated with **new**. When the String object expires, the **m_data** pointer expires. But the memory **m_data** pointed to remains allocated unless you use **delete** to free it.

2.2 Copy Constructor

A copy constructor is used to copy an object to a newly created object. It is used during initialization. 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.

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

```cpp
//main.cpp --- using the copy constructor Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{

    Complex c1(1, 2);
    Complex c2(c1);  //initialize c2 with c1 by default copy constructor


    cout << c2 << endl;

    cout << "Done!" << endl;
    return 0;
}
```

Using c1 as an argument to create c2 invokes the default copy constructor to initialize c2. This means that each member value of c1 is copied to that of c2 (Both member values of c1 and c2 are equal).

```
1 + 2i

Done!
```

Note: Complex c2(c1); has the same function as Complex c2 = c1;

This statement is not an assignment statement.

If you define your own copy constructor, the default copy constructor does not exist.

```
//complex.h --- Complex class including copy constructor
#ifndef _COPY_CONSTRUCTOR_COMPLEX_H
#define _COPY_CONSTRUCTOR_COMPLEX_H

class Complex {
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double re, double im);
    Complex(const Complex&);

    friend std::ostream& operator << (std::ostream& os, const Complex& other);


};

#endif //_COPY_CONSTRUCTOR_COMPLEX_H
```

```
// complex.cpp --- implementing Complex class methods
#include <iostream>
#include "complex.h"

Complex::Complex() : real(0), imag(0)
{
}

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

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


std::ostream& operator<< (std::ostream& os, const Complex& other)
{
    os << other.real << " + " << other.imag << "i" << std::endl;
    return os;
}
```

```cpp
//main.cpp --- using the copy constructor Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{

    Complex c1(1, 2);
    Complex c2(c1);  //initialize c2 with c1 by copy constructor
    Complex c3 = c1;  //initialize c3 with c1 by copy constructor

    cout << c2 << c3 << endl;

    cout << "Done!" << endl;
    return 0;
}
```

```
Copy Constructor called
Copy Constructor called
1 + 2i
1 + 2i

Done!
```

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:

```
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.

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.

Suppose this situation, a class member holds a pointer, is the default copy constructor appropriate?

```
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;
```

```
String s1("hello");
String s2("world");

String s3(s2);
```

The default copy constructor performs a **member-by-member copy** and copies by value. This means it just copies pointer.

When you create s3 by s2, it invokes the default copy constructor if you don't provide one. What the default copy constructor do is to have the two pointers points to the same string.

You should provide an explicit copy constructor and copy the string to the member. This is called ***deep copy***.

```
String::String(const String& str)
{
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
}
```

What makes defining the copy constructor necessary is the fact that some class members are **new-initialized pointers to data** rather than the data themselves.

2.3 Assignment Operator

An implicit assignment operator performs a **member-to-member copy**.

```
#include "String.h"
#include <iostream>

using namespace std;

int main()
{
    String s1("hello");
    String s2("world");

    String s3(s2);
    cout << s3 << endl;

    s3 = s1;

    cout << s3 << endl;
    cout << s2 << endl;
    cout << s1 << endl;

    return 0;
}
```

This assignment statement make both s1.m_data and s3.m_data point to the same address

You should provide an explicit assignment operator definition to make a ***deep copy***. The implementations is similar to that of the copy constructor, but there are some differences:

- Because the target object may already refer to previously allocated data, the function should use delete [] to free former obligations.
- The function should protect against assigning an object to itself; otherwise, the freeing of memory described previously could erase the object's contents before they are reassigned.
- The function returns a reference to the invoking object.

```
String& String::operator=(const String& str)
{
    if (this == &str)
        return *this;

    delete[] m_data;
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
    return *this;
}
```

object assigned to itself

free old string

get space for new string

copy the string

return reference to invoking object

```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:
    String(const char* cstr = 0);
    String(const String& str);
    String& operator=(const String& str);
    ~String();

    char* get_c_str() const { return m_data; }

    friend std::ostream& operator<<(std::ostream& os, const String& str);

};

#endif
```

```cpp
#include <cstring>
#include "String.h"

String::String(const char* cstr)
{
    if (cstr) {
        m_data = new char[strlen(cstr) + 1];
        strcpy(m_data, cstr);
    }
    else {
        m_data = new char[1];
        *m_data = '\0';
    }
}

String::String(const String& str)
{
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
}

String& String::operator=(const String& str)
{
    if (this == &str)
        return *this;

    delete[] m_data;
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
    return *this;
}

String::~String()
{
    delete[] m_data;
}

#include <iostream>
using namespace std;

ostream& operator<<(ostream& os, const String& str)
{
    os << str.get_c_str();
    return os;
}
```

```cpp
#include "String.h"
#include <iostream>

using namespace std;

int main()
{
    String s1("hello");
    String s2("world");

    String s3(s2);
    cout << s3 << endl;

    s3 = s1;

    cout << s3 << endl;
    cout << s2 << endl;
    cout << s1 << endl;

    return 0;
}
```

2.4 Returning object

When a member function or standard function returns an object, you have choices. The function could return a reference to an object, a constant reference to an object, an object, or a constant object.

1. Returning a reference to a const object

For example, suppose you wanted to write a function Max() that returned the larger of two *Vector* object.

```
// version 1
Vector Max(const Vector & v1, const Vector & v2)
{
    if (v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

```
// version 2
const Vector & Max(const Vector & v1, const Vector & v2)
{
    if (v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

Returning an object invokes the copy constructor, whereas returning a reference doesn't.
The reference should be to an object that exists when the calling function is executing.

2. Returning a reference to non-const object

Two common examples of returning a non-const object are overloading the **assignment operator** and overloading the << operator for use with **cout**. The first is done for reasons of efficiency, and the second for reasons of necessity.

```
String & String::operator=(const String & st)
{
   if (this == &st)
      return *this;

   delete [] str;
   len = st.len;
   str = new char[len + 1];
   std::strcpy(str, st.str);

   return *this;
}
```

The return value of operator=() is used for chained assignment

```
String s1("Good stuff");
String s2, s3;
s3 = s2 = s1;
```

Returning a reference allows the function to avoid calling the String copy constructor to create a new String object. In this case, the return type is not const because the operator=() method return a reference to s2, which it does modify.

The return value of operator<<() is used for chained output

```
ostream & operator<<(ostream & os, const String & st)
{
   os << st.str;
   return os;
}
```

The return type has to be ostream & and not just ostream.The ostream class does not have a public copy constructor.

```
String s1("Good stuff");
cout << s1 << "is coming!";
```

3. Returning an object

If the object being returned is local to the called function, then it should not be returned by reference because the local object has its destructor called when the function terminates. Thus, when control return to the calling function, there is no object left to which the reference can refer.

```
Vector force1(50,60);
Vector force2(10,70);
Vector net;
net = force1 + force2;
```

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
```

The sum is a new, temporary object computed in Vector::operator+(), and the function shouldn't return a reference to a temporary object either. Instead, it should return an actual vector object, not a reference.

There is the added expense of calling the copy constructor to create the returned object, but that is unavoidable.

4. Returning an const object

The definition of Vector::operator+() allows these two usage as follows:

```
net = force1 + force2;

force1 + force2 = net;
```

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
```

The expression force1 + force2 stands for the temporary object which the copy constructor constructs. In the statement 1, the temporary object is assigned to net, but in statement 2, the sum of force1 and force2 is assigned to an temporary object. This causes misuse.

Declare the return type as a const object. Then statement 1 is still allowed but the statement 2 becomes invalid.

2.5 Using Pointers to Objects

C++ programs often use pointers to objects.

```
// saying2.cpp  --- using pointers to objects
//compile with string1.cpp

#include <iostream>
#include <cstdlib>      // (or stdlib.h) for rand() , srand()
#include <ctime>        // (or time.h) for time()
#include "string1.h"

const int ArSize = 10;
const int MaxLen = 81;

int main()
{
    using namespace std;
    String name;
    cout << "Hi, what's your name?\n>>";
    cin >> name;

    cout << name << ", please enter up to " << ArSize << " short sayings <empty line to quit>:\n";
    String sayings[ArSize];
    char temp[MaxLen];      //temporary string storage
    int i;

    for(i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get(temp, MaxLen);
        while(cin && cin.get() != '\n')
            continue;
        if(!cin || temp[0] == '\0')    //empty line?
            break;                  // i not increment
        else
            sayings[i] = temp;          //overloaded assignment
    }
```

Define two pointers to point the first object of the array. Note that these two pointers do not create new object, they merely point to the existing object.

```
int total = i;                  // total # of lines reads

if(total > 0)
{
    cout << "Here are your sayings:\n";
    for(i = 0; i < total; i++)
        cout << sayings[i] << '\n';


    //use pointers to keep track of shortest, first strings
    String * shortest = &sayings[0];     //initialize to the first object
    String * first = &sayings[0];
    for(i = 0; i < total; i++)
    {
        if(sayings[i].length() < shortest->length())
            shortest = &sayings[i];

        if(sayings[i] < *first)             //compare values
            first = &sayings[i];            //assign address
    }

    cout << "Shortest saying: \n" << *shortest << endl;
    cout << "First alphabetically: \n" << *first << endl;

    srand(time(0));
    int choice = rand() % total;      //pick index at random

    //use new to create, initialize new String object
    String *favorite = new String(sayings[choice]);
    cout << "My favorite saying:\n" << *favorite << endl;
    delete favorite;
}
else
    cout << "Not much to say, eh?\n";

cout << "Bye. \n";

return 0;
}
```

Use ***delete*** to delete the object

The pointer ***favorite*** provides the only access to the nameless object created by ***new*** and initializes the new ***String*** object by using the object saying[choice]. That invokes the copy constructor.

Several points about using pointers to objects:

1. Declare a pointer to an object by the usual notation:

String * glamour;

2. Initialize a pointer to point to an existing object:

String * first = &sayings[0];

3. Initialize a pointer by using new. Invoke the appropriate class constructor to initialize the newly created object:

```
// invokes default constructor
String * gleep = new String;

// invokes the String(const char *) constructor
String * glop = new String("my my my");

// invokes the String(const String &) constructor
String * favorite = new String(sayings[choice]);
```

4. Use the -> operator to access a class method via a pointer:

if (sayings[i].length() < shortest->length())

5. Apply the dereferencing operator(*) to a pointer to an object to obtain an object:

```
if (sayings[i] < *first)       // compare object values
    first = &sayings[i];
```