Advanced Programming

Lab 13

CONTENTS

- Learn how to define a class with a pointer member

- Returning objects

2 Knowledge Points

2.1 Class with a pointer as its member

2.2 Returning object

2.1 Class with a pointer as its member

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

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

class String
{
private:
    char *m_data;


public:
    String(const char *cstr = 0);


};
#endif
```

```cpp
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 definition. It should <mark>use **new operator** in the constructor to allocate space for the string</mark>. The constructor must **allocate enough memory** to hold the string, and then it must **copy the string** to that location.
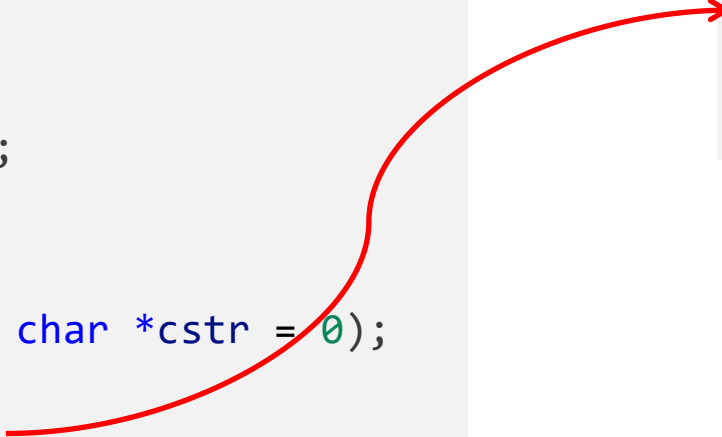
```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char *m_data;


public:
    String(const char *cstr = 0);

    ~String();


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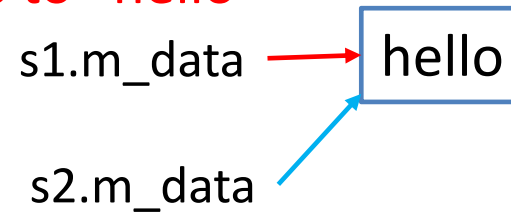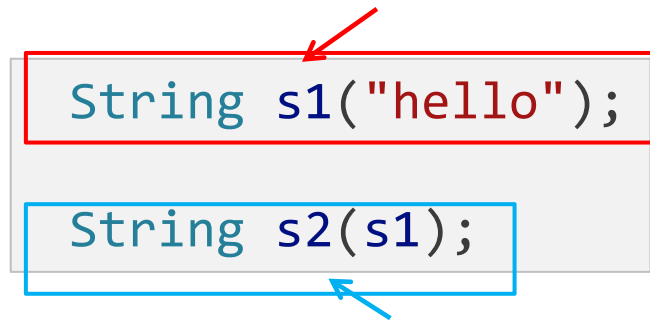
```cpp
String::~String()
{
    delete[] m_data;

}
```

The destructor must **delete** the member which points to the 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.

If a class member holds a pointer, is the default **copy constructor** appropriate?

invoke constructor, s1.m_data points to "hello"

s1.m_data ⟶ hello

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

s2.m_data

```
String s2(s1);
```

invoke copy constructor, s2.m_data points to "hello"

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

When you create s2 by s1, it invokes the default copy constructor because you don't provide one. What the default copy constructor do is to have the two pointers points to the same string.

When object s1 is out of its scope, its destructor will be invoked, the memory where s1.m_data is pointed is free. However, when object s2 is disappear, its destructor will also be called, the memory where s2.m_data is pointed is free. These two pointers are pointed to the same memory, the same memory will be deleted **twice**, that can cause error.

```
free(): double free detected in tcache 2
Aborted
```

You should provide an explicit copy constructor rather than default copy constructor and copy the string to the member. This is called *deep copy*.
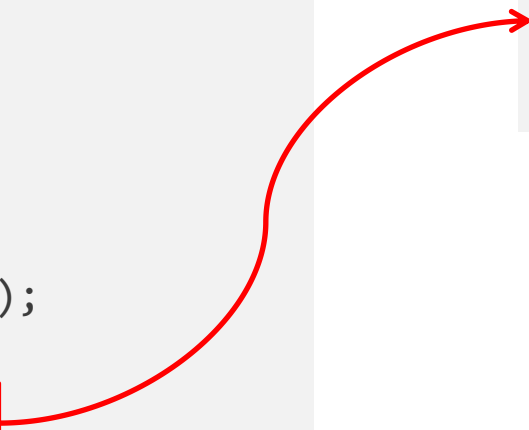
```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char *m_data;


public:
    String(const char *cstr = 0);

    String(const String &other);

    ~String();
};
#endif
```

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

new-initialized pointers 是指通过 new 关键字动态分配内存的指针。这些指针并不直接存储数据，而是存储动态分配内存块的地址。

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.

How about the default **copy assignment operator**? Is it appropriate?

The default assignment operator performs a **member-to-member copy**.

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

s2 = s1;
```

s1.m_data ──→ hello

s2.m_data ──→ world

s2 = s1

s1.m_data ──→ hello

s2.m_data ──→ world

memory leak

This assignment statement makes both **s1.m_data** and **s2.m_data** point to the same memory.

When release s1 and s2, the same memory will be deleted twice. Besides that, the invoking default assignment operator can cause memory leak.

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:

- Check for self-assignment
- Because the target object may already refer to previously allocated data, the function should use **delete []** to free former obligations.
- **Allocate enough memory** to hold the new string, and then it must **copy the string** to that location.
- The function returns a reference to the invoking object.

```cpp
String &String::operator=(const String &str)
{
    delete[] m_data;                          // free old string

    m_data = new char[strlen(str.m_data) + 1];  // get space for new string
    strcpy(m_data, str.m_data);                 // copy the string

    return *this;                               // return reference to invoking object
}
```
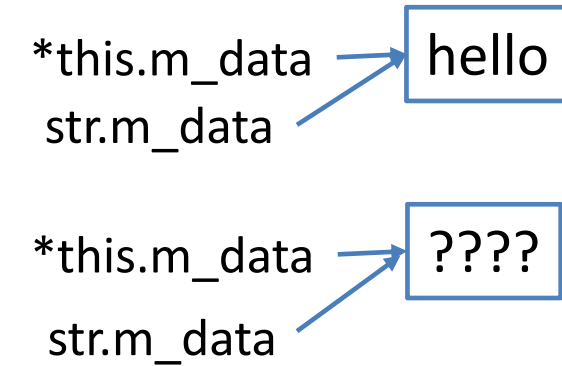
Note: we do not check self-assignment in this version.

If we do not check self-assignment, what will happen?

Suppose: we have *s1 = s1;* it will invoke *operator=(const String& str)*function, *this* and *str* are the same objects.

Without checking self-assignment, the statement *delete[] m_data;* in the copy assignment operator function will be executed, both ***this.m_data** and **str.m_data** point to the uncertain memory.

*this.m_data ⟶ hello
str.m_data

*this.m_data ⟶ ????
str.m_data

```
String &String::operator=(const String &str)
{
    if (this == &str)        ← check if an object
        return *this;          is assigned to itself

    delete[] m_data;


    m_data = new char[strlen(str.m_data)
+ 1];
    strcpy(m_data, str.m_data);


    return *this;
}
```

Note: we do check self-assignment in this version.

```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';
    }
}
```

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

```cpp
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;
}
```

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

```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;
}
```

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

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

Hard copy(deep copy)

```cpp
#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; }
};
```

```cpp
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(shallow 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.

```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.

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.

A smart pointer as a data member

```
#pragma once
#include <iostream>
#include <memory>

class Stringptr
{
private:
    std::shared_ptr<std::string> dataptr;
    int i;

public:

    Stringptr(const std::string& s = std::string(), int m = 0) : dataptr(std::make_shared<std::string>(s)), i(m) { }

    friend std::ostream& operator<<(std::ostream& os, const Stringptr& str)
    {
        os << *str.dataptr << "," << str.i;
        return os;
    }

};
```

Define a smart pointer as a data member

Initialization list is used. Do not use assignment statement for smart pointer in constructor.

2.2 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**.

2.2.1. 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. Thus version 2 does less work and is more efficient.
- The reference should be to an object that exists when the calling function is executing.
- Both v1 and v2 are declared as being const references, so the return type has to be const to match.

2.2.2. 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.

```cpp
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.

```cpp
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

```cpp
ostream& operator<<(ostream& os, const String& st)
{
    os << st.str;
    return os;
}
```
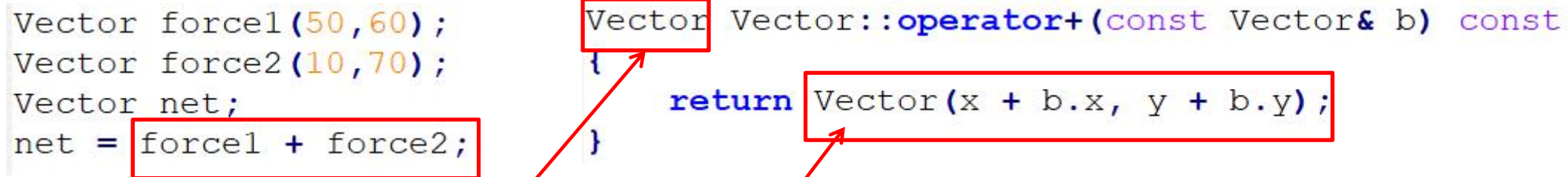
The return type has to be ostream & and not just ostream.The ostream class does not have a public copy constructor.

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

2.2.3. 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 Vector::operator+(const Vector& b) const
Vector force2(10,70);          {
Vector net;                        return Vector(x + b.x, y + b.y);
net = force1 + force2;         }
```

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.

2.2.4. Returning an const object

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

```
net = force1 + force2;

force1 + force2 = net;
```

在重载 operator+ 时，操作会创建并返回一个临时对象（temporary object）。这个临时对象是运算结果的副本，其生命周期非常短，一般只在当前表达式中有效。

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.

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

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

[ ] operator (array subscript operator)

User-defined classes that provide array-like access that allows both reading and writing(modifying) typically define two overloads for operator**[]**: const and non-const variants.

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

class String
{
private:
    char *m_data;


public:
    String(const char *cstr = 0);
    String(const String &other);

    ~String();


    String &operator=(const String &str);


    char& operator[](std::size_t position) { return m_data[position]; }
    const char& operator[](std::size_t position) const { return m_data[position]; }


    char* get_c_str() const { return m_data; }


    friend std::ostream &operator<<(std::ostream &os, const String &str);
};
#endif
```
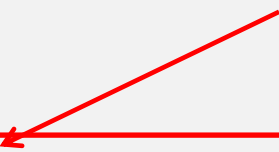
Usually, we overload [] operator with two versions, const version and non-const version for reading(rvalue) and writing(lvalue).

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

using namespace std;


int main()
{
    String s1("hello");
    const String s2("world");
    cout << "s1[0]:" << s1[0] << ",s2[0]:" << s2[0] << endl;


    char a = s1[1];
    char b = s2[2];
    cout << "a:" << a << ",b:" << b << endl;


    s1[0] = 'X';


    // s2[0] = 'Z';


    cout << "s2:" << s2 << endl;
    cout << "s1:" << s1 << endl;


    cout << "Done." << endl;


    return 0;
}
```

For non-const or const string, reading its value is allowed by its corresponding [] operator function respectively.

For non-const string, you can modify its value by non-const [] operator function

For const string, you can not modify its value by const [] operator function.

Note: Neither version of the [] operator function can match both non-const string and const string.

Exercise

1. The declaration of Stack as follows:

```cpp
// stack.h -- class declaration for the stack ADT
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};        // constant specific to class
    Item  * pitems;         // holds stack items
    int size;               // number of elements in stack
    int top;                // index for top stack item
public:
    Stack(int n = MAX);     // creates stack with n elements
    Stack(const Stack & st);
    ~Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item); // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);  // pop top into item
    Stack & operator=(const Stack & st);
};
```

Implement all the methods and write a program to demonstrates all the methods, including copy constructor and assignment operator.

2.Create a class Matrix to describe a matrix. The element type is float. One member of the class is a pointer(or a smart pointer) who points to 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);
a(1,2) = 3;
b(2,3) = 4;
Matrix c = a + b;
Matrix d = a;
d = b;
```

```
a is:
0 0 0 0
0 0 3 0
0 0 0 0
b is:
0 0 0 0
0 0 0 0
0 0 0 4
c is:
0 0 0 0
0 0 3 0
0 0 0 4
Before assignment,d is:
0 0 0 0
0 0 3 0
0 0 0 0
After assignment,d is:
0 0 0 0
0 0 0 0
0 0 0 4
```