第五天课堂笔记

一、回顾知识点

1.1 静态成员

```
static修饰成员变量或成员函数

static的成员变量
    不占对象空间， 所属类的， 与类存储在一起
    访问的方式：
        通过类名::静态成员变量
        类的对象也可以访问静态成员变量， 所有类的对象共有的


static 的成员函数：
     函数内不能访问非静态成员变量， 只能访问静态成员（变量和函数）。
```

1.2 const成员函数和修饰类对象

```
const修饰的函数内部不能修改成员变量。
    void xxx() const {}
const修饰的类对象（常对象），不能通过对象修改它的属性（成员变量），只能调用const修饰的成员函数。
    const X x;
```

1.3 友元

> 关键字 friend

```
友元的全局函数：
     在全局函数内，可以访问类对象的所有成员。
     friend 返回值类型 全局函数名(类名 &对象名);

友元的成员函数
    将A个类的成员函数设置为B类的友元函数， 在A类声明（不实现），在B类定义时，声明A类的某个成员函数是B类的友元。则在A的成员函数内，可以通过作为参数B类对象，来访问B类的一切。

    friend 返回值类型 A::成员函数(B &b);

友元的类:
   friend 类名;
```

1.4 运算符重载

> 运算符重载是一个函数（可以全局函数，成员函数）。
>
> 格式： 返回值类型 operator运算符 (参数列表);
>
> 参数的个数，依据运算符的操作数及功能。

可重载的运算符：

```
+  可以实现两个类的对象相加
++/-- 可以实现一个类对象的自加或自减。
      ++/-- 在前，  重载的函数参数个数为0（不需要）
      ++/-- 在后，  重载的函数的参个数为1（int类型占位参数）
<, >, >=, <=, ==  关系运算符的重载 （成员函数的形式，需要一个在参数）
*,  ->    可以实现代理设计模式，  在一个类中代理另一个类的操作。
          运算符重载的函数是不需要参数的，
          -> 重载返回指针变量
          * 重载返回指针变量值（类的对象）
```

二、运算符重载II

2.1 =赋值重载

> 【注意】=重载时，可能会调用类本身的拷贝构造函数。如果左值是没有创建的对象时，会调用`拷贝构造函数`，如果左值是已创建的类对象，会执行`=重载函数`，实现数据的拷贝。

如：

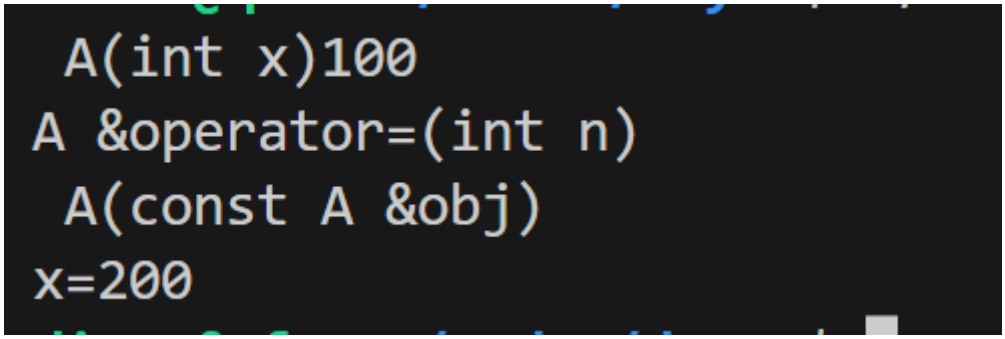
```
#include <iostream>

using namespace std;

class A
{
private:
    int x;

public:
    A(int x)
    {
        cout << " A(int x)" << x << endl;
        this->x = x;
    }
    A(const A &obj)
    {
        cout << " A(const A &obj)" << endl;
        this->x = obj.x;
    }
    A &operator=(A &other)
    {
        cout << "A &operator=(A &other)" << endl;
        this->x = other.x;
        return *this;
    }
    A &operator=(int n)
    {
        cout << "A &operator=(int n)" << endl;
        this->x = n;
        return *this;
    }
    void show()
    {
        cout << "x=" << x << endl;
    }
};
```

```
int main(int argc, char const *argv[])
{
    A a1(100);
    a1 = 200;  // operator=(200)
    A a3 = a1; // A(a1);
    a3.show();
    // a3 = a2; // operator=() 调用赋值重载函数
    // a3.show();
    return 0;
}
```



2.2 () 函数调用重载

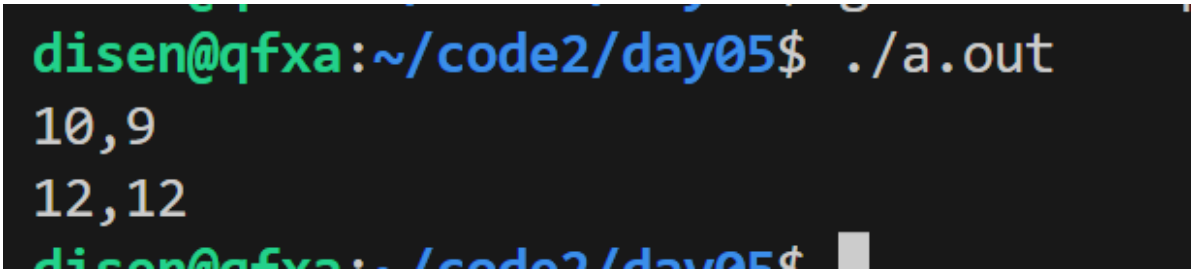
> 当类对象作为函数调用时，会执行 operator()(参数列表)函数。

如：

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

class A{
private:
   int x,y;
public:
   A(int x, int y){
      this->x = x;
      this->y = y;
   }
   void show(){
      cout << x << "," << y << endl;
   }
   void operator()(int a, int b){
       this->x += a;
       this->y += b;
   }
};

int main(){
    A a1(10, 9);
    a1.show();
    a1(2, 3);  // 类对象作为函数使用，实现 2+x, 3+y
    a1.show();
    return 0;
}
```



2.3 不要重载&&和||

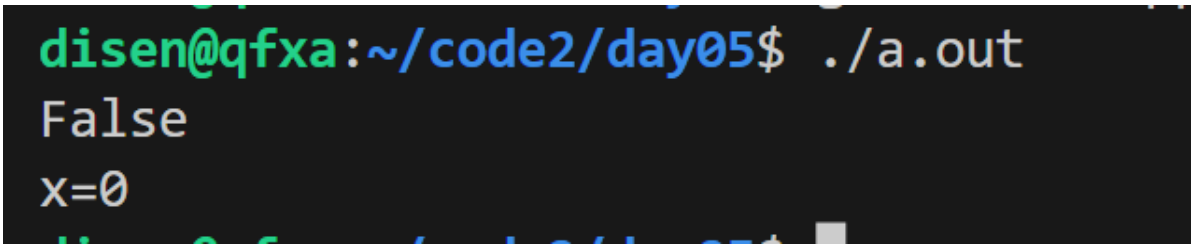
> 因为逻辑与或逻辑或存在逻辑短路的问题，但是在重载时，有可能达不到想要的效果。

如： 普通的&&或||的表达式应用

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    int x = 0, y = 1;
    if (x && (x += y))
    {
        cout << "True" << endl;
    }
    else
    {
        cout << "False" << endl;
    }
    cout << "x=" << x << endl;
}
```



如： 逻辑运算符重载的情况

```
#include <iostream>

using namespace std;
class A
{
private:
    int x;

public:
    A(int x) { this->x = x; }
    A &operator+=(A &other)
    {
```

```
        this->x += other.x;
        return *this;
    }
    bool operator&&(A &other)
    {
        return this->x && other.x;
    }
};

int main(int argc, char const *argv[])
{
    A a1(0), a2(1);
    // 先调用 (a1+= a2) 的 +=运算符重载函数，导到a1的x值从0变成1
    // 再调用 && 重载函数，得出 1 && 1 表达式结果
    // 总结： 实际上是调用&& 重载函数时触发了+=重载，+=重载先执行，再执行&&重载内部的逻辑
    if (a1 && (a1 += a2))
    {
        cout << "True" << endl;
    }
    else
    {
        cout << "False" << endl;
    }
}
```

```
disen@qfxa:~/code2/day05$ ./a.out
True
```

2.4 符号重载总结

```
=, [], () 和 -> 操作符只能通过成员函数进行重载
<< 和 >>只能通过全局函数配合友元函数进行重载
不要重载 && 和 || 操作符，因为无法实现短路规则
```

常规建议

运算符	建议使用
所有的一元运算符	成员
= () [] -> ->*	必须是成员
+= -= /= *= ^= &= != %= >>= <<=	成员
其它二员运算符	非成员

2.5 自定义String字符串类

> 模拟c++的string类的， 实现字符串创建、拼接、赋值、取字符串中第index位置的字符

mystring.h头文件

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

```
#include <cstring>
#include <cstdlib>
using namespace std;

class MyString
{
    friend ostream &operator<<(ostream &cout, const MyString &str);
    friend istream &operator>>(istream &cin, MyString &str);

public:
    MyString(const char *str);
    MyString(const MyString &str);
    ~MyString();

public:
    // 拼接字符串
    MyString &operator+(MyString &other);
    MyString &operator+(const char *other);

    // 字符串赋值
    MyString &operator=(MyString &other);
    MyString &operator=(const char *other);

    // 读取字符串第index位置的字符
    char operator[](int index);

private:
    char *mStr;
    int mSize;
};
#endif
```

mystring.cpp源文件

```
#include "mystring.h"

ostream &operator<<(ostream &cout, const MyString &str)
{
    cout << str.mStr;
    return cout;
}
istream &operator>>(istream &cin, MyString &str)
{
    cin >> str.mStr;
    return cin;
}

MyString::MyString(const char *str)
{
    mSize = strlen(str);
    this->mStr = new char[mSize + 1];
    strcpy(this->mStr, str);
}

MyString::MyString(const MyString &str)
{
    mSize = str.mSize;
```

```
    char *p = new char[mSize + 1];
    strcpy(p, str.mStr);
    delete[] this->mStr;
    this->mStr = p;
}
MyString::~MyString()
{
    delete[] mStr;
}

MyString &MyString::operator+(MyString &other)
{
    mSize += other.mSize;
    char *p = new char[mSize + 1];
    strcpy(p, this->mStr);
    strcat(p, other.mStr); // 字符串连接
    delete[] mStr;         // 删除之前的空间
    mStr = p;
}
MyString &MyString::operator+(const char *other)
{
    mSize += strlen(other);
    char *p = new char[mSize + 1];
    strcpy(p, this->mStr);
    strcat(p, other); // 字符串连接
    delete[] mStr;    // 删除之前的空间
    mStr = p;
}

MyString &MyString::operator=(MyString &other)
{
    mSize = other.mSize;
    char *p = new char[mSize + 1];
    strcpy(p, other.mStr);
    delete[] mStr; // 删除之前的空间
    mStr = p;
}
MyString &MyString::operator=(const char *other)
{
    mSize = strlen(other);
    char *p = new char[mSize + 1];
    strcpy(p, other);
    delete[] mStr; // 删除之前的空间
    mStr = p;
}

bool MyString::operator==(MyString &other)
{
    return strcmp(this->mStr, other.mStr) == 0;
}
bool MyString::operator==(const char *other)
{
    return strcmp(this->mStr, other) == 0;
}
// 读取字符串第index位置的字符
char MyString::operator[](int index)
{
    cout << "长度: " << mSize << endl;
```

```
    // index支持负数：  表示从右边开始取字符
    // index是负数，  计算机存储是补码，如果直接位运算，以补码的方式 & 位运算的
    // 需要手动取反补码（反码+1）
    int absIndex = (~index + 1) & 0x7fffffff; // abs(index);
    cout << "absIndex = " << absIndex;
    if (index >= 0 && index < mSize) // 从左向右取
    {
        return this->mStr[index];
    }
    else if (absIndex <= mSize)
    {
        // 从右向左取  mSize=7
        // index = -1    mSize-1 =  6
        // index = -2    mSize-2 = 5
        // index = -7    mSize-7 = 0
        return this->mStr[mSize + index];
    }
    else
    {
        return '\0';
    }
}
```

main.cpp :

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

using namespace std;

int main(int argc, char const *argv[])
{
    MyString s1("disen");
    MyString s2(", 666!");
    cout << s1 + s2 << endl;
    // MyString s3 = s1; // 拷贝构造函数
    MyString s3 = MyString("jack, 999!");
    s1 = s3;
    cout << s1 << endl;
    // s1 = MyString("lucy,555!");  初始化类对象使用，在此不能使用
    cout << "第一个字符" << s1[0] << "最后一个字符：" << s1[-1] << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day05/mystring$ ./a.out
disen, 666!
jack, 999!
长度: 10
absIndex = 1长度: 10
absIndex = 0第一个字符j最后一个字符: !
```

三、继承和派生

3.1 继承的概述

为什么存在继承：

```
1）减少重复的代码， 减轻程序整体的体量。
2）继承的好处，可以将共性的内容封装成一个基类（父类）， 遇到专项业务时，可以扩展基类变为一个新类，在新类中重点扩展功能。
```

c++中继承的最重要的特征是代码重用。

子类与父类的关系：

```
类是由具有相同特征（属性）和行为（方法）的多个客观事物（对象）抽象出来的。
子类是从父类中派生出来的类， 子类继承了父类， 父类由多个类的共同的部分抽象出来的。
父类： 基类
子类： 派生类， 可以扩展父类的功能（体量比父类大，功能比父类多）
       子类的组成部分： 1） 父类的， 2） 自定义（新扩展）的
```

子类继承父类的语法：

```
class 子类名: 继承方式  父类名 {

};
```

继承方式：

```
public（公有继承）： 继承父类的成员访问权限不变。 【建议】
protected(保护继承)： 从父类中继承的成员访问权限 变为 受保护的(protected)
private(私有 继承)： 从父类中继承的成员访问权限变为私有的。
```

继承源（单父类， 多父类）：

```
单父类： 单继承
多父类： 多继承， :继承方式 父类名, 继承方式 父类名2, ...
```

如：

```
#include <iostream>

using namespace std;

class Person
{
private:
    string name;
    int age;

public:
    Person(const string &name, int age)
    {
        this->name = name;
        this->age = age;
```

```
    }
    void hi()
    {
        cout << this->name << ", " << this->age << endl;
    }
};

class Worker : public Person
{
private:
    int salary; // 月薪
    int year;   // 年限
public:
    Worker(const string &name, int age, int salary, int year) : Person(name,
age)
    {
        this->salary = salary;
        this->year = year;
    }

    void hi() // 重写的父类的函数，覆盖了父类的hi()成员函数了
    {
        // 先调用父类的hi
        Person::hi();
        cout << salary << "," << year << endl;
    }
};

int main(int argc, char const *argv[])
{
    Worker w1 = Worker("disen", 18, 2000, 2);
    w1.hi(); // 从父类继承过来的
    return 0;
}
```

【小结】

```
1） 子类 继承父成的成员函数、成员变量， 不能继承父类的构造与析构函数。
    每一个类的构造函数和析构函数，只负责当前类对象的创建与回收的。

2)  子类的构造函数定义时，可以调用父类的构造函数进行父类成员变量的初始化
    子类名(参数列表):父类名(参数列表),... {

    }

3) 子类定义成员函数名同父类的成员函数名，则表示子类重写父类的成员函数 （功能重定义）
   在重写的函数内， 可以通过 父类名::成员函数名(参数列表) 调用父类的成员函数（原功能）。
```

3.2 派生类访问控制

派生类的访问权限规则如下：

公有派生		私有派生		保护派生	
基类属性	派生类权限	基类属性	派生类权限	基类属性	派生类权限
私有	不能访问	私有	不能访问	私有	不能访问
保护	保护	保护	私有	保护	保护
公有	公有	公有	私有	公有	保护