1. 友元

1.1 简述

我们已知道类具有封装和信息隐藏的特性。只有类的成员函数才能访问类的私有成员，程序中的其他函数是无法访问私有成员的。非成员函数可以访问类中的公有成员，但是如果将数据成员都定义为公有的，这又破坏了隐藏的特性。另外,应该看到在某些情况下，特别是在对某些成员函数多次调用时， 由于参数传递，类型检查和安全性检查等都需要时间开销，而影响程序的运行效率。

为了解决上述问题，提出一种使用友元的方案。友元是一种定义在类外部的普通函数或类，但它需要在类体内进行说明。为了与该类的成员函数加以区别，在说明时前面加以关键字**friend**。友元不是成员函数，但是它可以访问类中的私有成员。友元的作用在于提高程序的运行效率， 但是，它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。不过，类的访问权限确实在某些应用场合显得有些呆板，从而容忍了友元这一特别语法现象。

1.2 友元函数

如果把一个正常函数的声明复制到类中,且在前面加上关键字**friend**,那么这个函数是该类的友元函数;函数可以访问该类型对象的私有成员.

1.2.1 声明

```
1 把一个正常函数的声明复制一份到类中,且在前面加上关键字friend;                    Eg: 5-1 :: House.h :: 9行 10行;
2 把一个正常类函数的声明复制一份到类中(要有类名和作用域符),且在前面加上关键字friend;  Eg: 5-1 :: House.h :: 14行 15行;
```

1.2.2 使用

在函数体内可以访问类对象的私有成员: 成员变量,成员函数;

Eg: 5-1 :: Main.cpp :: 31行; 5-1 :: Main.cpp :: 36行; 5-1 :: Design.cpp :: 14行; 5-1 :: Design.cpp :: 19行;

1.3 友元类

如果把一个哦对的声明复制到另一类中,且在前面加上关键字**friend**,那么这个类是另一类的友元类;类成员函数可以访问另一类对象的私有成员.

1.3.1 声明

```
1 把一个类的声明复制一份到另一类中,且在前面加上关键字friend;
```

1.3.2 使用

在类函数体内可以访问类对象的私有成员: 成员变量,成员函数;

Eg: 5-1 :: Master.cpp :: 14行; 5-1 :: Master.cpp :: 19行;

2. 运算符重载

2.1 简述

运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。

其实运算符的调用,可以看作是函数的调用,只是使用方式有些许区别;

2.2 运算符重载分类

1. 重载为类的成员函数;
2. 重载为类的非成员函数(非成员函数通常是友元);

2.3 声明

```
1. 重载为类的成员函数:
类型 operator关键字 运算符 ( 参数表 );

2. 重载为类的非成员函数:
friend关键字 类型 operator关键字 运算符 ( 参数表 );
```

2.4 定义

定义与正常的成员函数&非成员函数相同;

2.5 使用

1. 重载为类的成员函数: 把`operator关键字 运算符`当作成员函数名,使用方法与其相同了;
   还有一种简化的调用方法, `对象 运算符 参数表对象`;
2. 重载为类的非成员函数: 把`operator关键字 运算符`当作成非员函数名,使用方法与其相同了;

2.6 注意

```
(1) 除了
    1. .        类属关系运算符
    2. →        成员指针运算符
    3. ::       作用域运算符
    4. sizeof   运算符
    5. ?:       三目运算符
    6. #        预处理符号
    C++中的所有运算符都可以重载。
(2) 重载运算符限制在C++语言中已有的运算符范围内的允许重载的运算符之中，不能创建新的运算符。
(3) 运算符重载实质上是函数重载，因此编译程序对运算符重载的选择，遵循函数重载的选择原则。
(4) 重载之后的运算符不能改变运算符的优先级和结合性，也不能改变运算符操作数的个数及语法结构。
(5) 运算符重载不能改变该运算符用于内部类型对象的含义。它只能和用户自定义类型的对象一起使用，或者用于用户自定义类型的对象和内部类型的对象混合使用时。
(6) 运算符重载是针对新类型数据的实际需要对原有运算符进行的适当的改造，重载的功能应当与原有功能相类似，避免没有目的地使用重载运算符。
```

3. new和delete

基本不会去重载的,可以不了解;

4. 赋值运算

```
class CSum
{
private:
    int m_iSum0{};
    int m_iSum1{};

public:
    CSum(int iSum0, int iSum1);
    ~CSum();

    CSum& operator= (const CSum& r);
};


int main()

{
```

```
    CSum sum0(1, 1);
    CSum sum1(2, 2);

    sum0 = sum1;    //调用=

    return 0;
}


CSum::CSum(int iSum0, int iSum1)
{
    m_iSum0 = iSum0;
    m_iSum1 = iSum1;
}

CSum::~CSum()
{
}

CSum & CSum::operator=(const CSum & r)
{
    m_iSum0 = r.m_iSum0;
    m_iSum1 = r.m_iSum1;
    return *this;
}

```

5. 自增自减运算符

```
#include <stdio.h>

class CSum
{
public:
    int m_iSum0{};
    int m_iSum1{};

public:
    CSum(int iSum0, int iSum1);
    ~CSum();

    CSum& operator++();     //前置++
    CSum& operator++(int);  //后置++;为了与前置++作区别,加一个参数,只是让编译识别而已;
};


int main()
{
    CSum sum0(1, 1);
    CSum sum1(2, 2);

    sum1 = sum0++;                  //这里到底在做什么???
    //printf("%d", sum0.m_iSum0);
    sum1 = ++sum0;                  //这里到底在做什么???
    //printf("%d", sum0.m_iSum0);

    return 0;
}


CSum::CSum(int iSum0, int iSum1)
{
    m_iSum0 = iSum0;
    m_iSum1 = iSum1;
}

```

```
CSum::~CSum()
{
}

CSum & CSum::operator++()
{
    ++m_iSum0;
    ++m_iSum1;
    return *this;
}

CSum & CSum::operator++(int)
{
    m_iSum0++;
    m_iSum1++;
    return *this;
}
```

6. 作业

写一个字符串类CString.

1. 重载+,实现 "123" + "456" (= "123456");
2. 重载+=,实现 "123 += "456" (= "123456");
3. 重载=;
4. 重写拷贝构造函数;