C++面向对象程序设计

目标

- 培养正规的、大气的编程习惯
- 以良好的方式编写C++ class
- 学习Classes之间的关系
    - 继承
    - 符合
    - 委托

C++历史

- B语言（1970年）
- C语言（1972年）
- 1983年，Bjarne Stroustrup在AT&T贝尔实验室开始设计C++，并于1985年完成第一个版本。
- Java语言（1995年）是由Sun公司设计的一种面向对象的编程语言，它的设计目标是取代C++语言。
- C#语言（2000年）是由微软公司设计的一种面向对象的编程语言，它的设计目标是取代Java语言。

C++ 演化

1. C++ 98
2. C++ 03
3. C++ 11
4. C++ 14 C++ = C++语言 + C++标准库

C++ 与 C语言的区别

- C++是C语言的超集
- C++是面向对象的
- 将C语言的函数和数据封装成类
- class是C++的核心

classes的两种经典分类

- 带指针的类 complex
- 不带指针的类 string

C++程序的代码基本形式

- 头文件 .h
    - 自己的头文件

```
#include "complex.h"
```

    - 标准库头文件

```
#include <iostream>
```

- 源文件 .cpp
- 标准库 .h

C++头文件中的防御式声明

```
#ifndef COMPLEX_H
#define COMPLEX_H
// 前置声明
//
#endif
```

C++模板简介

- 模板是泛型编程的基础
- 模板通过 template 关键字来定义
- 在类定义前加 `template <typename T>`

```
template <typename T>
class complex
{
    T re, im;
public:
    complex(T r, T i) : re(r), im(i) {}
    complex(T r) : re(r), im(0) {}
    complex() : re(0), im(0) {}
    T real() const { return re; }
    void real(T r) { re = r; }
    T imag() const { return im; }
    void imag(T i) { im = i; }
};
```

inline(内联) 函数

- 函数在类的内部定义，自动成为内联函数
- 函数在类的外部定义，需要在函数前加 inline 关键字
- inline关键字只是"建议"编译器将函数内联，编译器可以忽略这个建议

C++中的访问级别（access level）

- public

    - 类的外部可以访问
    - 将函数声明为public，可以让用户使用这个函数
- private
    - 类的外部不可以访问
    - 将函数声明为private，可以隐藏实现细节
    - 将数据声明为private，可以保护数据
    - 将所有数据都声明为private。这样，用户只能使用类提供的接口，而不能直接访问数据
- protected
    - 子类可以访问，类的外部不可以访问

构造函数（constructor）

- 构造函数是一种特殊的成员函数，它的名字与类名相同
- 构造函数没有返回值，也不能声明为void

```
complex(double r, double i) : re(r), im(i) {}
```

等价于（不完全相同）

```
complex(double r, double i)
{
    re = r;
    im = i;
}
```

上述两种写法的区别在于，前者的初始化列表（initializer list）可以避免一些不必要的拷贝操作。后者的写法，会先创建一个临时对象，然后再将临时对象赋值给类的成员变量（效率上会差一些）。

构造函数可以重载

- 重载的构造函数可以有不同的**参数列表**
- 构造函数有默认值时，不能够再重载一个无参构造函数

```
complex(double r = 0, double i = 0) : re(r), im(i) {}
complex() : re(0), im(0) {}
```

上述两个构造函数不能同时存在，否则会报错。原因是，编译器无法区分 complex()该调用哪个构造函数。

```
complex(double r) : re(r), im(0) {}
complex() : re(0), im(0) {}
```

上述两个构造函数可以同时存在。

ctors放在private中

单例模式中，将构造函数放在private中，可以防止用户创建对象。

```
class Singleton
{
private:
    Singleton() {}
    static Singleton* instance;
public:
    static Singleton* getInstance()
    {
        if (instance == nullptr)
            instance = new Singleton();
        return instance;
    }
};
```

const member function(常量成员函数)

```
class complex
{
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {}
    double real() const { return re; }
    double imag() const { return im; }
private:
    double re, im;
};
```

- **常量成员函数不能修改类的成员变量**
- 在常量成员函数中，可以调用其他的常量成员函数
- 应该将所有不改变对象状态的函数都声明为常量成员函数

举例

```
complex c1(2, 1);
cout << c1.real() << endl;
cout << c1.imag() << endl;
```

上述代码中，**c1不是一个常量对象，因此可以调用非常量成员函数，也可以调用常量成员函数。**

```
```c++
const complex c1(2, 1);
```
```

```
cout << c1.real() << endl;
cout << c1.imag() << endl;
```

上述代码中，**c1是一个常量对象，因此只能调用常量成员函数**。因为编译器认为调用非常量成员函数可能会修改对象的状态，这是不允许的。

参数传递：pass by value vs. pass by reference(to const)

```
void f(complex z)
{
    cout << z.real() << endl;
    cout << z.imag() << endl;
}
```

```
void f(complex& z)
{
    cout << z.real() << endl;
    cout << z.imag() << endl;
}
```

```
void f(const complex& z)
{
    cout << z.real() << endl;
    cout << z.imag() << endl;
}
```

上述代码中，**z是一个复数对象的引用**。在函数f中，**z是一个常量引用**。因此，**z不能被修改。**

引用（reference）

- 引用在底层就是指针
- 传引用的效率比传值要高
- **所有的参数传递尽量传引用**

返回值传递 return by value vs. return by reference(to const)

```
class complex
{
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {}
    // return by reference
    complex& operator+=(const complex&);
    double real() const { return re; }
    double imag() const { return im; }
```

```
private:
    double re, im;
    friend complex& __doapl(complex*, const complex&);
};
```

友元(friend)

```
class complex
{
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {}
    // return by reference
    complex& operator+=(const complex&);
    double real() const { return re; }
    double imag() const { return im; }
private:
    double re, im;
    // 友元函数
    friend complex& __doapl(complex*, const complex&);
};
inline complex&
__doapl(complex* ths, const complex& r)
{
    ths->re += r.re;
    ths->im += r.im;
    return *ths;
}
```

- **友元函数可以访问类的私有成员变量**
- 相同class的各个对象互为友元

```
class complex
{
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {}
    int func(const complex& r)
    {
        return r.re + r.im;
    }
private:
    double re, im;
};
```

上述代码中，**func函数可以访问complex类的私有成员变量。**

class body外的各种定义(definition)

什么情况下可以 pass by reference(to const)？ 什么情况下可以 return by reference(to const)？

- 如果函数不是需要返回内部定义的局部变量（一出函数就消亡），而是返回函数外界定义的对象（如该对象通过参数传递指针的方式进入函数），那么就可以返回对该对象的引用。

操作符重载（1），成员函数 this指针

```
class complex
{
public:
   complex(double r = 0, double i = 0) : re(r), im(i) {}
   complex& operator+=(const complex&);
   double real() const { return re; }
   double imag() const { return im; }
private:
   double re, im;
   friend complex& __doapl(complex*, const complex&);
};
inline complex&
complex::operator+=(const complex& r)
{
   re += r.re;
   im += r.im;
   return *this;
}
```

上述代码中，**operator+=是一个成员函数**。因此，**this指针指向调用该成员函数的对象**。

```
complex c1(2, 1);
complex c2(4, 2);
c1 += c2;
```

上述代码中，**c1是调用operator+=的对象**。因此，**this指针指向c1**。

任何一个成员函数都有一个this指针，指向调用该函数的对象。因此，**this指针是一个隐含的参数**。

return by reference 语法分析

传递者无需知道接收者是以reference还是value的方式接收的。

返回值是complex&，是为了能够连续调用函数。

```
complex c1(2, 1);
complex c2(4, 2);
c1 += c2 += c1;
```

为什么参数传递要使用reference？

- reference传递参数的效率比value传递参数的效率高，原因是reference传递参数不需要复制参数，而value传递参数需要复制参数。

操作符重载（2），非成员函数

```
class complex
{
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {}
    complex& operator+=(const complex&);
    double real() const { return re; }
    double imag() const { return im; }
private:
    double re, im;
    friend complex& __doapl(complex*, const complex&);
};
inline complex&
__doapl(complex* ths, const complex& r)
{
    ths->re += r.re;
    ths->im += r.im;
    return *ths;
}
inline complex&
complex::operator+=(const complex& r)
{
    return __doapl(this, r);
}

inline complex
operator+(const complex& x, const complex& y)
{
    return complex(real(x) + real(y), imag(x) + imag(y));
}

inline complex
operator+(const complex& x, double y)
{
    return complex(real(x) + y, imag(x));
}

inline complex
operator+(double x, const complex& y)
{
    return complex(x + real(y), imag(y));
}
```

- 上述代码中，**operator+是一个非成员函数**。因此，**该函数没有this指针**。
- 上面三个函数绝不可能return by reference，因为它们的返回值是一个临时对象，一出函数就消亡。在函数外界定义的对象，才可以return by reference。而且对于常量对象，应该return by reference to const。

- 临时对象：typename(); 上面三个函数的返回值都是complex()，因此，它们的返回值都是临时对象。

重载操作符的函数通过参数的个数和类型来区分不同的操作符。

```
class complex
{
public:
    complex(double r = 0, double i = 0) : re(r), im(i) {}
    complex& operator+=(const complex&);
    double real() const { return re; }
    double imag() const { return im; }
private:
    double re, im;
    friend complex& __doapl(complex*, const complex&);
};
inline complex&
__doapl(complex* ths, const complex& r)
{
    ths->re += r.re;
    ths->im += r.im;
    return *ths;
}
inline complex&
complex::operator+=(const complex& r)
{
    return __doapl(this, r);
}
inline complex
operator+(const complex& x, const complex& y)
{
    return complex(real(x) + real(y), imag(x) + imag(y));
}
inline complex
operatoe==(const complex& x, const complex& y)
{
    return real(x) == real(y) && imag(x) == imag(y);
}
```

操作符的分类

- 一元操作符：只有一个操作数，如：+a, -a, !a, ~a, ++a, --a, a++, a--, (type)a, *a, &a, sizeof a
- 二元操作符：有两个操作数，如：a+b, a-b, a*b, a/b, a%b, a==b, a!=b, a>b, a>=b, a<b, a<=b, a&&b, a||b, a&b, a|b, a^b, a<<b, a>>b, a,b
- 三元操作符：有三个操作数，如：a?b:c

<<操作符重载（output operator）

```
#include <iostream>
using namespace std;
ostream& operator<<(ostream& os, const complex& x)
```

```
{
    return os << '(' << real(x) << ',' << imag(x) << ')';
}
```

- output operator必须写成非成员函数，因为<<左边是ostream对象，而不是complex对象。
- 只有操作符左边的参数是对象本身，才能写成成员函数。否则，必须写成非成员函数。
- 能写成成员函数的运算符，只有：=, [], (), ->, ++, --, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

C++中带指针的类的设计--String类

拷贝构造和拷贝赋值

系统自动生成的拷贝构造和拷贝赋值函数，只是简单的将对象的每个成员变量逐个复制。如果类中有指针成员，那么这两个函数就会出现问题。 **因此，如果类中有指针成员，那么必须自己定义拷贝构造和拷贝赋值函数。**

String类

```
class String
{
public:
    String(const char* cstr = 0);
    String(const String& str); // 拷贝构造函数
    String& operator=(const String& str); // 拷贝赋值函数
    ~String(); // 析构函数
    char* get_c_str() const { return m_data; }
private:
    char* m_data;//用一个指针而不是一个数组，是因为数组的大小是固定的，而String类的大小是不固定的。
};
```

三大函数

1. 拷贝构造函数

```
inline
String::String(const String& cstr=0)
{
    if(cstr)
    {
        m_data = new char[strlen(cstr.m_data) + 1]; // 为指针分配内存,具体大小为cstr.m_data的长度+1，因为要存放'\0'
        strcpy(m_data, cstr.m_data);
    }
    else // 未指定初值
    {
        m_data = new char[1];
        *m_data = '\0';
```

```
    }
}
```

2. 拷贝赋值函数

```
inline
String& String::operator=(const String& str)
{
    if(this == &str) // 判断是否是自我赋值
        return *this;
    delete[] m_data; // 释放原来的内存
    m_data = new char[strlen(str.m_data) + 1]; // 为指针分配内存,具体大小为
cstr.m_data的长度+1，因为要存放'\0'
    strcpy(m_data, str.m_data);
    return *this;
}
```

3. 析构函数

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

上述代码中，因为使用 new 分配了内存，所以必须使用 delete 释放内存。如果不释放内存，那么就会造成内存泄漏。

堆和栈

Stack

Stack,是**存在于某作用域内的一块内存区域**，用于**存放局部变量**。Stack的大小是有限的，一般是几百KB。Stack的生命周期是从函数调用开始，到函数调用结束。Stack的生命周期是由编译器自动管理的，不需要程序员手动管理。

Heap

Heap，是指由操作系统管理的一块内存区域，用于存放**动态分配的内存**。Heap的大小是没有限制的，一般是几GB。Heap的生命周期是从**new**开始，到**delete**结束。Heap的生命周期是由程序员手动管理的，不需要编译器自动管理。

heap objects 的生命周期

```
class A {};
A* p = new A; // 在heap上创建一个A对象
```

```
delete p; // 在heap上销毁一个A对象
```

上述代码是正确的，在同一个作用域中，new生成的对象，必须在同一个作用域中delete。

```
class A {};
A* p = new A;
```

上述代码会造成内存泄露，因为当作用域结束，p所指的heap object仍然存在，但是指针p的生命周期已经结束，所以p所指的heap object就无法访问了，作用域之外再也看不到p。

Stack和Heap的区别

- Stack是由编译器自动管理的，Heap是由程序员手动管理的。
- Stack的大小是有限的，Heap的大小是没有限制的。

new 的剖析

- new:**先分配内存，再调用构造函数**

```
Complex* p = new Complex(1, 2);
```

上述代码等价于

```
void* mem = operator new(sizeof(Complex)); // 分配内存，指向void*类型的指针
p = static_cast<Complex*>(mem); // 将void*类型的指针转换为Complex*类型的指针
p->Complex::Complex(1, 2); // 调用构造函数
```

delete 的剖析

- delete:**先调用析构函数，再释放内存**

```
String* ps = new String("hello");
...
delete ps;
```

上述代码等价于

```
String::~String(ps); // 调用析构函数
operator delete(ps); // 释放内存,其内部调用了free(ps)
```

array new 和 array delete

- array new 要和 array delete 配对使用

```
int* p = new int[10];
...
delete[] p; // 唤起10次dtor
delete p; // 错误，只唤起1次dtor，会造成内存泄漏（会导致数组中的对象的析构函数不被调用）
```