C++面向对象编程

0. 前置知识

1. C++代码的基本形式

- 声明 (header files)
- .cpp
- 标准库（header files）

header的一般写法

```
#ifndef __COMPLEX__ // 头文件的防卫式声明
#define __COMPLEX__

// 前置声明
class ostream;
class complex;

// 类的声明
class complex{
...
};

// 类的定义
complex::function...
#endif
```

延续上节课的知识，使用编译命令#ifndef，保证complex.h的内容在被多次include的情况下，不会被重复编译。

（#ifndef，即if not define的含义，在相应的宏未定义的情况下，才会编译其至#endif之间的代码）

通常来说，我们在.h头文件中编写类的声明，而在.cpp文件中编写类成员的定义。

2. 如何确认编译器支持的C++版本

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

int main(){
    cout<<__cplusplus<<endl;
    return 0;
}
```

设计复数类

首先，我们用复数的例子体现面向对象编程的抽象过程。

C++中提供了整数、浮点数等基本的数据类型，但是没有提供复数类型（尽管在STL中有了），我们现在设计一个复数类型，它应当支持如下操作：

```cpp
int main(){
  // 初始化
  complex c1();
  complex c2(4396);
  complex c3(7777, 7777.7);

  // 运算
  complex c4;
  c4 += c1;
  c3 = c1 + c2;

  // 输出
  cout << c4.real() << "," << c4.imag() << endl;
  cout << c4 << endl;
}
```

复数有实部和虚部，体现在两个数值数据上（这里我们使用double类型，如果希望泛用，可以参考Extra中的模板），记作`re`和`im`。

我们希望能够像int，double那样声明一个复数（`complex c1;`），或者在初始化时进行赋值（`complex c2(1,2.0)`），这个过程通过带有默认参数的构造函数实现。

接下来，复数应当可以进行加减乘除等运算，这里我们仅用+=作为例子，介绍运算符重载。

最后，我们考虑数据封装，外部调用者不应当直接修改数据成员`re`和`im`，所以我们将其设置为`private`，并提供`real()`和`imge()`的公有访问接口。

全部的代码如下：

```
#ifndef __COMPLEX__
#define __COMPLEX__

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& __do_assign_plus(complex *, const complex&);
};

ostream& operator << (ostream& os, const complex& c){
    os << "(" << c.real()<< "," << c.imag() << ")" << endl;
    return os;
}

#endif // __COMPLEX__
```

一些细节

1 函数若在`class`中完成定义，如上面代码中的real()，imge()，则自动成为inline的候选人，至于是否真的成为inline，由编译器决定

2 访问级别：`public`和`private`

1. `public`可以被外界调用
2. `private`只能在被类内定义的函数访问，诸如`c1.re`，这样的访问操作是不允许的

3 构造函数中使用了`initialization list`的语法，理论上来说，这样写要在大括号中写赋值操作要好。

```
// 1. 初始化列表
complex(double r = 0, double i = 0)
        : re (r), im (i)
    {  }

// 2. 在函数体中进行赋值
complex (double r = 0,double i = 0)
{ re = r; im = i; }
```

简单地解释，变量的数值设定有两个阶段：初始化，赋值。

使用初始化列表语法，则是在初始化阶段就完成赋值；

而在构造函数体中写，则是在先将变量初始化为默认值，再进行赋值，虽然结果一致，但是会造成效率损失

4 重载函数要避免二义性

```
// 构造函数 1
complex(double r = 0, double i = 0) : re (r), im (i) {  }

// 构造函数 2
complex () : re (0), im (0) { }
```

当使用 `complex c1;` 时，两个函数定义都适用，编译器不知道应该匹配哪一个重载。

5 const常量成员函数

```
double real() const
{
  ...
}
```

在类成员函数末尾的`const`标识，显式指出该函数不会更改类的数据。编程时，如果应当尽可能地使用`const`。

为什么呢？请看下面例子：

```
// example 1
complex c1(2,1);
std::cout<<c1.real();
std::cout<<c1.imge();

// example 2
const complex c2(2,1);
std::cout<<c2.real();    // 如果我们没有用const限定real()和imge()，则不能如此调用
std::cout<<c2.imge();    // 尽管使用者并没有修改数据的意图，编译器仍然认为real()可能会修改数据
```

6 参数传递：pass by value 和 pass by reference (to const)

```
// pass by value
complex(double r=0,double i=0);

// pass by reference to const
complex & operator += (const complex&);
```

在效率上，pass by value相当于传递一个临时变量的速度，而pass by reference相当于传递一个指针的速度。一般的原则是，尽可能使用pass by reference，从而提高效率。

（如果仔细考虑，某些变量可能小于4个字节，如char、short，这时传值的效率会高些）

如果在pass by reference时，希望调用者不修改原来的参数，则可以使用pass by reference to const

7 返回值传递：return by value 和 return by reference (to const)

```
// return by value
double real()const;

// return by reference
complex& operator += (const complex&)
```

8 (friend function)友元函数

友元函数可以直接访问类的私有成员

```
// 类内声明友元函数
friend complex & __do_assign_plus(complex *, const complex &);

// 类外定义
inline complex& __do_assign_plus(complex * ths, const complex & r);
{
    ths->re += r.re;
    ths->im += r.im;
    return *ths;
}
```

9 运算符重载，以`+=`为例

首先考虑+=的用法：`c2 += c1`;形式上看，+=运算符有两个操作数，c2和c1，但我们将运算符重载作为成员函数时，之前的调用可以被视为`c2.operator+=(c1)`，所以形式上，+=只需要接收它的右操作数即可。

```
// 类内声明
complex& operator += (const complex&);

// 定义
inline complex&
complex::operator += (const complex& rhs)
{
    return __do_assign_plus(this, rhs);
}
```

但有些时候，必须返回临时对象：

```
// 类内声明
complex complex::operator + (const complex& rhs) const;

// 定义
inline complex
complex::operator + (const complex& rhs) const
{
    return complex(this->re + rhs.re,
                   this->im + rhs.im);
}
```

但如果想要重载`<<`运算符，就要在类外写全局函数实现：

```
ostream& operator << (ostream& os, const complex& c){
    os << "(" << c.real()<< "," << c.imag() << ")" << endl;
    return os;
}
```

Extra

1. 使用模板

上面我们给出了一个浮点数复数类，但是如果我们又想要一个整数类，那么我们是否要重新写一遍呢？

C++提供template（模板），来解决这种问题，例子如下：

```
template<class T>
class complex
{
public:
    complex(double r = 0, double i = 0)
        : re (r), im (i)
    {  }
    complex& operator += (const complex&);
    T real () const { return re;}
    T imag () const { return im;}
private:
    T re, im;

    friend complex& __doapl(complex *, const complex&);
};

// usage
complex<int> c1(2,1);
complex<double> c2;
```

2. 构造函数放在private——Singleton设计模式

Singleton，单件，当我们希望某个类在全局有且只有一个实例时，可以使用该设计模式。

该设计模式通常用来实现工厂类。

```
class A{
public:
    static A & getInstance();
    setup() {}
private:
    A();
    A(const A&rhs);
    ...
};

A & A::getInstance()
{
    static A a;
    return a;
}
```

```
// 调用方法：
A a = A::getInstance();
a.setup();
```