C++中有一个重要特性，那就是模板类型。类似于Objective-C中的泛型。C++通过类模板来实现泛型支持。

1 基础的类模板

类模板，可以定义相同的操作，拥有不同数据类型的成员属性。

通常使用template来声明。告诉编译器，碰到T不要报错，表示一种泛型.

如下，声明一个普通的类模板：

template <typename T>

class Complex{

public:

//构造函数

Complex(T a, T b)

{

this->a = a;

this->b = b;

}

//运算符重载

Complex<T> operator+(Complex &c)

{

Complex<T> tmp(this->a+c.a, this->b+c.b);

return tmp;

}

private:

T a;

T b;

}

int main()

{

//对象的定义，必须声明模板类型，因为要分配内容

Complex<int> a(10,20);

Complex<int> b(20,30);

Complex<int> c = a + b;

return 0;

}

2 模板类的继承

在模板类的继承中，需要注意以下两点：

如果父类自定义了构造函数，记得子类要使用构造函数列表来初始化

继承的时候，如果子类不是模板类，则必须指明当前的父类的类型，因为要分配内存空间

继承的时候，如果子类是模板类，要么指定父类的类型，要么用子类的泛型来指定父类

template <typename T>

class Parent{

public:

Parent(T p)

{

this->p = p;

}

private:

T p;

};

//如果子类不是模板类，需要指明父类的具体类型

class ChildOne:public Parent<int>{

public:

ChildOne(int a,int b):Parent(b)

{

this->cone = a;

}

private:

int cone;

};

//如果子类是模板类，可以用子类的泛型来表示父类

template <typename T>

class ChildTwo:public Parent<T>{

public:

ChildTwo(T a, T b):Parent<T>(b)

{

this->ctwo = a;

}

private:

T ctwo;

};

3 内部声明定义普通模板函数和友元模板函数

普通模板函数和友元模板函数，声明和定义都写在类的内部，也不会有什么报错。正常。

template <typename T>

class Complex {

//友元函数实现运算符重载

friend ostream& operator<<(ostream &out, Complex &c)

{

out<<c.a << " + " << c.b << "i";

return out;

}

public:

Complex(T a, T b)

{

this->a = a;

this->b = b;

}

//运算符重载+

Complex operator+(Complex &c)

{

Complex temp(this->a + c.a, this->b + c.b);

return temp;

}

//普通加法函数

Complex myAdd(Complex &c1, Complex &c2)

{

Complex temp(c1.a + c2.a, c1.b + c2.b);

return temp;

}

private:

T a;

T b;

};

int main()

{

Complex<int> c1(1,2);

Complex<int> c2(3,4);

Complex<int> c = c1 + c2;

cout<<c<<endl;

return 0;

}

4 内部声明友元模板函数+外部定义友元模板函数

如果普通的模板函数声明在内的内部，定义在类的外部，不管是否处于同一个文件，就跟普通的函数一样，不会出现任何错误提示。但是如果是友元函数就会出现报错，是因为有二次编译这个机制存在。

4.1 模板类和模板函数的机制

在编译器进行编译的时候，编译器会产生类的模板函数的声明，当时实际确认类型后调用的时候，会根据调用的类型进行再次帮我们生成对应类型的函数声明和定义。我们称之为二次编译。同样，因为

这个机制，会经常报错找不到类的函数的实现。在模板类的友元函数外部定义时，也会出现这个错误。解决方法是 “ 类的前置声明和函数的前置声明 ”。

按照普通模板函数的样式处理友元函数

#include <iostream>

using namespace std;

template <typename T>

class Complex {

//友元函数实现运算符重载

friend ostream& operator<<(ostream &out, Complex<T> &c);

public:

Complex(T a, T b);

//运算符重载+

Complex<T> operator+(Complex<T> &c);

//普通加法函数

Complex<T> myAdd(Complex<T> &c1, Complex<T> &c2);

private:

T a;

T b;

};

//友元函数的实现

template <typename T>

ostream& operator<<(ostream &out, Complex<T> &c)

{

out<<c.a << " + " << c.b << "i";

return out;

}

//函数的实现

template <typename T>

Complex<T>::Complex(T a, T b)

{

this->a = a;

this->b = b;

}

template <typename T>

Complex<T> Complex<T>::operator+(Complex<T> &c)

{

Complex temp(this->a + c.a, this->b + c.b);

return temp;

}

template <typename T>

Complex<T> Complex<T>::myAdd(Complex<T> &c1, Complex<T> &c2)

{

Complex temp(c1.a + c2.a, c1.b + c2.b);

return temp;

}

int main()

{

Complex<int> c1(1,2);

Complex<int> c2(3,4);

Complex<int> c = c1 + c2;

cout<<c<<endl;

return 0;

}

友元函数的定义写在类的外部--错误信息

Undefined symbols for architecture x86\_64:

"operator<<(std::\_\_1::basic\_ostream<char, std::\_\_1::char\_traits<char> >&, Complex<int>&)", referenced from:

\_main in demo1.o

ld: symbol(s) not found for architecture x86\_64

clang: error: linker command failed with exit code 1 (use -v to see invocation)

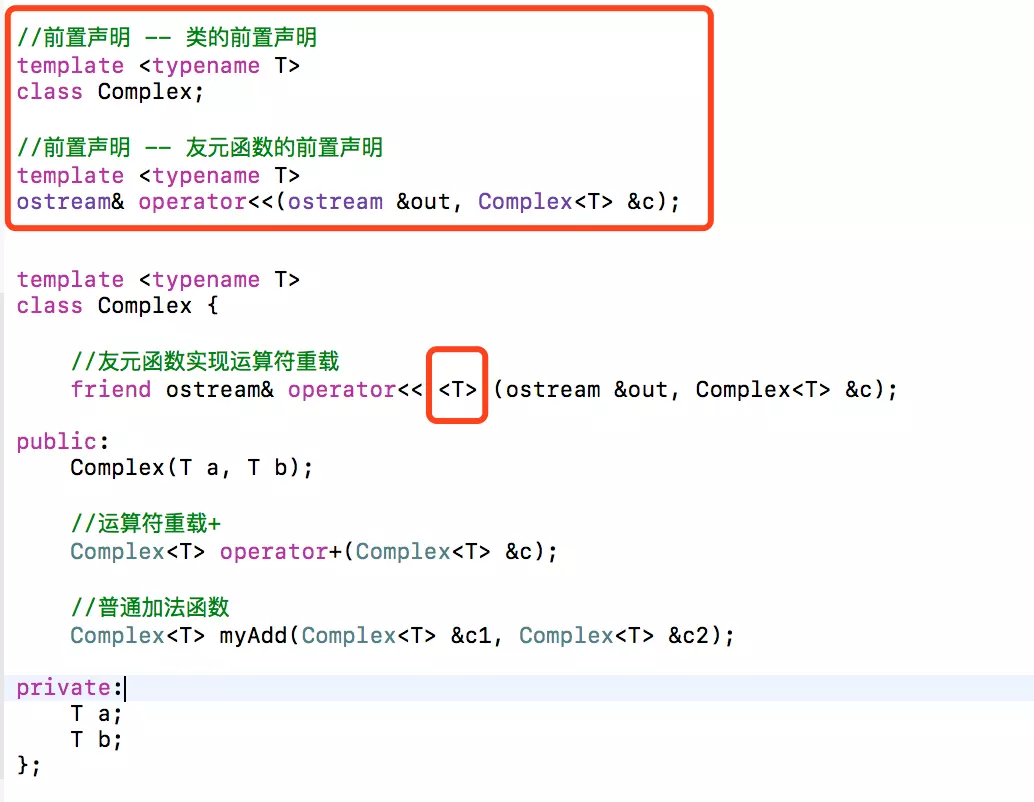
上面的错误信息，就是典型的二次编译的错误信息，找不到友元函数的函数实现。所以，如果友元模板函数的定义写在函数的外部，需要进行类和函数的前置声明，来让编译器找到函数的实现

4.2 前置声明解决二次编译问题

类的前置声明

友元模板函数的前置声明

友元模板函数声明需要增加泛型支持



5 声明和定义分别在不同的文件（模板函数、模板友元）

类的声明和实现，分别在不同的文件下，需要增加一个hpp文件支持。或者尽量将模板函数与模板友元放在一个文件下。

类的声明与函数的声明写在.h文件

类的实现及函数的实现写在.cpp文件

将.cpp文件改成.hpp文件

在主函数中调用.hpp文件，而不是引用.h文件

如果碰到.h和.hpp文件都存在的情况下，引用.hpp文件。

demo2.h文件

存放类的声明和函数的声明

#include <iostream>

using namespace std;

//类的前置声明

template <typename T>

class Complex;

//友元函数的声明

template <typename T>

ostream& operator<<(ostream &out, Complex<T> &c);

template <typename T>

class Complex {

//友元函数实现运算符重载

friend ostream& operator<< <T> (ostream &out, Complex<T> &c);

public:

Complex(T a, T b);

//运算符重载+

Complex<T> operator+(Complex<T> &c);

//普通加法函数

Complex<T> myAdd(Complex<T> &c1, Complex<T> &c2);

private:

T a;

T b;

};

demo2.hpp文件

包括模板函数的实现

#include "demo2.h"

//友元函数的实现

template <typename T>

ostream& operator<<(ostream &out, Complex<T> &c)

{

out<<c.a << " + " << c.b << "i";

return out;

}

//函数的实现

template <typename T>

Complex<T>::Complex(T a, T b)

{

this->a = a;

this->b = b;

}

template <typename T>

Complex<T> Complex<T>::operator+(Complex<T> &c)

{

Complex temp(this->a + c.a, this->b + c.b);

return temp;

}

template <typename T>

Complex<T> Complex<T>::myAdd(Complex<T> &c1, Complex<T> &c2)

{

Complex temp(c1.a + c2.a, c1.b + c2.b);

return temp;

}

main.cpp文件

需要调用hpp文件

#include <iostream>

using namespace std;

#include "demo2.hpp"

int main()

{

Complex<int> c1(1,2);

Complex<int> c2(3,4);

Complex<int> c = c1 + c2;

cout<<c<<endl;

return 0;

}