1. 类模板

写法：  
template<class NameType, class AgeType>

Class Person

{

Public:

Person(NameType name, AgeType age)

{

This-> m\_Name = name;

This->m\_Age = age;

}

NameType m\_Name;

AgeType m\_Age;

}

类模板不支持自动类型推导，需要显式指定类型：  
Person<string,int> p(“大哥”, 100);

类模板的类型可以有默认参数，但是函数模板没有默认参数

template<class NameType, class AgeType=int>

Class Person{}

1. 当类模板作为函数的参数

类模板作为函数的参数有以下几种传参的形式：

1. 指定传入类型

指定要传入的类模板的类型，如下所示：

Void doWork(Person<string, int> & p ) {}

在传入的类模板中指定了类模板的参数类型。

1. 函数模板化

把类模板作为函数的参数，并且用函数模板来实现，类模板的参数类型作为函数模板的参数，函数模板可以自动推导参数的类型。

Template<class T1, class T2>

Void doWork(Person<T1, T2> & p){}

1. 将类模板整体作为一个函数参数类型

将整个类模板都作为函数模板的一个参数类型。

Template<class T>

Void doWork(T & p) {}

代码如下：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include<string>

using namespace std;

//类模板

template<class NameType, class AgeType>

class Person

{

public:

Person(NameType name, AgeType age)

{

this->m\_Name = name;

this->m\_Age = age;

}

void PrintPerson()

{

cout << "Name: " << this->m\_Name << "Age: " << this->m\_Age << endl;

}

public:

NameType m\_Name;

AgeType m\_Age;

};

//类模板作为函数参数

//1.指定模板类型

void DoBus1(Person<string, int>& p)

{

p.m\_Age += 20;

p.m\_Name += " vip";

p.PrintPerson();

}

//2.类模板类型作为函数模板类型

template<class T1, class T2>

void DoBus2(Person<T1,T2>& p)

{

p.m\_Age += 30;

p.m\_Name += " vip";

p.PrintPerson();

}

//3.类模板整体作为函数模板的参数类型

template<class T >

void DoBus3(T& p)

{

p.m\_Age += 40;

p.m\_Name += " vip";

p.PrintPerson();

}

void test()

{

Person<string, int> p("jack", 10);

DoBus2(p);

}

int main()

{

test();

return 0;

}

三． 继承类模板的问题

当子类继承一个类模板时，我们必须要为类模板指定参数类型，不然类模板无法为参数类型分配指定的内存空间。因为，如果不指定参数类型，那么类模板不知道应该分配多少内存给参数。

1. 在子类中直接指定类型

Template<class T>

Class Base{}

Class Child : public Base<int>{}

子类后面指定了<int>类型，那么基类模板类就知道参数类型T为int。

1. 子类同样为类模板，无须直接指定参数类型

Template<class T>

Class Base{}

Template<class T1, class T2>

Class Child2 : public Base<T2>

{}

可以看到，子类为类模板，T1为子类的参数类型，T2为基类类模板的参数类型。这两个参数类型都可以在用户调用时指定。

四．类模板类外实现成员函数

如何在类模板的类外实现成员函数呢？

与普通类的类外实现成员函数一样，我们需要添加类的作用域。但不同的是，类模板的作用域需要通过类模板的定义来实现。

Template<class T1, class T2>

Void Person<T1,T2>::showPerson(){}

五． 类模板的分文件编写问题

当我们将类模板分别写到 Person.h文件和Person.cpp文件中，Person.h只是声明，Person.cpp实现类模板，最终实现会出现错误。

因为，C++是单元编译的，所谓单元编译是对每一个文件分别编译的，因此分别对Person.cpp编译，对Person.h编译。语法当然是没有问题的，但是在Person.h中，类模板的成员函数一开始是不会创建的，因为确定不了具体的类型参数。因此，即使在类模板实现的cpp文件中引入了Person.h头文件，但是Person.cpp中成员函数的代码并没有生成，也就无法链接到Person.cpp的成员函数代码。

如何解决：

建议类模板不要分文件编写。将Person.cpp中的代码都写到Person.h中，即类模板在类内声明，在类外实现。同时，将Person.h的后缀名改为Person.hpp。.hpp一般用于写模板所用。

Person.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T1, class T2>

class Person

{

public:

Person(T1, T2);

void showPerson();

T1 m\_name;

T2 m\_age;

};

template<class T1, class T2>

Person<T1, T2>::Person(T1 name, T2 age)

{

this->m\_name = name;

this->m\_age = age;

}

template<class T1, class T2>

inline void Person<T1, T2>::showPerson()

{

cout << "姓名： " << this->m\_name << "年龄： " << this->m\_age << endl;

}

六. 类模板的友元函数

1.直接在类模板内声明并定义友元函数

在类模板内直接声明定义友元函数，则该友元函数默认为全局函数，可以直接调用。

Template<class T1,class T2>

Class Person

{  
 friend void printPerson( Person<T1,T2> &p) {}

}

2.友元函数类模板外实现

友元函数在类模板内声明，在类模板外实现，需要的步骤比较多。

如果我们在类模板内声明了友元函数如下所示：

Template<class T1,class T2>

Class Person

{  
 friend void printPerson( Person<T1,T2> &p) {}

}

而在类模板外定义了友元函数，由于T1，T2在类模板外是未定义的，因此需要加上模板。如下所示：

Template<class T1, class T2>

Void printPerson(Person<T1, T2> &p) {}

但是可以很明显地看出，类模板内声明地友元函数是普通函数，而类外实现地友元函数是一个模板函数，这明显是不匹配的。

1. 友元函数声明为模板函数

因此，我们需要为类模板内声明的友元函数添加<>，如下所示：

friend void printPerson<>( Person<T1,T2> &p) {}

这样，就告诉了编译器这是模板函数的声明，这样两个函数就匹配了。

2.提前声明模板函数，类模板

但是，这样还是不行，由于在类模板外实现模板函数，而友元函数声明是在类模板的内部，因此，我们需要在类外提前声明该模板函数，同时，该模板函数声明中的Person类模板也是未知的，也需要提前声明。

Template<class T1, class T2> class Person;

Template<class T1, class T2> void printPerson(Person T1, Person T2)

七. 类模板封装小型数组

#pragma once

#include<iostream>

using namespace std;

template<class T>

class MyArray

{

public:

//构造

explicit MyArray(int capacity) //防止隐式类型转换

{

this->m\_Capacity = capacity;

this->m\_Size = 0;

this->pAddress = new T[this->m\_Capacity]; //开辟堆区空间

}

//拷贝构造

MyArray(const MyArray& array)

{

this->m\_Capacity = array.m\_Capacity;

this->m\_Size = array.m\_Size;

this->pAddress = new T[this->m\_Capacity];

//获取拷贝数据

for (int i = 0; i < this->m\_Size; ++i)

{

this->pAddress[i] = array[i];

}

}

//析构

~MyArray()

{

//堆空间释放

if (this->pAddress != NULL)

{

delete[]this->pAddress;

this->pAddress = NULL;

}

}

// =重载

MyArray& operator=(MyArray& array)

{

//判断原始数据是否存在，如果存在就清空

if (this->pAddress != NULL)

{

delete[]this->pAddress;

this->pAddress = NULL;

}

this->m\_Capacity = array.m\_Capacity;

this->m\_Size = array.m\_Size;

this->pAddress = new T[this->m\_Capacity];

for (int i = 0; i < this->m\_Size; ++i)

{

this->pAddress[i] = array[i];

}

}

// []重载

T& operator[](int index)

{

return this->pAddress[index];

}

//尾插法

void push\_back(T val)

{

this->pAddress[this->m\_Size] = val;

this->m\_Size++;

}

//获取数组大小

int getSize()

{

return this->m\_Size;

}

//获取数组容量

int getCap()

{

return this->m\_Capacity;

}

private:

T\* pAddress; //指向堆区的指针

int m\_Capacity; //容量

int m\_Size; //大小

};