#ifndef Color\_H

#define Color\_H

#include <string>

#include <type\_traits>

#include <utility>

class Color{

private:

std::string \_value;

public:

Color(std::string value):\_value(value){};

std::string getValue() const{ return \_value; }

};

// 函数模板

// TColor 为模板参数名称，TColor 可以时任意类型，但必须实现 getValue，否则在编译时报错

// 多个模板参数 template<typename TColor, typename TColor1>

template<typename TColor>

bool IsEqualOfColor(TColor leftColor, TColor rightColor)

{

return leftColor.getValue() == rightColor.getValue();

}

// 非类型模板参数

// 非类型模板参数表示一个值，由编译器推断的值代替该参数

// 这里将list声明为引用，因为我们无法拷贝一个数组

template<int TSize>

int GetSize(const int (&list)[TSize]){

return TSize;

}

// 通过尾置返回，我们可以指定返回类型

// 这里返回类型是 string

template<typename T>

auto GetColorValue(T color) -> decltype(color.getValue());

// 这种情况下，传给f1的实参必须时左值

template<typename T>

void f1(T &t){};

// 这种情况下，传给f2的实参可以是左值，也可以是右值

// 如 f2(5) 时，5可以被绑定到 const int & 上，所以 T 被推断为 int

// 以下代码没毛病：

// const int &t = 5;

// int &&b = 5; const int &c = b;

template<typename T>

void f2(const T &t){};

// 这种情况下，传给f3的实参即可以时左值，也可以是右值

// 当 t 为 20 时，T 的被推断为 int，（int &&t = 20 ，没毛病）

// 当 f3(a) a 为 int 时，问题出现了（int a = 1; int &&t = a;），左值无法赋值给右值

// 当出现这种情况时，有两个例外规则

// 例外1：

// 当 t 为 int a 时，此时编译器推断 T 为 int&，则 t 的类型为 int& &&

// 当 t 为 int &&b 时，此时编译器推断 T 为 int&&，则 t 的类型为 int&& &&

// 例外2：

// T& &, T& &&, T&& & 类型将被折叠成 T&

// T&& &&，将被折叠成 T&&

// 这两个例外规则暗示我们可以将任何类型传给 T&& 模板类型参数

template<typename T>

void f3(T &&t){};

// 类型转换

// 针对类型的操作

template<typename T>

void TypeConvert(T &&t){

// 移除T的引用类型

// 如：T为int& 或 int&&，则typename remove\_reference<T>::type返回int

// 则下面的代码为 int & a;

typename std::remove\_reference<T>::type & a;

}

// 关于move标准库函数原理

template<typename T>

typename std::remove\_reference<T>::type&& move(T&& t){

return static\_cast<typename std::remove\_reference<T>::type&&>(t);

}

// 使用forward保持类型信息(#include <utility>)

// forward返回给的类型的右值引用

template<typename T>

void f4(T &&t){

// f4(5)时，T为int，则std::forward<T>(t)返回 t 的类型为 int&&

// f4(int)时，T为int&，则std::forward<T>(t)返回 t 的类型为 int& &&

// f4(int&&)时，T为int&&，则std::forward<T>(t)返回 t 的类型为 int&& &&

f3(std::forward<T>(t));

};

// 可变参数模板

template<typename T>

void f5(T t){

}

template<typename T, typename... Args>

void f5(T t, Args ... args){

sizeof...(args); // 返回参数包参数的数量

// ...为扩展动作，此动作会将参数和类型进行扩展

// 例如：

// f5(4, "a", 'b')

// 那么 t 为 4， args 为 { "a", 'b' }

// f5(args...)后代码为 f5("a", 'b')

// 这会调用 void f5(T t, Args ... args)

// 那么 t 为 "a"， args 为 { 'b' }

// f5(args...)后代码为 f5('b')

// 这会调用 void f5(T t)

f5(args...);

// 如果 f5(1, 1.1)

// 执行 f6(forward<Args>(args)...)

// 那么 f6(forward<int>(1), forward<double>(1.1))

}

#endif

#ifndef Pan\_H

#define Pan\_H

#include <iostream>

#include <string>

#include "./Color.h"

// SetColor1作为友元时，必须进行声明

template <typename>

void SetColor1();

// 模板的头文件通常即包含声明，也包含定义

// 类模板

// 编译器不能为类模板推断模板参数类型

// = Color 为指定默认实参

template <typename TColor = Color>

class Pan

{

// 只应用于特定类型的友元

// 例如SetColor1<Color>只能应用于Pan<Color>

// 注意：这是一个模板实例，所以必须有前置声明

friend void SetColor1<TColor>();

// 能应用于所有类型的友元

friend void SetColor2();

// 友元模板声明

// SetColor3的所有实例都是Pan模板的友元

template <typename X>

friend void SetColor3();

private:

TColor \_color;

public:

Pan() = default;

Pan(TColor color) : \_color(color){};

TColor GetColor() const;

// 对于给定的模板参数类型，都只有一个Instance，不同模板参数类型的 Instance 不同

// 例：TColor 为 Color 的类型只有一个 Instance

static int Instance;

// 成员模板

// 成员模板可以自定义函数的模板参数

template <typename T>

void SetMoney(T t, TColor color);

};

// 类外定义模板成员

template <typename TColor>

TColor Pan<TColor>::GetColor() const

{

return \_color;

}

// 特例化成员函数

// 特例化可以应用于类或函数上

// 通过 template <> 来声明一个特例化实例

// 当 TColor 为 Color 将使用下面的函数而不是模板函数

template <>

Color Pan<Color>::GetColor() const

{

return \_color;

}

// 部分特例化

// 部分特例化只能应用于类

// 当TColor是引用类型时将使用下面的模板

template <typename TColor>

class Pan<TColor&>{

};

// 在类外定义成员模板时必须指定类模板参数类型

template <typename TColor>

template <typename T>

void Pan<TColor>::SetMoney(T t, TColor color)

{

}

#endif

#include <iostream>

#include <vector>

#include <type\_traits>

#include "Color.h"

#include "Pan.h"

using namespace std;

// 为类模板定义别名

// 定义各个 模板 APan，类型是 Pan<T>

template<typename T> using APan = Pan<T>;

// 我们每使用一个 IsEqualOfColor<string> ，编译器就会帮我们实例一个，

// 使用 extern template 声明一个实例，编译器便不会帮我们生成这个实例，但必须确保在使用之前该模板类型被实例

extern template bool IsEqualOfColor<string>(string, string);

int main()

{

Color color1("#fff");

Color color2("#fff");

// 使用模板函数

// 当我们指定一个TColor的类型时，编译器就会为我们生成一个对应的函数

// 例如：TColor为Color，则编译器为我们生成

// bool IsEqualOfColor(Color leftColor, Color rightColor)函数

// 而我们在 IsEqualOfColor 中用到的 getValue 方法，如果 Color 未定义改方法，则编译会报错

if(IsEqualOfColor<Color>(color1, color2)){

cout << "equal" << endl;

}

// 类模板对象

Color color3("#fff");

Pan<Color> pan1(color3);

// 默认情况下，模板的成员函数只有在用到时，编译器才会去实例成员函数的一个对应类型

// 这一特性告诉我们，我们指定的模板参数不需要完全符合我们类的需求，只要符合我们调用的成员函数的需求即可

Color color4 = pan1.GetColor();

// 使用默认模板实参

Pan<> pan2();

cout << "enter key" << endl;

cin.get();

return 0;

}

// 类型推断

void TypeInfer(){

Color color1("#fff");

Color color2("#fff");

// 不指定模板参数也可以，编译器会使用实参的类型进行推断

// 编译器可以从参数类型推断模板参数类型，但无法从返回类型推断模板参数类型

// 如果IsEqualOfColor的返回类型是TResult，则必须显示指定TResult的类型

if(IsEqualOfColor(color1, color2)){

cout << "equal" << endl;

}

// 当我们用函数模板初始化一个函数指针时，模板参数类型从函数指针推断

bool (\*funp)(Color, Color) = IsEqualOfColor;

// list 的数量为2，所以编译器推断 TSize 为 2

int list[] = {1, 3};

cout << GetSize(list) << endl; // 输出2

}