type trait：处理template中类型

例：程序stl\_test20

template <typename T>

void FooImpl(const T& value, *true\_type*)

{

*cout* << "Foo() called for pointr to " << \*value << *endl*;

}

template <typename T>

void FooImpl(const T& value, *false\_type*)

{

*cout* << "Foo() called for value to " << value << *endl*;

}

template <typename T>

void Foo(const T& value)

{

// is\_pointer<T>可以判断Type是否为指针

FooImpl(value, *is\_pointer*<T>());

}

int main(int argc, char\* argv[])

{

int n = 5;

Foo<int>(n);

int\* ptr = new int(4);

Foo<int\*>(ptr);

delete ptr;

return 0;

}

输出：

Foo() called for value to 5

Foo() called for pointr to 4

type trait在处理一些繁琐的类型重载中很有作用。

针对整数类型的弹性重载

常规的针对整数类型的重载：

// 整型

void Foo(short);

void Foo(int)

void Foo(unsigned short)

// 浮点型

void Foo(float)

void Foo(double)

使用type trait

// 整型

template <typename T>

void FooImpl(T val, true\_type)

// 浮点型

template <typename T>

void FooImpl(T val, false\_type)

template <typename T>

void Foo(T val)

{

FooImpl(val, std::is\_integral<T>());

}

参考C++标准库 第2版.pdf，5.4.2节type trait可以处理非常多的类型判断。

外覆器：wrapper

1. 引用外覆器：reference\_wrapper

例：程序stl\_test21

class Test

{

public:

Test(int a) : number\_(a)

{

}

public:

void Add()

{

++number\_;

}

void Print() const

{

*cout* << "number is " << number\_ << *endl*;

}

private:

int number\_;

};

int *main*(int argc, char\* argv[])

{

Test t(1);

// vector<Test&> coll; // error，引用不能作为STL容器的类型

// coll中的元素将被当成引用对待

*vector*<*reference\_wrapper*<Test>> coll;

*vector*<Test> val;

coll.*push\_back*(t);

val.*push\_back*(t);

t.Add();

t.Print(); // 2

for (auto m : coll)

{

// m.get()返回的是Test&类型

m.*get*().Print(); // 2

}

for (auto n : val)

{

n.Print(); // 1

}

return 0;

}

输出为：

number is 2

number is 2

number is 1

函数外覆器：Function Type Wrapper

std::function<>：可以把“可调用对象”当作对象来使用。

例：程序stl\_test22

// Copyright 2017.刘珅珅

// author：刘珅珅

// 函数外覆器：wrapper

#include <functional>

#include <iostream>

#include <vector>

using namespace *std*;

void Func(int x, int y)

{

*cout* << "function x: " << x << " y: " << y << *endl*;

}

class A {

private:

// C++11允许非静态成员在其声明处进行初始化

int i = 0;

public:

void MemberFunc(int x, int y) const

{

*cout* << "member function x: " << x << " y: " << y << *endl*;

}

};

int *main*(int argc, char\* argv[])

{

*vector*<function<void(int, int)>> tasks;

tasks.*push\_back*(Func);

tasks.*push\_back*([](int x, int y) {

*cout* << "lambda x: " << x << " y: " << y << *endl*;

});

for (function<void(int, int)> f : tasks)

{

f(33, 66);

}

// 类成员函数的外覆器

function<void(int, int)> member\_func\_wrapper;

A a;

member\_func\_wrapper = *bind*(&A::MemberFunc, a, *std*::*placeholders*::*\_1*, *std*::*placeholders*::*\_2*);

member\_func\_wrapper(33, 66);

return 0;

}

输出：

function x: 33 y: 66

lambda x: 33 y: 66

member function x: 33 y: 66