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

辅助函数：

例：程序stl\_test23

// Copyright 2017.刘珅珅

// author：刘珅珅

// 辅助函数：min, max, minmax

#include <algorithm>

#include <iostream>

using namespace *std*;

// 通过指向的值比较两个指针大小

bool int\_ptr\_less(int\* a, int\* b)

{

return \*a < \*b;

}

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

{

int x = 17;

int y = 42;

int z = 33;

int tmp = 99;

int\* ptr\_x = &x;

*cout* << "pointer x " << ptr\_x << *endl*;

int\* ptr\_y = &y;

*cout* << "pointer y " << ptr\_y << *endl*;

int\* ptr\_z = &z;

*cout* << "pointer z " << ptr\_z << *endl*;

// 两个实参的max,min,minmax返回的是引用

// 在本例中返回的是，指针的引用而且是const引用

// ptr\_max引用的是ptr\_y，当修改了ptr\_y的指向后

// ptr\_max的指向也随之修改

int\* const& ptr\_max = *std*::*max*(ptr\_x, ptr\_y, int\_ptr\_less);

*cout* << "pointer max " << ptr\_max << *endl*;

ptr\_y = &tmp;

*cout* << "pointer y " << ptr\_y << *endl*;

*cout* << "pointer max " << ptr\_max << *endl*;

// 实参为初值列表版本返回的是比较值的拷贝

int\* ptr\_min = *std*::*min*({ ptr\_x, ptr\_y }, int\_ptr\_less);

*pair*<int\*, int\*> extremes = *minmax*({ ptr\_x, ptr\_y, ptr\_z }, int\_ptr\_less);

// first为最小值

*cout* << "pointer min " << extremes.*first* << *endl*;

// second为最大值

*cout* << "pointer max " << extremes.*second* << *endl*;

int i = 0;

long k = 9;

// std::max(i, k); // error，实参类型不一致

*std*::*max*<long>(i, k); // ok，声明template实参类型，确定返回类型

return 0;

}

输出为：

pointer x 008FFC98

pointer y 008FFC8C

pointer z 008FFC80

pointer max 008FFC8C

pointer y 008FFC74

pointer max 008FFC74

pointer min 008FFC98

pointer max 008FFC74

辅助函数：swap

例：程序stl\_test24

// Copyright 2017.刘珅珅

// author：刘珅珅

// swap：两值互换

#include <algorithm>

#include <iostream>

#include <vector>

using namespace *std*;

class Container

{

public:

Container(int number, *string* str)

{

number\_ = number;

str\_ = str;

}

void swap(Container& x)

{

*std*::*swap*(number\_, x.number\_);

*std*::*swap*(str\_, x.str\_);

}

void Print()

{

*cout* << "number " << number\_ << " str " << str\_.*c\_str*() << *endl*;

}

private:

int number\_;

*string* str\_;

};

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

{

int a[5] = { 1, 2, 3, 4, 5 };

int b[5] = { 6, 7, 8, 9, 10 };

*std*::*swap*(a, b);

*cout* << "array a" << *endl*;

for (int i : a)

{

*cout* << i << " ";

}

*cout* << *endl*;

*cout* << "array b" << *endl*;

for (int j : b)

{

*cout* << j << " ";

}

*cout* << *endl*;

Container c1(5, "hello");

Container c2(6, "world");

*cout* << "before swap " << *endl*;

c1.Print();

c1.swap(c2);

*cout* << "after swap" << *endl*;

c1.Print();

return 0;

}

输出：

array a

6 7 8 9 10

array b

1 2 3 4 5

before swap

number 5 str hello

after swap

number 6 str world

在一些比较复杂的类型中，可以提供swap函数进行交换特定的内部成员，而不是通过对象赋值的方式，可以大量节省时间。

ratio：编译期分数运算

例：程序stl\_test25

typedef *ratio*<5, 3> FiveThirds;

*cout* << FiveThirds::*num* << "/" << FiveThirds::*den* << *endl*; // 5/3

typedef *ratio*<25, 15> AlsoFiveThirds;

*cout* << AlsoFiveThirds::*num* << "/" << AlsoFiveThirds::*den* << *endl*; // 5/3

*ratio*<42, 42> one;

*cout* << one.*num* << "/" << one.*den* << *endl*; // 1/1

*ratio*<0> zero;

*cout* << zero.*num* << "/" << zero.*den* << *endl*; // 0/1

typedef *ratio*<7, -3> Neg;

*cout* << Neg::*num* << "/" << Neg::*den* << *endl*; // -7/3

输出：

5/3

5/3

1/1

0/1

-7/3

四则运算：

例：程序stl\_test25

// 四则运算

// 加

ratio\_add<*ratio*<2, 7>, *ratio*<2, 6>>::*type* sum;

*cout* << "sum " << sum.*num* << "/" << sum.*den* << *endl*; // 13/21:6/21 + 7/21

// 减

ratio\_subtract<*ratio*<2, 7>, *ratio*<2, 6>>::*type* diff;

*cout* << "diff " << diff.*num* << "/" << diff.*den* << *endl*; // -1/21:6/21 - 7/21

// 乘

ratio\_multiply<*ratio*<1, 2>, *ratio*<1, 4>>::*type* mult;

*cout* << "multiply " << mult.*num* << "/" << mult.*den* << *endl*; // 1/8:1/2 \* 1/4

// 除

ratio\_divide<*ratio*<1, 2>, *ratio*<1, 4>>::*type* div;

*cout* << "divide " << div.*num* << "/" << div.*den* << *endl*; // 2/1: 1/2 / 1/4

// 是否相等

if (*ratio\_equal*<*ratio*<5, 3>, *ratio*<25, 15>>::*value*)

{

*cout* << "equal" << *endl*;

}