A Research Paper Writing Service In 2021

a research paper writing service in 2021

This article is a continuation of the first part about templates and template functions in C ++.Pattern Member Functions

Class member function can also be template.For example, we have a Math class with a static ABS function that calculates the absolute value of the number:

STRUCT MATH {STATIC INT ABS (INT VALUE) {RETURN (Value <0? -Value: Value);}};

This implementation is only for type int, and after all, the parameters can be other types.Making the template all class does not make sense, so we will make the template only a member function:

#include <iostream> Struct Math {Template <TypeName T> Static T ABS (Const T

Value) {Return (Value <0? -Value: Value);}};int main () {std :: cout << Math :: ABS (-3.67) << STD :: ENDL;}

As you can see, everything is simple.The template may be not only static functions.But it is necessary to consider that virtual functions cannot be template:

struct My_class {template <TypeName T> Virtual void foo () {} // <- error};

Also template can be operators and designers:

#include <iostream> Template <TypeName T> Struct My_class {My_Class (Const T

Val = T ()): M_X (VAL) {} My_Class (Const My_Class

src): m_x (src.m_x) {} My_class

Operator = (Const My_Class

rhv) {m_x = rhv.m_x;} BOOL Operator == (Const My_Class

RHV) {RETURN M_X == RHV.M_X;} T GETX () const {Return M_X;} Private: T M_X;};INT MAIN () {My_class <int> OBJ1 (6);My_class <Short> Obj2 (6);STD :: COUT << OBJ1.GETX () << STD :: ENDL;STD :: COUT << OBJ2.GETX () << STD :: ENDL;}

My_class has one significant disadvantage - these are objects of absolutely different types.If we try

to compare OBJ1 and OBJ2 objects, we will get a compilation time error, because these objects compare the compiler is unknown.

Of course, you can compare the results of calling the GETX function, but it still does not solve all problems - you cannot assign one object to another or construct one object from the other (if different types are different).

To get out of the situation, we will define template versions of designers and operators.

#include <iostream> Template <TypeName T> struct My_class {// # 1 Template <TypeName U> Friend Class My_Class;TEMPLATE <TYPENAME U> MY_CLASS (Const U

Val = U ()): M_X (VAL) {} // # 2 My_Class (Const My_Class

SRC): M_X (src.m_x) {} // # 3 template <TypeName U> My_class (Const My_class <U>

src): m_x (src.m_x) {} My_class

Operator = (Const My_Class

rhv) {m_x = rhv.m_x;} Template <TypeName U> My_class

Operator = (Const My_Class <U>

rhv) {m_x = rhv.m_x;} Template <TypeName U> Bool Operator == (Const My_Class <U>

RHV) {RETURN M_X == RHV.M_X;} T GETX () const {Return M_X;} Private: T M_X;};INT MAIN () {My_class <int> OBJ1 (6);My_Class <Short> Obj2 (8);My_class <Double> OBJ3 (OBJ1);OBJ3 = OBJ2;STD :: COUT << (OBJ1 == OBJ2) << STD :: ENDL;STD :: COUT << (OBJ3 == OBJ1) << STD :: ENDL;}

As you can see, now transformations are possible, of course, if conversions are possible between M_X objects in the classes themselves, i.e.Write my_class <STD :: String> OBJ4 (OBJ3);In this case, it will not work, becauseThere is no appropriate conversion from type Double to Std :: String type.

As you might notice, I removed the non-sabrmed constructor and the comparison operator, leaving only their template versions.But it was possible to leave them to better organize work with the same types (for example, avoid overhead transformation costs).

Also, notice that in the class there remains a non-sabrped copy constructor (# 2), becauseIf it is removed, the compiler will generate it independently, i.e.The template constructor (# 3) does not replace the copy constructor (# 2) and if you do not define the copy constructor explicitly, the compiler will generate it yourself.The same applies to the assignment operator.In C ++ 11, the same

applies to the MOVE versions.

Also, it is worth noting that with different values of the template arguments, we obtain different types, it means that My_class <T> does not have access to private data of the MY_CLASS <U> class (and to protected, if not the heir).Therefore, to appeal to private data, you must declare these classes with friends (# 1).If you remove this announcement, we get the error of the "M_X IS Private" view.

This example also can be seen that the template class may have template member functions.I would like to note that the template class can have virtual functions, but virtual functions cannot be template.Arguments of non-type template.

The template arguments may not only be types.Consider a small example

#include <iostream> template <size_t n> struct My_type {void foo () const {std :: cout << n << std :: endl;}};int Main () {My_Type <5> O1;MY_TYPE <7> O2;O2.foo ();o1.foo ();}

After execution, we obtain the value 7 and 5. These parameters are set at the compilation stage and is not entirely obvious to what it may be necessary.Soon we will look at the use of such arguments, look at the STD :: BITSET class template, in which the non-type argument sets the size of the set:

STD :: BITSET <16> BS;

As you know, with different arguments of the template, the compiler will create different types.That is, my_class <int> and my_class <double> are different types.Also with non-type parameters, i.e.My_type <5> and My_Type <7> are also different types.This property will also be needed in the future.But at the time we distracted from this.

To begin with, we will define what can be used as a non-type argument.To do this, refer to the standard.

14.1 / 4.

A NON-TYPE TEMPLATE-PARAMETER SHALL HAVE ONE OF THE FOLLOWING (Optionally CV-Qualified) Types:

INTEGRAL OR Enumeration Type

Pointer To Object or Pointer To Function,

LVALUE REFERENCE TO OBJECT OR LVALUE REFERENCE TO FUNCTION,

pointer to member

STD :: NULLPTR_T.

Wow, what choice.But alas, there are other limitations.

14.3.2 / 1.

A Template-Argument for a non-type, Non-Template Template-Parameter Shall Be One of:

An Integral Constant Expression (Including a Constant Expression of Literal Class Type That Can Be Used AS An Integral Constant Expression As Described In 5.19);Or.

The Name Of A Non-Type Template-Parameter;Or.

a constant expression (5.19) that designates the address of an object with static storage duration and external or internal linkage or a function with external or internal linkage, including function templates and function template-ids but excluding non-static class members, expressed (ignoringParentheses) as

ID-Expression, Except That The

May Be Omitted If The Name Refers to a Function or Array and Shall Be Omitted If The Corresponder IS a reference - ParameterOr.

A Constant Expression That Evaluates to A NULL POINTER VALUE (4.10);Or.

A Constant Expression That Evaluates to A NULL Member Pointer Value (4.11);Or.

A Pointer to Member Expressed AS Described in 5.3.1.

It's not all, but it is enough to start it.Let's try to refer to the object to the template parameter:

#InClude <iostream> class My_class {int m_x;Public: My_Class (int x): M_x (x) {} int getx () const {return M_X;}};// The template argument is the link to the object type My_Class Template <my_class

MC> Struct My_Type {void foo () const {// Use the STD :: Cout << MC.GETX () << STD :: ENDL to the template argument}};My_class OBJ1 (7);My_class OBJ2 (-5);int Main () {My_Type <OBJ1> O1;// O1 is now "connected" with OBJ1 MY_TYPE <OBJ2> O2 object;// O2 is now "connected" with OBJ2 object. Calling the corresponding functions o1.foo ();O2.foo ();}

To start this information, we have enough.Template template parameters.

Briefly consider the transfer of the template as the template argument.Create a function that takes a binary predicate as an argument and references to two variables.If the predicate returns True, we make swap transmitted variables:

#include <iostream> #include <Functional> #include <algorithm> template <TypeName Binpred, TypeName T> Void Foo (T

OBJ1, T.

OBJ2, BINPRED PRED) {IF (PRED (OBJ1, OBJ2)) STD :: SWAP (OBJ1, OBJ2);} int Main () {int x = 10;int y = 30;// Foo <STD :: LESS> (X, Y);Foo (X, Y, STD :: LESS <INT> ());STD :: COUT << X << " << Y << STD :: ENDL;}

Now let's try to transfer the std :: less predicate as the template parameter.To do this, you need to change the FOO function.Template template arguments (Template Template Arguments) are defined as

Template <template-parameter-list> class ..._ optifier_opt template <template-parameter-list> class IDENTFIER_OPT = ID-Expression

In this case, the use of the Class keyword is fundamentally and TypeName will not replace it.

Template <TEMPLATE <TYPENAME> Class Binpred, TypeName T> Void Foo (T

OBJ1, T.

OBJ2) {if (binpred <t> () (OBJ1, OBJ2)) STD :: SWAP (OBJ1, OBJ2);}

TEMPLATE <TYPENAME> Class Binpred - Here it is, our template argument.As you can see, the template argument has one argument, its name does not matter, so there is no.Binpred <T> () (OBJ1, OBJ2) - Here we create an object of class binpred <t> () and call the Operator () function with two parameters.

Now rewrite the Main function:

INT MAIN () {int x = 10;int y = 30;Foo <STD :: LESS> (X, Y);// For STD :: LESS No need to specify the type of parameter, becauseWe pass the STD :: COUT << X << " << Y << STD :: ENDL;}

So we made another tiny chamber in learning templates.