Function and Class Template

By Dan Saks

Function Template:

It is not a function, it is a tool for cranking out functions

A single declaration that can generate declarations for similar but distinct functions when you know what type of data is passed.

Each generated function implements the algorithm for operands of different types.

Function Template Declaration:

Template <type T>

Void Swap (t &a, T &b) //Enough info for the compiler to generate calls to swap function

This is a template that holds a swap function with a to-be-determined type T.

Function Template Definition:

For the compiler to manufactionre the function called by the template.

Template<typename T>

Void swap (T &a, t &b){

T temp {a};

a=b;

B=temp;

}

Template Vs Function Parameter:

In the case of a function template there are two parameters lists:

A. Template parameter list

B. Function parameter list

Template <type T> //Template parameter

Void Swap (t &a, T &b){ //Function parameter

//code..

}

Template Parameter: Typename T

where T is the placeholder for a type to be determined

Argument substitution of the type takes place at compile time

The scope of T begins with type name T and ends with }

Function Parameter: Void swap (T &a, t &b)

Function parameters are the same as arguments passed in a normal function

What changes compile-time computation to run-time computation?

We can no longer pass arguments at run time because of constexpr and const, the pass of arguments has turned at compile time

Template Instantiation (part of translation process):

When c++ compilers automatically generate function definitions from templates as needed is called template instantiation

You use the template and the compiler automatically figures out what code has to be generated to support this template in the running program

Template instantiation results in instantiated functions (function definition generated from a function template).

A compiler does not instantiate duplicate function definitions. If the same instantiation is called it triggers an instantiation that has already been created earlier.

Swap<string>(s,t); //triggers instatiation

Swap<string>(u,v); //uses previous instatiation

Swap<int>(a,b); //triggers different instatiation

Swap now has two instances one for string and the other one for integers.

Multiple Type Parameters:

A template can have more than one template parameter

Template <typename U, typename T>

U find (U first, U last, T const &v);

Search through the start of U to the end of U looking for the value v where U is an iterator type

Instantiation:

i = find <double const \*, double> (b, e, x);

Template <typename U, typename T>

Typename VS Class

Class means the same as tpyename in c++ template

Temolate<class T> // same as template<tyoename T>

Void swap(T &a, T &b);

Note: class and typename are only interchangeable in templates.

Class templates:

It is a generalization of an object type class template is not a real class but an algorithm for making class

Syntax:

Template<typename T>

Class rational {

Public:

Rantional ();

Rantional (T n);

Ratiantional (T n, T d);

-----------------------------

Other member function

------------------------------

Private:

Void reduce ();

T num, den;

};

The compile will automatically instantiate class definition as needed.

Class definition generated from a class template is an instantiation of a class.

Changing the type:

Rational<long> r1 {10};

Defining “irat” as an alias for the type rational<int> using anyone of thr processes:

i. Typedef rational<int> irat;

or

ii. Using irat = rational<int>

irat ri {4,5}; //changes the type from long to int

Container Class Templates:

A container class is an object that holds other objects example arrays, linked lists, etc.

STL provides various container class templates, including:

i. List<T> - a double linked list of T objects

ii. Vector<T> - a variable-length array of T objects

iii. Set<T> - an ordered set of T objects

T is a type parameter representing the type in the containers

Note: Template isntatiation can be passed as template argument: list <rational<int>> ratios;

Member functions:

1. Template <typename T>

2. Class rational {

3. Public:

4. Rational ( );

5. Rational(T n); //declartion

6. //Other code

7. };

Note: template initiated at line 1, its scope ends at line 8.

Defination:

Template <typename T>

Inline Rational <T> :: rational (T n): num {n}, den {1} { } //first rational is class name and second is member name

Inside the scope of the class definition, you can use the word rational or rational<T>. Optional only inside the scope.

In a loose sense, a member function of a class is a template

Both of them are equivalent

Where<T> is ignored:

Rational ( );

Rational (T n);

Rational (T n, T d);

OR

Where <T> is present:

Rational <T>( );

Rational <T>(T n);

Rational <T>(T n, T d);

Outside the scope:

Member function definition outside the scope is preceded by the class name and scope resolution operator

Template<typename T>

Rational <T>

Rational <T> : : operator += (rational const &ro){ }

<T> mandatory outside the class template scope

: : reenter the scope of class

If you add <T> to the second rational after brackets that means <T> is explicitly specified.

Constructors and Destructors:

When writing constructors and destructors, you are not allowed to put<T>.

Constructor:

Template <typename T>

Rational <T>: : rational (T n): //Rational <T>: : rational <T> (T n): this syntax not allowed

Num {n}, den {1} { }

Destructor:

Template <typename T>

Rational <T> : : ~ rational ( ) { } //Rational <T> : : ~ rational <T> ( ) { } this syntax is not allowed

Note: write C<T> as just C everywhere inside the scope of a class template <T> and after the scope resolution operator does not put <T>.

To avoid any syntax error or any other complexity you can just put the destructors and constructors inside

Public: //contructor: implicitly inline

Rational (T n): num {n}, den {1} { }

Inline functions:

Keyword inline when used with a function, the compiler will place the code that functions in place of the function call. Hence the compiler makes the declaration, the definition.

Inline increases the size of executable code.

The best practice is to not use it with a very large code of a function.

Statice Data Member in C++:

Shared by all objects

Initlaized by zero, if no other initialization is given.

We cannot put it in the class definition, hence initialized outside using scope resolution operator.

Note: NO MATTER HOW MANY OBJECTS OF THE CLASS ATE CREATED, THERE IS ONLY ONE OF THE STATIC MEMBER.

Syntax:

Class Myclass{

Public:

Int x;

Static int count; //declared inside a class

My class ( ) {cout ++; } //contrustor

};

Int Myclass : : count = 0; //deifnation of count outside the class

Int main ( ) {

Code;

}

Static Member function in C++:

1. Static member function is declared by the word “static”

2. They are independent of any particular object of the class. Hence can even be called in no object is created

3. Accessed by using scope resolution operator (: :)

4. Cannot access “this” pointer of the class as it has a class scope

5. A static member function can access only:

a) Static data members

b) Other static mener funcitons

c) Any function outside the class

Syntax:

Static int getcount ( )

{ return count; }

Int main ( )

{ cout<<Myclass::getcount( ); }

Static Data Member in Templates:

Static data memeber can be declared and defined in two different ways

Method 1: Prior to C++ 17

Template <type name T>

Class gadget {

Static unsigned long counter; //declaration

}

Template <typename T>

Usigned long gadget <T> counter = 0; //defination

Method 2: C++ 17 feature

Template<typename T>

Class gadget{

Inline static unsigned long counter = 0; //declaration and deifnation

};

Static constant data memeber (a feature of C++ 11):

Template<typename T>

Class gadget {

Static int const threshold = 37;

};

Class template type members:

A class template can have members that are types:

Method 1: Outside the class

Template<typename T>

Class list {

Public:

Class iterator; //type member declaration. A nested member called iterator for containers like vectors and lists.

//rest of the code

};

Template<typename T> //memeber type defination

Class list <T>:: iterator {

//code

};

Method 2: Inside the class

Template<typename T>

Class list {

Public:

Class iterator { //member type defination};

};

Use your class iterator outside the scope of the list class by its, “fully-qualified name”.

For(list<string>: : iterator i=ls.begin( ); i!=ls.end( ); ++i)

{//code body}

Template Arguments As Expressions:

A template argument can be an expression rather than a type

Bitset<32> b1; // a sequence of 32 bits

STL provides a class template that represents a set of N bits. Bitset represents a fixed-size sequence of bits

To instantiate bitset class you don’t have to give a data type instead you give in a specific number of bits

Header and source file:

Non-Template Functions:

- Declaration in header files

- Defination in soucre files

Template Functions:

Typically place all template declarations, including definitions in headers

Nothing is kept in source files

Downside: the template function will then be included in multiple separate compiled source files and the instantiation of the same thing multiple times, separately compiled source files or units.

C++ linker can solve this problem as they are smart enough to see whats going on and essentially they throw away all but one copy of the instantiation

Angular bracket omission:

C++ often lets you omit the angle-bracketed template argument list a call to a function template

You can use just the template name as the function name

Example:

i = abs (j); //correct i = abs <int>(j); //wrong

In this case, the compile looks in the parenthesis and uses that to deduce what's missing in the angular brackets. The process of deducing the template argument from the function call argument is known as template argument deduction.

i = abs (j); //calls i = abs <int>(j)

i = abs (j); is what you are allowed to type but i = abs <int>(j) is what is instantiated

Type Deduction Failure:

Template <typename T>

Void swap(T &a, T &b);

--------------

Int i, j;

String s, t;

---------------

Swap(I,s); //error: can’t deduce template argument

The swap requires two function arguments of the same type, where we have pass arguments of two different types.

The compiler deduces the type argument by matching the form of the parameter with the form of the argument:

T const \*a [ ] ---> char const \*name [ ]

Where T is char type as the rest of the signature is an exact match of types.

Note: Type deduction only happens on function arguments, not on return types.

T f( ); // T doesnot give return type hence fails to run

Int i = f ( ); //can’t deduce type

Int i = f<int>( ); // work if stated explicitly

Example of type deduction:

Rational <int> r1, r2;

Swap(r1, r2);

Rational<int> r3 (r2); // before c++17 this was how we told the compiler r3 is an integer and r3 = r2

Rational r4 (r1); // after c++ 17, this statment states intializeation of r4 with r1 and the compiler deduces the type of r4 from r1.

Two-Phase Translation:

The compiler is limited in what it can do when it first encounters a template definition. It cannot generate code for an instantiation, as it does not know what T is yet.

Phase One:

The compile scoops up the template and stores it in the symbol table, for later use.

The compiler states that it will get back to the code late once it needs to instantiate something

This process happens just once for each template

Phase Two:

The second phase occurs when the compile instantiates the template for a particular combination of template arguments. This happens at each instantiation (every new combination of arguments being passed to the template will trigger this second phase).

When it inserts type or non-type arguments to instantiate it needs to actually instantiate something and fully compile it to semantic analysis.

Dependent Vs Non-dependent Names:

Dependent Names:

A name appearing in a template whose meaning depends on one or more template parameters is a dependent name.

Template <typename T>

T : : size\_Type munge (T const &a){

T : : size\_type \*I (T : : npos);

}

Hence size\_type and npos both are dependent names as they depend on T type parameter.

A dependent name may have a different meaning in each instantiation, the meaning of the name won’t be known till we know the meaning of T.

Non-dependent:

A non-dependent name has the same meaning in every instantiation of the template

They have the same meaning in every template they are not dependent on the type of parameter

At phase 1 the compiler does not care if it's not a type or a template name.

Types distinguish declaration from expression.

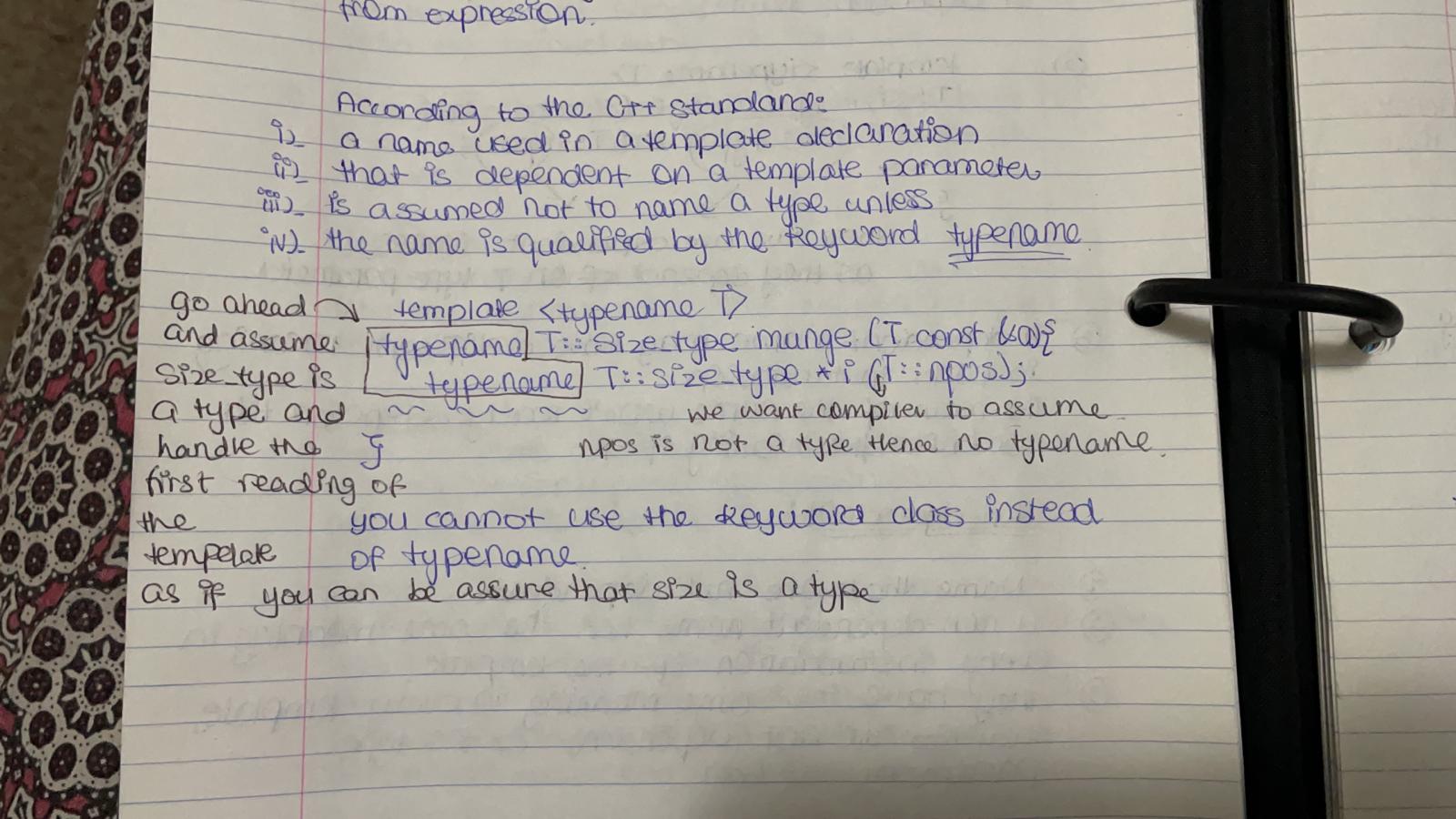
According to the C++ standard:

1. A name used in a template declaration

2. That is dependent on a template parameter

3. Is assumed not to name a type unless

4. The name is qualified by the keyword “typename”



Template Specialization:

When you take a template and a combination of type arguments and pair them together that's a specialization. The template ID turns to specialization when you plug in a type.

Template ID: template\_name <argument\_list>

Template Specialization: swap <int>

This is specilaization of the swap template.

Template Instantiation:

Normally C++ compilers automatically generate function definitions from templates as needed. The act of generating a function definition or class definition from a template is called template instantiation

NOTE: EVERY INSTATION ARISES FROM A SPECIALIZATION, NOT EVERY SPECIALIZATION PRODUCES AN INSTANTIAITION.

Explicit Vs Implicit Instantiation:

Implicit Instantiaon:

By default, a temple instantiation is an implicit instantiation

The compiler determines what code it must generate to satisfy the use of a template, and generate that code automatically.

Explicit Instantiation:

Explicit instantiation is when you specify exactly where the compiler should instantiate a particular specialization.

A program must contain exactly one definition for every template instantiation in the program

If you explicitly instantiate a temple in one translation unit, you must ensure that the specialization is not instantiated elsewhere

Example

Before C++ 11:

Template void swap <string> (string &a, string &b);

Or

Template void swap (int &a, int &b);

After C++ 11:

Extern opt template declaration

Extern template voud swap <string> (string &a, string &b);

Explicit Specialization:

C++ lets you alter the behuour of a particular template specialization

Sytnax:

Tempalate< > //declaration

This takes the general template and tailors it to work for a particular type of argument. This ensures that a general rule is not applied. It ensures that our template looks like the same interface but it has different behavior.

Example:

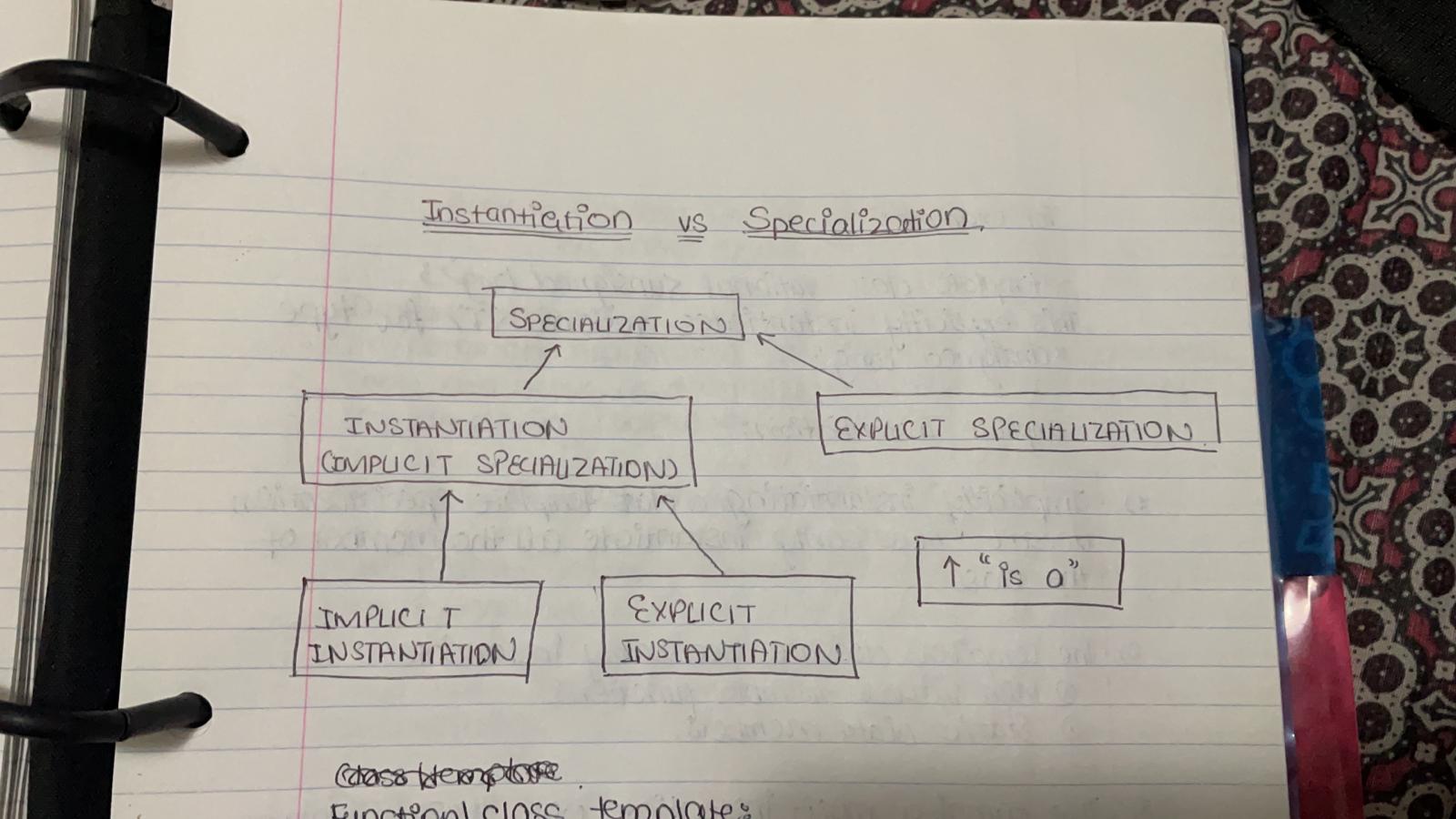
Template < >

Char const \*

Max <char const \*> (char const \*a, char const \* b) //This declaration is valid only if max is already declared as the name of a function template

{return strcmp (a,b)>0?a:b;}

You can overload a function to simply get the explicit specialization. During overload resolution, the function will complete with the function template as the best match for calls to max. For calls with operands of type char \* or char const \*, the overloaded function will emerge as the best match.



Function/ Class Template:

1. Specialization

2. Explicit Specialization

3. Implicit Instantiation

4. Explicit Instantiation

Class Template instantiation:

By default the compiler instantiates class template implicitly

However, you can explicitly instantiate a class template using a declaration of the form:

Extern opt template declaration

Example:

Template class rational<unsigned long>;

This explicitly instantiates rational <T> for type “unsigned long”

Implicit Instantiation:

Implicitly instantiating a class template specialization doesn’t necessarily instantiate all the members of the class

The compiler does not necessarily instantiate:

a) Non-virtual member functions

b) Static data members

The compiler must instantiate only those members that the program access

However explicitly instantiating a class template specialization, instantiates all the members of that specialization

Syntax:

Template rationa<int> : : ratuonal (int n);