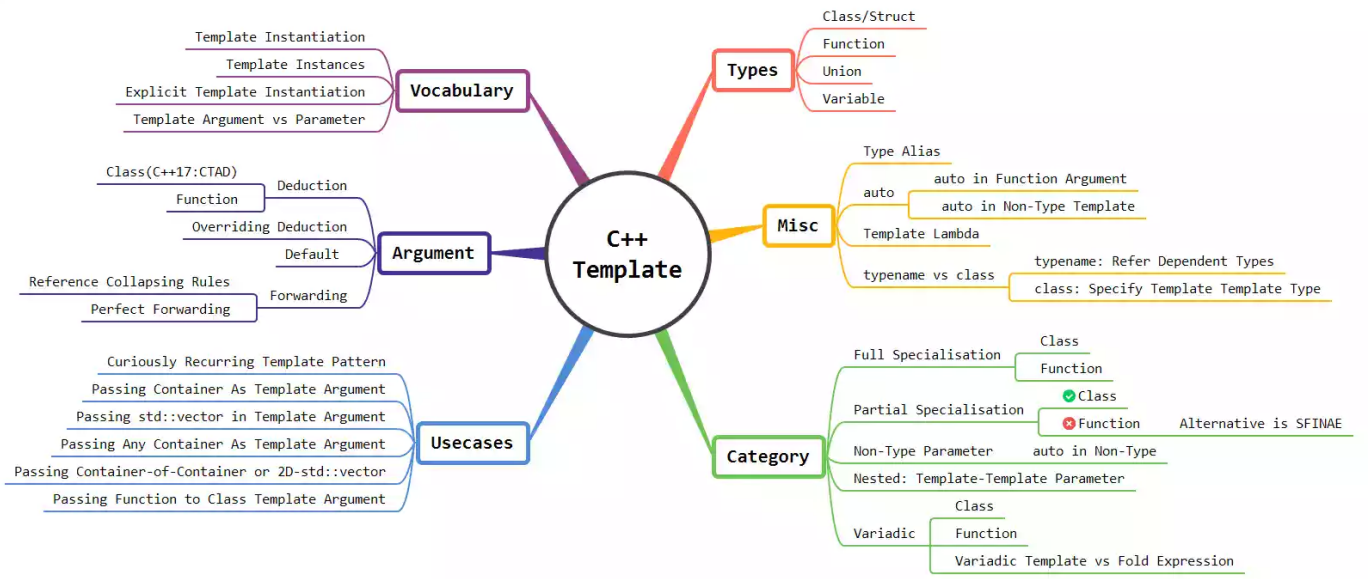
C++ Template: A Quick UpToDate Look(C++11/14/17/20)



Reading Time: 18 minutes

I know, it's been a while since the last time I published something newbies-friendly on my blog. The main reason is that most of my readers are either experienced devs or from C background having modest C++ encounter. But while programming in C++ you need a completely different mindset as both C & C++ belongs to different programming paradigm. And I always strive to show them a better way of doing things in C++. Anyway, I found the topic which is lengthy, reasonably complex(at least it was for me), newbies-friendly as well as energizing for experienced folks(if Modern C++ jargons, rules & features added) i.e. C++ Template.

I will start with a simple class/function template and as we move along, will increase the complexity. And also cover the advance topics like the variadic template, nested template, CRTP, template vs fold-expression, etc. But, yes! we would not take deeper dive otherwise this would become a book rather than an article.

Note: *I would recommend you to use cppinsights online tool wherever you feel confused. It helps you to see Template Instances, Template Argument Deduction, etc. Basically, it helps you to see code from the compiler's perspective.*

Terminology/Jargon/Idiom You May Face

- **Template Instantiation**: It is a process of generating a concrete class/struct/union/function out of templated class/struct/union/function for a particular combination of template arguments. For example, if you use vector<int> & vector<char>, it will create two different concrete classes during compilation. This process of creating concrete classes is known as Template Instantiation.
- **Template Instances**: Outcome of Template Instantiation is Template Instances i.e. concrete classes.
- **Explicit Template Instantiation**: Usually template instantiation done at the time of object declaration. But you can also force the compiler to instantiate class/struct/union/function with particular type without even creating the object. It may appear in the program anywhere after the template definition, and for a given argument-list. Will see this later in the article.
- **Template Argument vs Template Parameter**: In expression template<typename *T*> void print(*T a*) { };, T is parameter & when you call print(5);, 5 which is of type int is template argument. This is a trivial thing for some pips. But not for non-native English speaker or beginners. So, this ambiguity has to be clear.

C++ Template Types

Class Template

```
template <typename T1, typename T2>
class pair {
public:
    T1  first;
    T2  second;
};

pair<int, char> p1;
pair<float, float> p2;
```

- The basic idea of a class template is that the template parameter i.e. T1 & T2 gets substituted by an appropriate deduced type at compile time. The result is that the same class can be reused for multiple types.
- And the user has to specify which type they want to use when an object of the class is declared.

Function Template

```
template <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}

min<int>(4, 5);              // Case 1
min<float>(4.1f, 5.1f);      // Case 2
```

- In both of the above case, the template arguments used to replace the types of the parameters i.e. T.
- One additional property of template functions (unlike class template till C++17) is that the compiler can infer the template parameters based on the parameters passed to the function. So, passing <int> & <float> after the function name is redundant.

Union Template

- Yes! a union can also be templatized. In fact, the standard library provides some utilities like std::optional, std::variant, etc. which directly or indirectly uses templatized union.

```
template <typename T>
union test {
    uint8_t     ch[sizeof(T)];
    T           variable;
};
```

- As you can see above, ***templatized unions are also particularly useful to represent a type simultaneously as a byte array***.

Variable Template

- Yes! This may a bit socking. But, ***you can templatise the variable also since C++14***.

```
template <class T>
constexpr T pi = T(3.1415926535897932385L); // variable template

cout << pi<float> << endl; // 3.14159
cout << pi<int> << endl;   // 3
```

- Now, you might be wondering that what is the point of the templatizing variable. But, consider the following example:

```
template <uint32_t val>
constexpr auto fib = fib<val - 1> + fib<val - 2>;

template <>
constexpr auto fib<0> = 0;

template <>
```

```
constexpr auto fib<1> = 1;

cout << fib<10> << endl;    // 55
```

- Above code gives you 10th Fibonacci term at compile time, without even creating class or function.

C++ Template Argument

Overriding Template Argument Deduction

```
template <typename T>
T min(T a, T b) {
    cout << typeid(T).name() << endl; // T will be deduce as `int`
    return a < b ? a : b;
}

min<int>(5.5f, 6.6f);     // Implicit conversion happens here
```

Default Template Arguments

```
template <class T, size_t N = 10>
struct array {
    T arr[N];
};

array<int> arr;
```

- Just like in case of the function arguments, template parameters can also have their default values.
- All template parameters with a default value have to be declared at the end of the template parameter list.

Template Argument Deduction

Function Template Argument Deduction

- Function template argument deduction is done by comparing the types of function arguments to function parameters, according to rules in the Standard. Which makes function templates far more usable than they would otherwise be. For example, given a function template like:

```
template <typename RanIt>
void sort(RanIt first, RanIt last){
    // . . .
}
```

- You can and should sort a std::vector<int> without explicitly specifying that RanIt is std::vector<int>::iterator. When the compiler sees sort(v.begin(), v.end());, it knows what the types of v.begin() and v.end() are, so it can determine what RanIt should be.

Class Template Argument Deduction(CTAD)

- Until C++17, template classes could not apply type deduction in their initialization as template function do. For example

```
//...
pair p4{1, 'A'};              // Not OK until C++17: Can't deduce type in initialization
//...
```

- But ***from C++17, the compiler can deduce types in class/struct initialization*** & this to work, class/struct must have an appropriate constructor. But this limitation is also relaxed in C++20. So technically ***from C++20, you can construct the object with aggregate initialization & without specifying types explicitly***.
- Until C++17, the standard provided some std::make_ utility functions to counter such situations as below.

Inferring Template Argument Through Function Template

- You might have seen many functions like std::make_pair(), std::make_unique(), std::make_share(), etc. Which can typically & unsophistically implement as:

```
template <typename T1, typename T2>
pair<T1, T2> make_pair(T1&& t1, T2&& t2) {
    return {forward<T1>(t1), forward<T2>(t2)};
}
```

- But have you ever wonder why these helper functions are there in the standard library? How does this even help?

```
pair<int, char> p1{1, 'A'};          // Rather using this

auto p2 = make_pair(1, 2);           // Use this instead
auto p3 = make_pair<float>(1, 2.4f); // Or specify types explicitly
```

- Rather than specifying the arguments explicitly, you can leverage the feature of inferring template argument from function template to construct the object. In the above case, template argument deduction is done by the utility function make_pair. As a result, we have created the object of pair without specifying the type explicitly.
- And as discussed earlier from C++17, you can construct the object without even specifying types explicitly so std::vector v{1,2,3,4}; is perfectly valid statement.

Template Argument Forwarding

C++ Template Reference Collapsing Rules

- Apart from accepting type & value in the template parameter. You can enable the template to accept both lvalue and rvalue references. And to do this you need to adhere to the rules of reference collapsing as follows:

  1. T& & becomes T&
  2. T& && become T&
  3. T&& & becomes T&
  4. T&& && becomes T&&

```
template <typename T>
void f(T &&t);
```

- In the above case, the real type of t depends on the context. For example:

```
int x = 0;

f(0); // deduces as rvalue reference i.e. f(int&&)
f(x); // deduces as lvalue reference i.e. f(int&)
```

- In case of f(0);, 0 is rvalue of type int, hence T = int&&, thus f(int&& &&t) becomes f(int&& t).
- In case of f(x);, x is lvalue of type int, hence T = int&, thus f(int& &&t) becomes f(int& t).

Perfect Forwarding | Forwarding Reference | Universal Reference

- In order to perfectly forward t to another function, one must use std::forward as:

```
template <typename T>
void func1(T &&t) {
    func2(std::forward<T>(t));  // Forward appropriate lvalue or rvalue reference to another function
}
```

- Forwarding references can also be used with variadic templates:

```
template <typename... Args>
void func1(Args&&... args) {
    func2(std::forward<Args>(args)...);
}
```

Why Do We Need Forwarding Reference in First Place?

- Answer to this question lies in move semantics. Though, short answer to this question is "To perform copy/move depending upon value category type".

C++ Template Category

Full Template Specialization

- Template has a facility to define implementation for specific instantiations of a template class/struct/union/function/method.

Function Template Specialization

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }

template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

- In the above case, a user that writes sqrt(4.0) will get the generic implementation whereas sqrt(4) will get the specialized implementation.

Class Template Specialization

```
template <typename T>       // Common case
struct Vector {
    void print() {}
};

template <>                 // Special case
struct Vector<bool> {
    void print_bool() {}
};

Vector<int> v1;
v1.print_bool();    // Not OK: Chose common case Vector<T>
v1.print()          // OK

Vector<bool> v2;    // OK : Chose special case Vector<bool>
```

Partial Template Specialization

Partial Class Template Specialization

- In contrast of a full template specialization, you can also specialise template partially with some of the arguments of existing template fixed. Partial template specialization is only available for template class/structs/union:

```
template <typename T1, typename T2>     // Common case
struct Pair {
    T1 first;
    T2 second;

    void print_first() {}
};

template <typename T>    // Partial specialization on first argument as int
struct Pair<int, T> {
    void print() {}
};

// Use case 1 ----------------------------------------------------------
Pair<char, float> p1;    // Chose common case
p1.print_first();        // OK
// p1.print();           // Not OK: p1 is common case & it doesn't have print() method
```

```
// Use case 2 ------------------------------------------------------
Pair<int, float> p2;     // Chose special case
p2.print();              // OK
// p2.print_first();     // Not OK: p2 is special case & it does not have print_first()

// Use case 3 ------------------------------------------------------
// Pair<int> p3;         // Not OK: Number of argument should be same as Primary template
```

Partial Function Template Specialization

- ***You cannot partially specialize method/function***. Function templates may only be fully specialized

```
template <typename T, typename U>
void foo(T t, U u) {
    cout << "Common case" << endl;
}

// OK.
template <>
void foo<int, int>(int a1, int a2) {
    cout << "Fully specialized case" << endl;
}

// Compilation error: partial function specialization is not allowed.
template <typename U>
void foo<string, U>(string t, U u) {
    cout << "Partial specialized case" << endl;
}

foo(1, 2.1); // Common case
foo(1, 2);   // Fully specialized case
```

Alternative To Partial Function Template Specialization

- As I have mentioned earlier, partial specialization of function templates is not allowed. You can use SFINAE with std::enable_if for work around as follows:

```
template <typename T, typename std::enable_if_t<!std::is_pointer<T>::value> * = nullptr>
void func(T val) {
    cout << "Value" << endl;
}

template <typename T, typename std::enable_if_t<std::is_pointer<T>::value> * = nullptr>
void func(T val) {  // NOTE: function signature is NOT-MODIFIED
    cout << "Pointer" << endl;
}

int a = 0;
func(a);
func(&a);
```

Non-Type Template Parameter

- As the name suggests, apart from types, you can also declare the template parameter as constant expressions like addresses, references, integrals, std::nullptr_t, enums, etc.
- Like all other template parameters, non-type template parameters can be explicitly specified, defaulted, or derived implicitly via Template Argument Deduction.
- The ***more specific use case of a non-type template is passing a plain array into a function without specifying its size explicitly***. A more relevant example of this is std::begin & std::end specialisation for array literal from the standard library:

```
template <  class T,
            size_t size>     // Non Type Template
```

```
T* begin(T (&arr)[size]) {   // Array size deduced implicitly
    return arr;
}

int arr[] = {1,2,3,4};
begin(arr);                  // Do not have to pass size explicitly
```

- Non-type template parameters are one of the ways to achieve template recurrence & enables Template Meta-programming.

Nested Template: Template Template Parameter

- Sometimes we have to pass templated type into another templated type. And in such case, you not only have to take care of main template type but also a nested template type. Very simple template- template parameter examples is:

```
template<
            template <typename> class C,
            typename T
        >
void print_container(C<T> &c) {
    // . . .
}

template <typename T>
class My_Type {
    // . . .
};

My_Type<int> t;
print_container(t);
```

Variadic Template

- It is often useful to define class/struct/union/function that accepts a variable number and type of arguments.
- If you have already used C you'll know that printf function can accept any number of arguments. Such functions are entirely implemented through macros or ellipses operator. And because of that it has several disadvantages like type-safety, cannot accept references as arguments, etc.

Variadic Class Template

Implementing Unsophisticated Tuple Class(>=C++14)

- Since C++11 standard library introduced std::tuple class that accept variable data members at compile time using the variadic template. And to understand its working, we will build our own ADT same as std::tuple
- The variadic template usually starts with the general (empty) definition, that also serves as the base-case for recursion termination in the later specialisation:

```
template <typename... T>
struct Tuple { };
```

- This already allows us to define an empty structure i.e. Tuple<> object;, albeit that isn't very useful yet. Next comes the recursive case specialisation:

```
template<
            typename T,
            typename... Rest
        >
struct Tuple<T, Rest...> {
    T               first;
    Tuple<Rest...>  rest;
```

```
    Tuple(const T& f, const Rest& ... r)
        : first(f)
        , rest(r...) {
    }
};

Tuple<bool> t1(false);                      // Case 1
Tuple<int, char, string> t2(1, 'a', "ABC"); // Case 2
```

How Does Variadic Class Template Works?

To understand variadic class template, consider use case 2 above i.e. Tuple<int, char, string> t2(1, 'a', "ABC");

- The declaration first matches against the specialization, yielding a structure with int first; and Tuple<char, string> rest; data members.
- The rest definition again matches with specialization, yielding a structure with char first; and Tuple<string> rest; data members.
- The rest definition again matches this specialization, creating its own string first; and Tuple<> rest; members.
- Finally, this last rest matches against the base-case definition, producing an empty structure.

You can visualize this as follows:

```
Tuple<int, char, string>
-> int first
-> Tuple<char, string> rest
    -> char first
    -> Tuple<string> rest
        -> string first
        -> Tuple<> rest
            -> (empty)
```

I have written a separate article on Variadic Template C++: Implementing Unsophisticated Tuple, if you are interested more in the variadic temple.

Variadic Function Template

- As we have seen earlier, variadic template starts with empty definition i.e. base case for recursion.

```
void print() {}
```

- Then the recursive case specialisation:

```
template<
            typename First,
            typename... Rest                    // Template parameter pack
        >
void print(First first, Rest... rest) {         // Function parameter pack
    cout << first << endl;
    print(rest...);                             // Parameter pack expansion
}
```

- This is now sufficient for us to use the print function with variable number and type of arguments. For example:

```
print(500, 'a', "ABC");
```

- You can further optimize the print function with forwarding reference, if constexpr() & sizeof() operator as:

```
template<
            typename First,
            typename... Rest
        >
void print(First&& first, Rest&&... rest) {
```

```
    if constexpr(sizeof...(rest) > 0) {             // Size of parameter pack
        cout << first << endl;
        print(std::forward<Rest>(rest)...);         // Forwarding reference
    }
    else {
        cout << first << endl;
    }
}
```

How Does Variadic Function Template Works?

- As you can see we have called print with 3 arguments i.e. print(500, 'a', "ABC");
- At the time of compilation compiler instantiate 3 different print function as follows:

    1. void print(int first, char __rest1, const char* __rest2)
    2. void print(char first, const char* __rest1)
    3. void print(const char* first)

- The first print(i.e. accept 3 arguments) will be called which prints the first argument & line print(rest…); expand with second print(i.e. accept 2 arguments). This will go on till argument count reaches to zero.
- That means in each call to print, the number of arguments is reduced by one & the rest of the arguments will be handled by a subsequent instance of print.
- Thus, the number of print instance after compilation is equal to the number of arguments, plus the base case instance of print. Hence, the ***variadic template also contributes to more code bloating***.
- You can get this much better if you put the above example in cppinsights. And try to understand all the template instances.

Fold Expressions vs Variadic Template

- As we saw, from C++11, the variadic template is a great addition to C++ Template. But it has nuisance like you need base case & recursive template implementation, etc.
- So, with ***C++17 standard introduced a new feature named as Fold Expression***. Which you can use with parameter pack as follows:

```
template <typename... Args>
void print(Args &&... args) {
    (void(cout << std::forward<Args>(args) << endl), ...);
}
```

- See, no cryptic boilerplate required. Isn’t this solution looks neater?
- There are total 3 types of folding: Unary fold, Binary fold & Fold over a comma. Here we have done left folding over a comma. You can read more about Fold Expression here.

Misc

C++ Template `typename` vs `class`

- typename and class are interchangeable in most of the cases.
- A general convention is typename used with the concrete type(i.e. in turn, does not depend on further template parameter) while class used with dependent type.
- But there are cases where either typename or class has to be certain. For example

To Refer Dependent Types

```
template<typename container>
class Example {
    using t1 = typename container::value_type; // value_type depends on template argument of container
    using t2 = std::vector<int>::value_type;   // value_type is concrete type, so doesn't require typename
};
```

- typename is a must while referencing a nested type that depends on template parameter.

To Specify Template Template Type

```
template<
            template <typename, typename> class C, // `class` is must prior to C++17
            typename T,
            typename Allocator
        >
void print_container(C<T, Allocator> container) {
    for (const T& v : container)
        cout << v << endl;
}

vector<int> v;
print_container(v);
```

- This is rectified in C++17, So now you can use typename also.

C++11: Template Type Alias

```
template<typename T>
using pointer = T*;

pointer<int> p = new int;   // Equivalent to: int* p = new int;


template <typename T>
using v = vector<T>;

v<int> dynamic_arr;         // Equivalent to: vector<int> dynamic_arr;
```

- typedef will also work fine, but would not encourage you to use. As it isn't part of Modern C++.

C++14/17: Template & auto Keyword

- ***Since C++14, you can use auto in function argument***. It's kind of template shorthand as follows:

```
void print(auto &c) { /*. . .*/ }

// Equivalent to

template <typename T>
void print(T &c) { /*. . .*/ }
```

- Although ***auto in function return-type is supported from C++11***. But, you have to mention the trailing return type. Which is rectified in C++14 & now return type is automatically deduced by compiler.
- ***From C++17, you can also use auto in non-type template***(I will cover this in later part this article) parameters.

C++20: Template Lambda Expression

- A generic lambda expression is supported since C++14 which declare parameters as auto. But there was no way to change this template parameter and use real template arguments. For example:

```
template <typename T>
void f(std::vector<T>&  vec) {
        //. . .
}
```

- How do you write the lambda for the above function which takes std::vector of type T? This was the limitation till C++17, but ***with C++20 it is possible templatized lambda*** as :

```
auto f = []<typename T>(std::vector<T>&  vec) {
    // . . .
};
```

```
std::vector<int> v;
f(v);
```

Explicit Template Instantiation

- An explicit instantiation creates and declares a concrete class/struct/union/function/variable from a template, without using it just yet.
- Generally, you have to implement the template in header files only. You can not put the implementation/definition of template methods in implementation files(i.e. cpp or .cc). If this seems new to you, then consider following minimalist example:

value.hpp
```
#pragma once

template <typename T>
class value {
    T val;
public:
    T get_value();
};
```

value.cpp
```
#include "value.hpp"

template <typename T>
T value<T>::get_value() {
    return val;
}
```

main.cpp
```
#include "value.hpp"

int main() {
    value<int> v1{9};
    cout << v1.get_value() << endl;
    return 0;
}
```

- If you compile above code you will get following error:

```
/tmp/main-4b4bef.o: In function `main':
main.cpp:(.text+0x1e): undefined reference to `value<int>::get_value()'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
compiler exit status 1
```

- If you do explicit initialization i.e. add template class value<int>; line at the end of value.cpp. Then the compilation gets successful.
- The “template class” command causes the compiler to explicitly instantiate the template class. In the above case, the compiler will stencil out value<int> inside of value.cpp.
- There are other solutions as well. Check out this StackOverflow link.

C++ Template Example Use Cases

Curiously Recurring Template Pattern

- CRTP widely employed for static polymorphism or code reusability without bearing the cost of virtual dispatch mechanism. Consider the following code:

```
template <typename specific_animal>
struct animal {
    void who() { implementation().who(); }

private:
    specific_animal &implementation() { return *static_cast<specific_animal *>(this); }
};
```

```
struct dog : animal<dog> {
    void who() { cout << "dog" << endl; }
};

struct cat : animal<cat> {
    void who() { cout << "cat" << endl; }
};


template <typename specific_animal>
void who_am_i(animal<specific_animal> *animal) {
    animal->who();
}


who_am_i(new dog); // Prints `dog`
who_am_i(new cat); // Prints `cat`
```

- We have not used virtual keyword & still achieved the functionality of polymorphism(more-specifically static polymorphism).
- I have written a separate article covering practical Examples of Curiously Recurring Template Pattern(CRTP).

Passing `std` Container as C++ Template Argument

- If you wanted to accept anything and figure it out later, you could write:

```
template <typename C>
void print_container(const C &container) {
    for (const auto &v : container)
        cout << v << endl;
}
```

- This naive way may fail if you pass anything other than standard container as other types may not have begin & end iterator.

Passing std::vector to C++ Template Function

Naive Way to Capture Container's Value Type

- But let say, you want to pass container & want to work with container's storage type also. You can do:

```
template<
            typename C,
            typename T = typename C::value_type
        >
void print_container(const C &container) {
    for (const T &v : container)
        cout << v << endl;
}
```

- We can provide the second type parameter to our function that uses SFINAE to verify that the thing is actually a container.
- All standard containers have a member type named value_type which is the type of the thing inside the container. We sniff for that type, and if no such type exists, then SFINAE kicks in, and that overload is removed from consideration.

Capturing Container's Value Type Explicitly

- But what if you are passing vector class which doesn't has value_type member?
- std::vector is defined as:

```
template<
            class T,
            class Allocator = std::allocator<T>
```

```
        >
class vector;
```

- And you can capture two template arguments of std::vector container explicitly as:

```
template<
            template <typename, typename> class C,
            typename T,
            typename Allocator
        >
void print_container(C<T, Allocator> container) {
    for (const T& v : container)
        cout << v << endl;
}
```

- Above template pattern would be same if you want pass container to class/struct/union.

Passing Any Container to C++ Template Function

- You see if you pass any other containers to the above solution. It won't work. So to make it generic we can use variadic template:

```
template<
            template <typename...> class C,
            typename... Args
        >
void print_container(C<Args...> container) {
    for (const auto &v : container)
        cout << v << endl;
}

vector<int>     v{1, 2, 3, 4}; // takes total 2 template type argument
print_container(v);

set<int>        s{1, 2, 3, 4}; // takes total 3 template type argument
print_container(s);
```

Passing Container-of-Container/2D-std::vector as C++ Template Argument

- This is the case of nested template i.e. template-template parameter. And there are the following solutions:

Explicit & Complex Solution

```
template<
            template <typename, typename> class C1,
            template <typename, typename> class C2,
            typename Alloc_C1, typename Alloc_C2,
            typename T
        >
void print_container(const C1<C2<T, Alloc_C2>, Alloc_C1> &container) {
    for (const C2<T, Alloc_C2> &container_in : container)
        for (const T &v : container_in)
            cout << v << endl;
}
```

- I know this is ugly, but seems more explicit.

Neat Solution

```
template<
            typename T1,
            typename T2 = typename T1::value_type,
            typename T3 = typename T2::value_type
        >
void print_container(const T1 &container) {
    for (const T2 &e : container)
```

```
        for (const T3 &x : e)
            cout << x << endl;
}
```

- As seen earlier including SFINAE.

Generic Solution: Using Variadic Template

```
template<
            template <typename...> class C,
            typename... Args
        >
void print_container(C<Args...> container) {
    for (const auto &container_2nd : container)
        for (const auto &v : container_2nd)
            cout << v << endl;
}
```

- This is our standard solution using the variadic template will work for a single container or any number of the nested container.

Passing Function to Class Template Argument

- Passing class/struct/union to another class/struct/union as template argument is common thing. But passing function to class/struct/union as template argument is bit rare. But yes it's possible indeed. Consider the Functional Decorator using a variadic class template.

```
// Need partial specialization for this to work
template <typename T>
struct Logger;

// Return type and argument list
template <typename R, typename... Args>
struct Logger<R(Args...)> {
    function<R(Args...)>    m_func;
    string                  m_name;
    Logger(function<R(Args...)> f, const string &n) : m_func{f}, m_name{n} { }

    R operator()(Args... args) {
        cout << "Entering " << m_name << endl;
        R result = m_func(args...);
        cout << "Exiting " << m_name << endl;
        return result;
    }
};

template <typename R, typename... Args>
auto make_logger(R (*func)(Args...), const string &name) {
    return Logger<R(Args...)>(function<R(Args...)>(func), name);
}

double add(double a, double b) { return a + b; }

int main() {
    auto logged_add = make_logger(add, "Add");
    auto result = logged_add(2, 3);
    return EXIT_SUCCESS;
}
```

- Above example may seem a bit complex to you at first sight. But if you have a clear understanding of variadic class temple then it won't take more than 30 seconds to understand what's going on here.

Conclusion

I hope I have covered most of the topics around C++ Template. And yes, this was a very long & intense article. But I bet you that if you do master the C++ template well, it will really give you an edge. And also open a door to sub-world of C++ i.e. template meta-programming.

C++ Template: C++'s its own interpreted sub language

– IndianWestCoast

Do you like it👍? Get such articles directly into the inbox…!?