Build Throughput Series: Template Metaprogramming Fundamentals

Xiang Fan

January 14th, 2021 | 5 | 0

Template metaprogramming is popular and seen in many code bases. However, it often contributes to long compile times. When investigating build throughput improvement opportunities in large codebases, our finding is that more than one million template specializations and template instantiations is quite common and often provides optimization opportunities for significant improvement.

In this blog post, I will walk through the differences between template specialization and template instantiation and how they are processed in the MSVC compiler. I will cover how to find these bottlenecks related to too many template specializations and instantiations in a different blog post (or you can read this blog post as a starting point).

Before we start, let us clarify some terms widely used in template metaprogramming.

- Primary template
  - Partial specialization
- Template specialization
  - Explicit specialization
- Template instantiation
  - Implicit template instantiation
  - Explicit template instantiation

They are better explained by an example:

```
// Primary template.
template<typename T> struct Trait {};
// Partial specialization.
template<typename T> struct Trait<T*> {};
// Explicit specialization.
template<> struct Trait<int> {};

// Implicit template instantiation of template specialization
'Trait<void>' from the primary template.
Trait<void> trait1;
// Implicit template instantiation of template specialization
'Trait<void*>' from the partial specialization.
Trait<void*> trait2;
// No template instantiation for explicit specialization.
Trait<int> trait3;
// Explicit template instantiation of template specialization
'Trait<char>' from the primary template.
template struct Trait<char>;
// Explicit template instantiation of template specialization
'Trait<char*>' from the partial specialization.
template struct Trait<char*>;
```

Template specialization and template instantiation are often used interchangeably. However, the distinction is important when evaluating build throughput.

Let us look at an example:

```
template<typename T> struct Vector
{
    void sort() { /**/ }
    void clear() { /**/ }
};

Vector<int> get_vector();

template<typename V> void sort_vector(V& v) { v.sort(); }

void test(Vector<long>& v)
{
    ::sort_vector(v); // I will explain why we use '::' here
later.
}
```

In the example above, the MSVC compiler does the following:

```
Start processing user code
    Process class template 'Vector'
    Process function 'get_vector'
        Specialize 'Vector<int>'
    Process function template 'sort_vector'
    Process function 'test'
        Specialize 'Vector<long>'
        Specialize 'sort_vector<Vector<long>>'
        Instantiate 'sort_vector<Vector<long>>' (delayed)
            Add 'sort_vector<Vector<long>>' to the pending
list
End processing user code
Start processing the pending list for delayed instantiation
    Iteration 1
        Instantiate 'sort_vector<Vector<long>>'
        Instantiate 'Vector<long>'
        Instantiate 'Vector<long>::sort' (delayed)
            Add 'Vector<long>::sort' to the pending list
    Iteration 2
        Instantiate 'Vector<long>::sort'
End processing the pending list
```

You can see that template specialization occurs at an earlier step in processing than template instantiation and is often cheaper.

When you specialize a function template (like `sort_vector<Vector<long>>` in the example), the compiler only processes its declaration and its definition is not processed. The compiler will create an internal representation for the specialization and add that to a map. If the same specialization is specialized again later, the compiler will find the internal representation from the map and reuse it to avoid duplicated work (known as *memoization*). The definition is processed when the specialization is instantiated.

Similarly, when you specialize a class template its definition is also not processed. Instantiation of class template specialization is a bit more complicated. By default, the member of the class template specialization is not instantiated when the specialization itself is instantiated (like `Vector<long>::clear`). The member is instantiated when it is used (like `Vector<long>::sort`) and MSVC will delay the instantiation if possible.

You may wonder what if I use `sort_vector` in `test`. It will change the processing order.

- When qualified name `::sort_vector` is used, it suppresses argument dependent lookup (ADL).
- When unqualified name `sort_vector` is used instead, ADL will compute the associated set of `v` and this forces the instantiation of `Vector<long>`. So, the instantiation is no longer delayed to the phase which processes the pending list.

With this information in mind, let us check some common patterns and see which requires template instantiation.

```
template<int N> struct Array { static_assert(N > 0, ""); };

struct Data
{
    Array<1> arr; // Array<1> is instantiated.
};

Array<2> transform(Array<3> *); // Neither Array<2> nor
Array<3> is instantiated.

void test()
{
    transform(nullptr); // Array<2> is instantiated, Array<3>
is not instantiated.
}
```

The `Array<1>` case: When it is used as the type of a member, the compiler needs to instantiate the specialization to know its information like the size. This is one of the most common reasons why a template specialization is instantiated in a header and is often hard to avoid.

The `Array<2>` case: Using a template specialization as the function return type does not require it to be instantiated (if there is no function definition). The same is true if it is used as the type of a function parameter. However, providing the function definition or calling the function will force the instantiation of the return type.

The `Array<3>` case: passing `nullptr` as the function argument does not require the instantiation because `nullptr` is always convertible to any pointer type. The same is true if you cast `nullptr` to `Array<3> *`. However, if the function argument is a pointer to a class, the compiler must instantiate `Array<3>` to see whether the conversion is valid.

In the next blog post, we will use some examples from the real-world code bases and find ways to reduce the number of template specializations and template instantiations.

Xiang Fan PRINCIPAL SOFTWARE ENGINEER, Visual C++ Compiler Front-End
Follow

Posted in C++

Read next

C++ with Visual Studio and WSL2

Our team released native support for C++ with the Windows Subsystem for Linux (WSL) in Visual Studio in

Build Throughput Series: More Efficient Template Metaprogramming

In the previous blog post I shared how template specialization and template instantiation are processed in