Build Throughput Series: More Efficient Template Metaprogramming

Xiang Fan

January 19th, 2021 | 0 | 0

In the previous blog post I shared how template specialization and template instantiation are processed in the MSVC compiler. We will now look at some examples from real-world code bases to show some ways to reduce the number of them.

Example 1

This example is extracted from our own MSVC compiler code base. The code tries to apply several stateless functors on an object. Because the functors are stateless, they are represented by a list of types. Here is the code:

```
// A helper class which represents a list of types.
template<typename...> struct TypeList;

// The definition of 'Object' is irrelevant and omitted.
struct Object;
// The function which applies a stateless functor. Its
definition is irrelevant and omitted.
template <typename Functor> void apply_functor(Object&
object);

// We have two functors.
struct Functor1;
struct Functor2;

// We want to apply the two functors above.
void apply(Object& object)
{
    using Functors = TypeList<Functor1, Functor2>;
    apply_all_functors<Functors>(object); //
'apply_all_functors' is not implemented yet.
}
```

Now let us see the initial implementation of `apply_all_functors`. We extract the functors from `TypeList` and apply them one by one:

```
#include <utility>

template <typename Functors>
struct apply_all_functors_impl {
    template <size_t I>
    static void apply(Object& object) {
        using Functor = TypeListAt<I, Functors>; //
'TypeListAt' is not implemented yet.

        apply_functor<Functor>(object);
    }

    template <size_t... I>
    static void apply_all(Object& object,
std::index_sequence<I...>) {
        (apply<I>(object), ...);
    }

    void operator()(Object& object) const
    {
        apply_all(object,
std::make_index_sequence<TypeListSize<Functors>>{}); //
'TypeListSize' is not implemented yet.
    }
};

template <typename Functors>
constexpr apply_all_functors_impl<Functors>
apply_all_functors{};
```

To extract the functor from the list, we need a sequence of indices. This is obtained using `std::make_index_sequence`. We then use a fold expression to efficiently iterate through the sequence and call `apply` to extract and apply the functor one by one.

The code above uses a class template so that the template arguments are shared across all its member functions. You can also use global function templates instead.

There are several ways to implement `TypeListAt` and `TypeListSize`. Here is one solution:

```
// Implementation of TypeListSize.
template<typename> struct TypeListSizeImpl;
template<typename... Types> struct
TypeListSizeImpl<TypeList<Types...>>
{
    static constexpr size_t value = sizeof...(Types);
};
template<typename Types> constexpr size_t TypeListSize =
TypeListSizeImpl<Types>::value;

// Implementation of TypeListAt.
template<size_t, typename> struct TypeListAtImpl;
template<size_t I, typename Type, typename... Types> struct
TypeListAtImpl<I, TypeList<Type, Types...>>
{
    using type = typename TypeListAtImpl<I - 1,
TypeList<Types...>>::type;
};
template<typename Type, typename... Types> struct
TypeListAtImpl<0, TypeList<Type, Types...>>
{
    using type = Type;
};

template<size_t I, typename Types> using TypeListAt = typename
TypeListAtImpl<I, Types>::type;
```

Now let us examine the number of template instantiations in the initial implementation (assume we have `N` functors):

1. We iterate through an integer sequence of `N` elements (with value `0, ..., N - 1`).
2. Each iteration specializes one `TypeListAt` which instantiates `O(I) TypeListAtImpl` specializations (`I` is the element in the integer sequence).

For example, when `TypeListAt<2, TypeList<T1, T2, T3>>` (I = 2, N = 3) is used, it goes through the following:

```
TypeListAt<2, TypeList<T1, T2, T3>> =>
TypeListAtImpl<2, TypeList<T1, T2, T3>>::type =>
TypeListAtImpl<1, TypeList<T2, T3>>::type =>
TypeListAtImpl<0, TypeList<T3>>::type =>
T3
```

So, `apply_all_functors_impl<TypeList<T1, ..., TN>>::operator()` instantiates `O(N^2)` template specializations.

How can we reduce the number? The core logic is to extract types from the helper class `TypeList`.

To reduce the number of template instantiations, we can extract directly without using `std::integer_sequence`. This takes advantage of function template argument deduction which can deduce the template arguments of a class template specialization used as the type of the function parameter.

Here is the more efficient version:

```
// Function template argument deduction can deduce the
functors from the helper class.
template <typename... Functors>
void apply_all_functors_impl (Object& object,
TypeList<Functors...>*)
{
    ((apply_functor<Functors>(object)), ...);
}

template <typename Functors>
void apply_all_functors (Object& object)
{
    apply_all_functors_impl(object, static_cast<Functors*>
(nullptr));
}
```

Now it only instantiates `O(N)` template specializations.

Note: I intentionally leave `TypeList` as undefined. The definition is not even needed for the `static_cast` as I mentioned in the previous blog post. This can avoid all the overheads associated with defining a class (like declaring lots of compiler generated special member functions, generating debug information, etc.) which can happen accidentally (see the next example for more details).

We apply this trick in the compiler code base and it cuts the memory usage to compile one expensive file by half. We also see noticeable compile time improvement.

Example 2

This example is extracted from the code base of an internal game studio. To my surprise, game developers love template metaprogramming 😊.

The code tries to obtain a list of trait classes from a type map.

```
#include <tuple>
#include <utility>

// This class contains some useful information of a type.
template <typename>
class trait {};

// TypeMap is a helper template which maps an index to a type.
template <template <int> class TypeMap, int N>
struct get_type_traits;

template<int> struct type_map;
template<> struct type_map<0> { using type = int; };
template<> struct type_map<1> { using type = float; };

// we want to get back 'std::tuple<trait<int>, trait<float>>'.
using type_traits = get_type_traits<type_map, 2>::type; //
'get_type_traits' is not implemented yet.
```

Here is the initial implementation:

```
template <template <int> class TypeMap, int N>
struct get_type_traits
{
private:
    template <int... I>
    static auto impl(std::integer_sequence<int, I...>)
    {
        return std::make_tuple(trait<typename
TypeMap<I>::type>{}...);
    }
public:
    using type = decltype(impl(std::make_integer_sequence<int,
N>{}));
};
```

It also uses the same `make_integer_sequence` trick in example 1.

`get_type_traits` itself doesn't have the `O(N^2)` specializations issue. But unfortunately, the current `std::tuple` implementation in MSVC has O(n^2) behavior to instantiate where `n` is the number of its template arguments.

This overhead can be completely avoided because the class only needs to get back a type which does not necessarily require instantiation.

However, the initial implementation forces the instantiation of `std::tuple` due to the definition of `impl`. As mentioned in the previous blog post, having a template specialization as the return type does not require instantiation if there is no function definition.

The solution is to specify the return type of `impl` explicitly and remove the definition. This trick is not always possible when the return type is complicated. But in this case, we can specify it as:

```
template <int... I>
static std::tuple<trait<typename TypeMap<I>::type>...>
impl(std::integer_sequence<int, I...>);
```

This change reduces the compile time by 0.9s where an `std::tuple` of 85 template arguments is used. We have seen such `std::tuple` (with lots of template arguments) usages in quite a few code bases.

Summary

Here is a list of simple tips which can help reduce the number and overhead of template specialization/instantiation:

1. Avoid instantiating a non-linear number of template specializations. Be aware of type traits which require a non-trivial number of specializations (e.g., those using recursion).
2. Leave class template as undefined if possible (e.g., help class which carries all the information in its template arguments).

3. Prefer variable templates to class templates for values (`variable_template<T>` is much cheaper than `class_template<T>::value` and `class_template<T>()` is the worst 😊)
4. Be aware of expensive template (like `std::tuple` with lots of template arguments) and switch to a simpler type if you use the template for a different purpose than what it is designed for (e.g., using `std::tuple` as a type list).

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

Posted in C++

Read next

MSVC Backend Updates in Visual Studio 2019 version 16.9 Preview 3
In Visual Studio 2019 version 16.9 Preview 3 we have continued to improve the C++ backend with new features, new and improved optimizations, build throughput ...
Helena Gregg
January 21, 2021
2 comments

Windows ARM64 support for CMake projects in Visual Studio
In Visual Studio 2019 version 16.9 Preview 3 we added support for deploying CMake projects to a remote Windows machine and debugging them with the Visual Studio remote ...
Erika Sweet
January 21, 2021
0 comment

0 comments

Comments are closed. Login to edit/delete your existing comments