Template Metaprogramming

Group 8

26.12.2022

GROUP 8

Members:

- 21127115 - Tran Thanh Ngan
- 21127118 - Lam Thanh Ngoc
- 21127173 - Nguyen Thien Tho
- 21127734 - Huynh Si Kha

Table of Contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Definition of Template Metaprogramming

- "Programming a program"
- "Compile-time"

Definition

- Used by compiler to generate temporary source code, which is merged by the compiler and then compiled.
- The output can include compile-time constants, data structures, and complete functions.
- The use can be thought of as compiler-time polymorphism.
- Used by a number of languages, the best-known being C++.

The reason why Template MP was chosen?

- Be used to move computations from run-time to compile-time
- Generate code using compile time computations
- Enable self-modifying code
- Time-saving for a future call

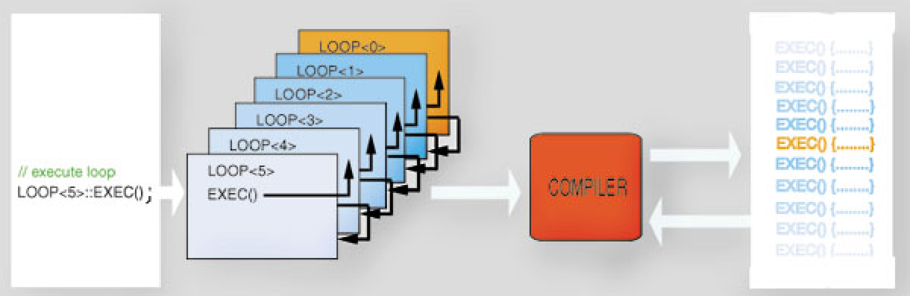


Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

When TMP should be used?

- A macro is not enough details.
- Using recursive function with predetermined number of loops for significant reduction in run time by avoiding calling the overhead of function as well as setting up stack variables.
- Constant value need calculating.

When TMP should not be used?

- A macro is enough details and small code size which more friendly than TMP
- Program has already had a long-time compilation

How to make TMP more practical for programmers?

- Some facts of TMP:
  - Like other const expressions, values of enumeration constants are evaluated at compile time.
  - When accessing to new argument of a template, compiler creates a new instance of the template.
- Naming appropriately:
  - Not naming at all, which means "don't look at this, it's irrelevant and it's there only for technical reasons".
  - Naming meaningfully to express the function of parameter.
- Stratify abstraction:
  - Stratification prevents code duplication in case programmers need the same technique for checking other expressions at some point.

Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Advantages of TMP

- It reduces repetition of the code.
- Executed exactly once at compile time regardless of how many times it is called at runtime.
- Created base on template $->$ better maintainability

Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Disadvantages of TMP

- Syntax is quite complex
- Compilation takes long time since code is generated in compile-time
- Cannot print a string during the compilation
- Make difficult debugging
- Not supported by all compilers (even though ISO C++ Standard)

Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Preprocessing the code using a macro I

1. **One-line macro preprocessor**

```cpp
// Defining macro
#define MAX(a,b) (((a) > (b)) ? (a) : (b))

// Initializing 2 int variables
int x = 10;
int y = 20;

// Consuming the MAX macro
int z = MAX(x,y);
```

Preprocessing the code using a macro II

2. **Multiline macro preprocessor**

```
// Defining multi line macro
#define SWAP(a,b) { \
(a) ^= (b); \
(b) ^= (a); \
(a) ^= (b); \
}

// Initializing two int variables
int x = 10;
int y = 20;

// Consuming the SWAP macro
SWAP(x,y);
```

Dissecting template metaprogramming in the Standard Library

- An incomplete function contained in the C ++ Standard Library is used to generate complete functions.
- The template metaprogramming is the C++ template to generate C++ types and code in compile time.

```
template<typename T>
class Array
{
    T element;
};
```

Building the template metaprogramming I

- Four factors → template metaprogramming: *type*, *value*, *branch*, and *recursion*.

```
struct ValueDataType
{
    typedef int valueDataType;
};
```

- Similarly, a value can be stored to the variable thanks to *enum* to access the value variable for its value.

```
struct ValuePlaceHolder
{
    enum { value = 1 };
};
```

Building the template metaprogramming II

1. **Mapping a function to the input parameters**

```
template<int A, int B>
struct Multiplexer
{
    enum { result = A * B };
};
```

To access the *result* variable:

```
int i = Multiplexer<2, 3>::result;
// i = 6
```

Building the template metaprogramming III

2. **Choosing the correct process based on the condition**

```
template<typename A, typename B>
struct CheckingType
{
    enum { result = 0 };
};

```

```
template<typename X>
struct CheckingType<X, X>
{
    enum { result = 1 };
};

```

Building the template metaprogramming IV

```
if (CheckingType<UnknownType, int>::result)
{
    // run the function if the UnknownType is int
}

else
{
    // otherwise run any function
}

```

Building the template metaprogramming V

3. **Repeating the process recursively**
   The factorial value can be calculated as below:

```
template <int I>
struct Factorial
{
    enum { value = I * Factorial<I-1>::value };
};
```

```
template <>
struct Factorial<0>
{
    enum { value = 1 };
};
```

```
int fact10 = Factorial<10>::value;
// fact10 = 3628800
```

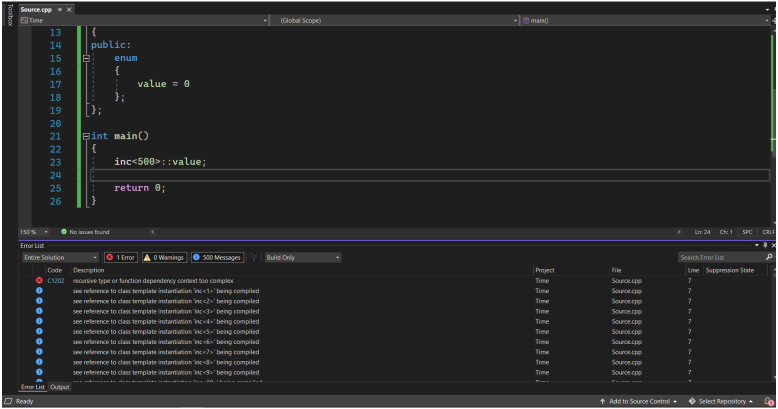
Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Demonstrate the power of metaprogramming

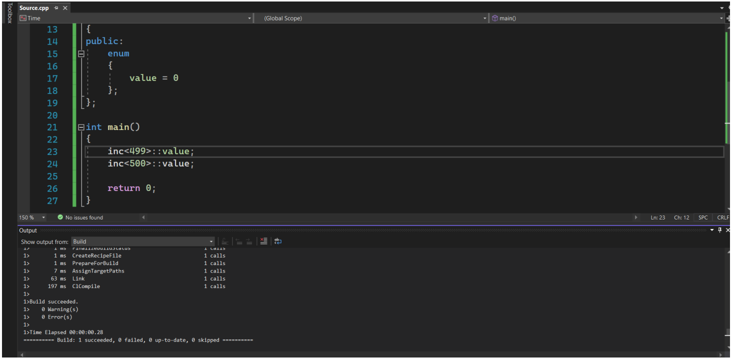
- Tasks that have been done since the last compilation would not need to be repeated or re-initialized.
- Demonstrate via *inc* $<$*int N*$>$ *function*: increase by 1 at time
- The evidence was done by comparing the compilation time and the number of recursive iterators.

First issue: The limitation of number of recursive iterators



Solution: "set a thief to catch a thief"

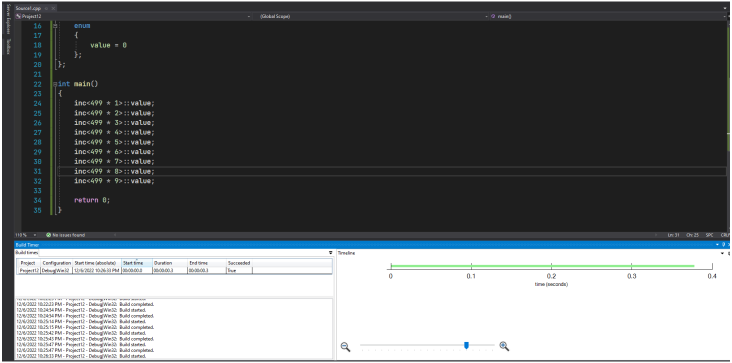
- Use smaller → larger



- Proof that the previous value will be save, no need to re-initialized.

Second: Use build timer extension to measure compilation time

First calling,



Second: Use build timer extension to measure compilation time

Second calling,

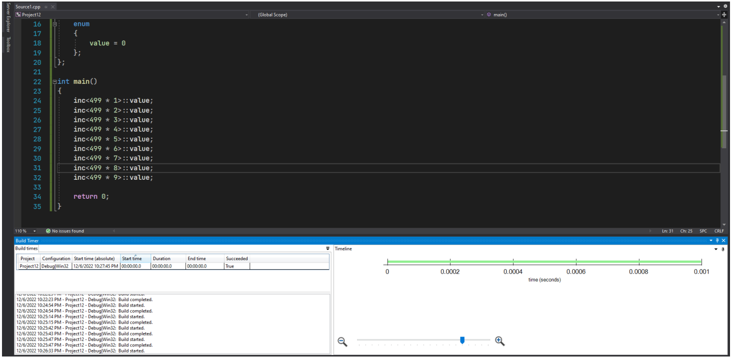


Table of contents

1. Introduction to Template Metaprogramming
2. Rules of Template Metaprogramming
3. Advantages of TMP
4. Disadvantages of TMP
5. How to write a metaprogramm with template?
6. Demonstrate the power of metaprogramming
7. Metaprogramming library

Metaprogramming library I

1. These are some typical funtions of metaprogramming library:
   - std::is_pointer
   - std::is_union
   - std::is_class
   - std::is_floating_point
   - std::is_enum
   - std::is_integral
2. Usage: Check whether the given type is the appropriate type or not.
3. Parameter: The single parameter *T (Trait class)* and check if T is the right type.
4. Return value: A boolean (true if it is the right type and reverse).

Metaprogramming library II

5. Syntax:

```
template <class T> struct is_"function name"

```

6. Return type:
   - **True:** if the type is "function name" type
   - **False:** if the type is not "function name" type

Metaprogramming library III

7. Illustration:

```
class GFG {};
int main()
{
  cout << boolalpha;
  cout << "is_pointer:"
    << endl;
  cout << "GFG: "
    << is_pointer<GFG>::value << '\n';
  cout << "GFG*: "
    << is_pointer<GFG*>::value << '\n';
  cout << "GFG&: "
    << is_pointer<GFG&>::value << '\n';
  cout << "nullptr_t: "
    << is_pointer<nullptr_t>::value << '\n';

  return 0;
}

```

Output:

```
is_pointer:
GFG: false
GFG*: true
GFG&: false
nullptr_t: false

```

Thank you

We apprepriate your attention!