UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

REPORT

OOP PROJECT: TEMPLATE METAPROGRAMMING

Trần Thanh Ngân - 21127115
Lâm Thanh Ngọc - 21127118
Nguyễn Thiên Thọ - 21127173
Huỳnh Sỉ Kha - 21127734

Lecturers:

Nguyễn Minh Huy

Nguyễn Lê Hoàng Dũng

Trần Duy Quang

26th December 2022

Contents

Name	**Student ID**	**Completion level**	**Assignment**
Trần Thanh Ngân	21127115	100%	Report, presentation
Lâm Thanh Ngọc	21127118	100%	Report, presentation
Nguyễn Thiên Thọ	21127173	100%	Algorithms, source code, demonstrate
Huỳnh Sĩ Kha	21127734	100%	Algorithms, source code, demonstrate

NOTES:

In section 7, the program uses extension *Visual Studio Build Timer* in order to calculate build time.

Click **here** for the link of the extension

1 What is Template Metaprogramming (TMP)?

Metaprogramming consists of "programming a program". This means computer programs are allowed to be computed to read, generate, analyze, transform other programs and even modify itself while running that are minimized the number of code lines for reducing development time.

Template Metaprogramming (TMP) is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. The output of these templates can include compile-time constants, data structures, and completed functions. The use of templates can be thought of as compiler-time polymorphism. The technique is used by a number of programming languages, the best-known being C++.

2 Why Template Metaprogramming?

TMP can be used to move computations from run-time to compile-time, generate code using compile time computations, and enable self-modifying code. The ability of a programming language to be its own metalanguage is called reflection. Reflection is a valuable language feature to facilitate metaprogramming. It also allows programs a greater flexibility to efficiently handle new situations without recompilation.

3 Rules of template metaprogramming:

3.1 When TMP should be used?

TMP is a great technique when used correctly. On the other hand, it might result in untidy code and the performance decreases. Below are some rules of thumb when to use TMP:

- A macro is not enough details. Then, before compiling, something that is more complex than a macro is required and asked to expand.
- Using recursive function with a 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 values need calculating. If constants that depend on other constants are used in a program, they might be a candidate for TMP.

3.2 When TMP should not be used?

TMP can't be used in these cases:

- When a macro is enough details and able to execute with a small code size as it's more friendly than TMP.
- The program that has already had a long-time-compilation since TMP often tends to increase it significantly.

3.3 How to make TMP code much more practical for programmers?

- Acknowledge some facts of TMP:

  - Like other const expressions, values of enumeration constants are evaluated at compile time.

  - When a new argument is recognized to 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 the parameter.

- Stratify abstraction:

  - Stratification prevents code duplication in case programmers need the same technique for checking other expressions at some point.

4 Advantages of TMP:

- It reduces repetition of the code.
- With respect to C++ prior to C++11, the syntax and idioms of template metaprogramming were esoteric compared to conventional C++ programming, and template metaprograms could be very difficult to understand. But from C++11 onward the syntax for value computation metaprogramming becomes more and more akin to "normal" C++, with less and less readability penalty.
- The program is executed exactly once at compile time regardless of how many times it is called at runtime.

- Template metaprogramming allows the programmer to focus on architecture and delegate to the compiler the generation of any implementation required by client code. Thus, template metaprogramming can accomplish truly generic code, facilitating code minimization and better maintainability.

5 Disadvantages of TMP:

Although the program is executed exactly once at compile time regardless of how many times it is displayed at runtime.
But:...

- The syntax is extremely complex.
- The compilation time takes longer since code is executed during compile-time.
- There is no debugger for the C++ compilation process and the length of type-names in template metaprogramming can easily reach thousands of characters due to deep template nesting, which makes debugging very difficult.
- There is no way for a template metaprogram to output a string during compilation.
- Template metaprograms may extend compilation times by orders of magnitude. Template metacode is interpreted rather than compiled.
- Complex computations quickly lead to very complex types in template metaprogramming. The complexity and size limits of different compilers vary, but the limits can be quickly reached.
- There are no variables through which information can be passed between expressions at compile time in C++.
- Template metaprogramming is based on many advanced C++ language features, which might not be supported by all compilers (even though they are in ISO C++ standard).

6 How to write a metaprogram with template?

6.1 Preprocessing the code using a macro

The C parameterized macro is also known as metafunctions and is one of the examples of metaprogramming.

1. **One-line macro preprocessor**

```
// 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);
```

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);
```

6.2 Dissecting template metaprogramming in the Standard Library

The Standard Library provided in the C++ language is mostly a template that contains an incomplete function. However, it will be 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;
};
```

6.3 Building the template metaprogramming

There are four factors that form the template metaprogramming: *type*, *value*, *branch*, and *recursion*.

a. **Adding a value to the variable in the template**

- When we are talking about the variables in template metaprogramming, it's actually not a variable since the value on it cannot be modified. What we need from the variable is its name so we can access it.
- The metafunction is a function that works with types. This means, the approach to use template metaprogramming is working with type parameters only when possible.

- For example, the *int* type is stored to the *valueDataType* alias name so we can access the data type using the *valueDataType* variable.

```cpp
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.

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

```

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

- The user parameters is retrieved and mapped to a function for adding the variable to the template metaprogramming.
- For example, to multiply 2 parameters, the template requires two arguments, A and B and they are used to get the value of result variable by being multiplied.

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

```

- To access the *result* variable:

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

```

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

- When there are more than a function, choosing which one to use depends on a base condition.
- For example, a condition branch can be constructed by providing two templates that have X and A/B as their type to check whether 2 arguments are the same type.

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

```

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

```

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

else
{
    // otherwise run any function
}

```

d. **Repeating the process recursively**

- Since the variable in the template is immutable, the sequence cannot be iterated.
- Instead, recursion are manipulated by 2 templates in which, one is used to calculate and another is used as a base case.
- For example, 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 };
};

```

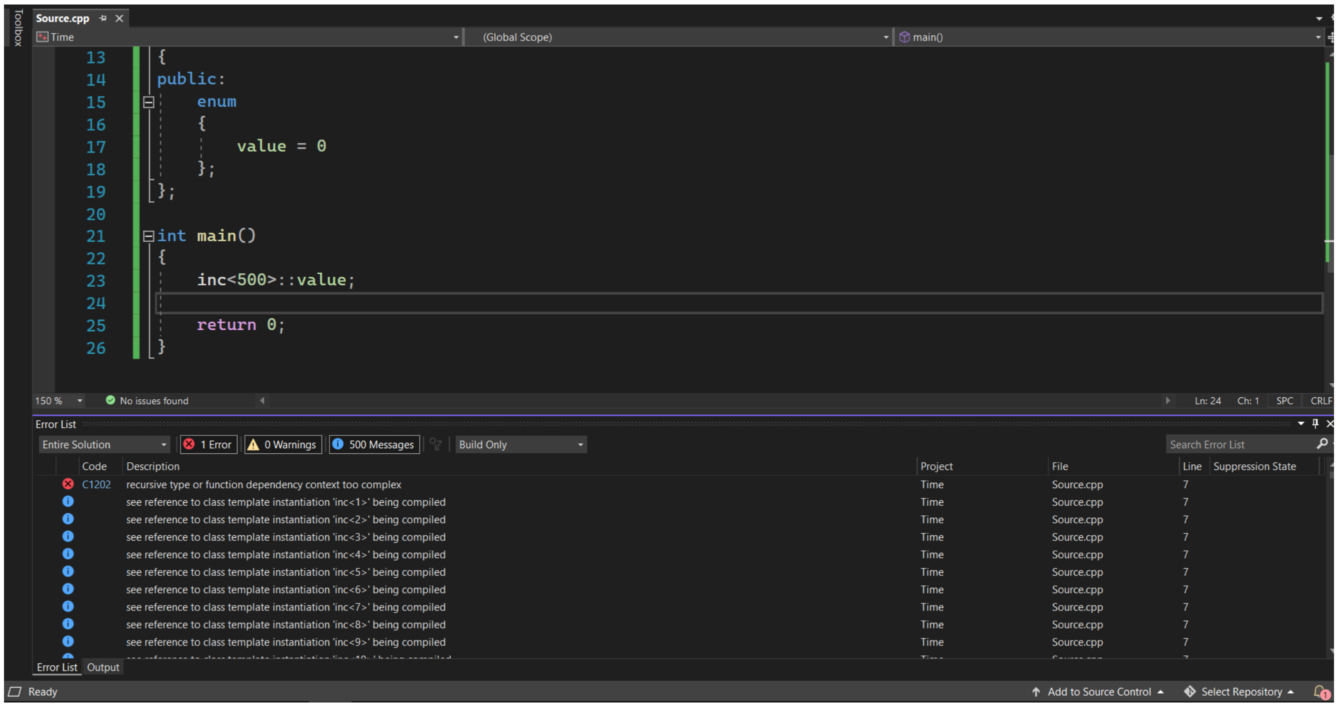
```
int fact10 = Factorial<10>::value;
// fact10 = 3628800

```

7 Demonstrate the power of metaprogramming

One of the most effective aspects of template metaprogramming is that, thanks to the creation of temporary source code in execute files, tasks that have been done since the last compilation would not need to be repeated or re-initiated. Our group has found the way to show that things:

The example is based on inc<int N> function

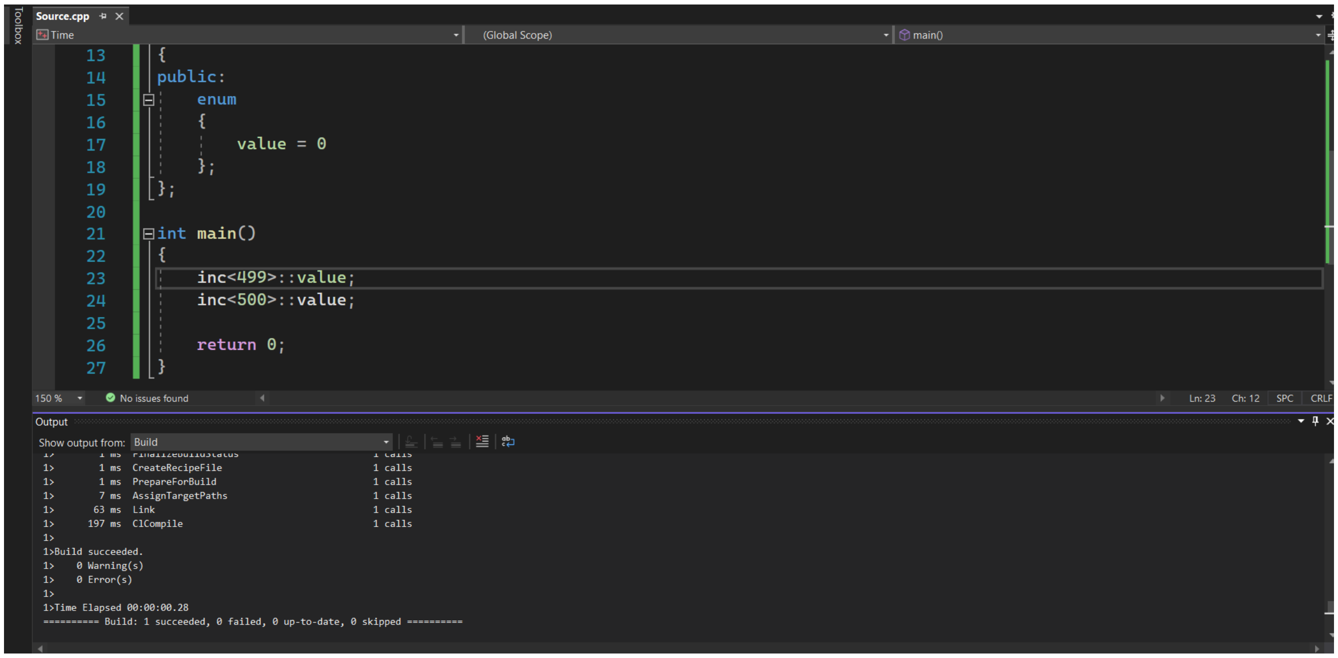
- Firstly, we will prove that the command line *inc<500>::value* sees reference to class template before it *"inc<499>::value"*



```
{
public:
    enum
    {
        value = 0
    };
};

int main()
{
    inc<500>::value;

    return 0;
}
```

```
Entire Solution   1 Error   0 Warnings   500 Messages   Build Only

    Code   Description                                                        Project  File        Line  Suppression State
    C1202  recursive type or function dependency context too complex          Time     Source.cpp  7
           see reference to class template instantiation 'inc<1>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<2>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<3>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<4>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<5>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<6>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<7>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<8>' being compiled  Time  Source.cpp  7
           see reference to class template instantiation 'inc<9>' being compiled  Time  Source.cpp  7
```

- Since we attempted to solve this issue, we came to the realization that: the number of recursive iterators is restricted by the processing capability of your machine, IDE version, etc. The following will call the previous (recursion rule), and we will build the larger value on the smaller, so the TMP also has a technique to perform this duty.



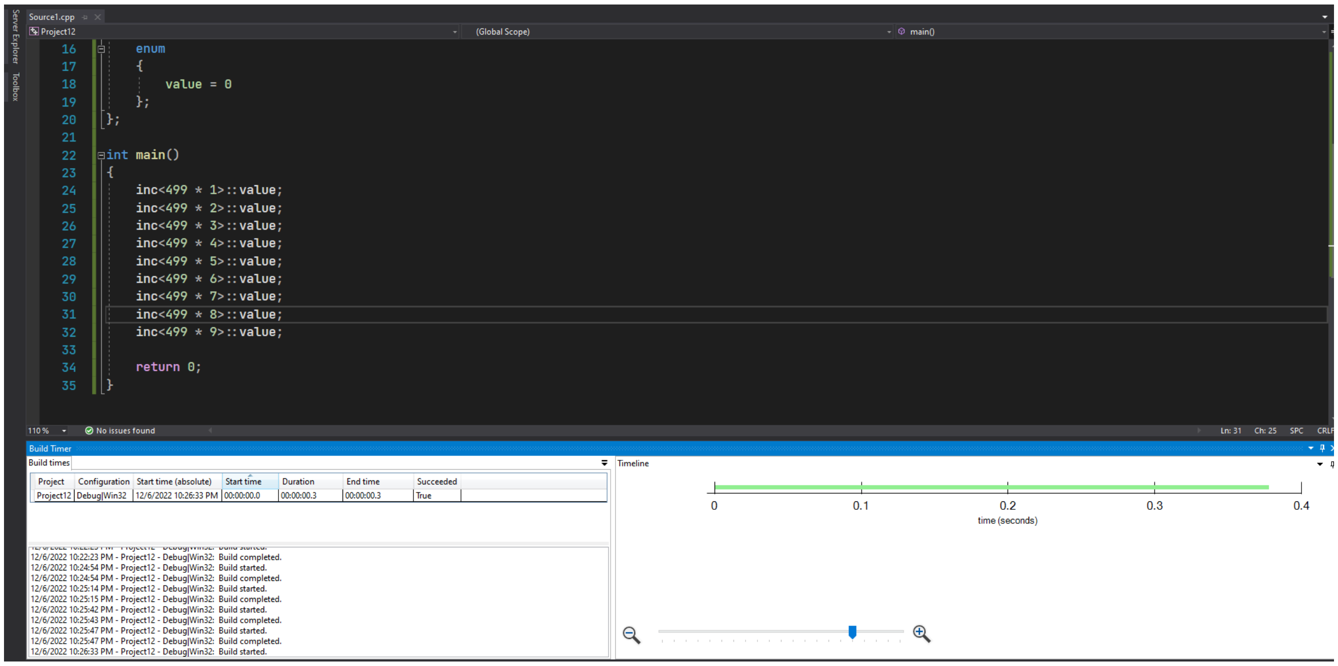
```
{
public:
    enum
    {
        value = 0
    };
};

int main()
{
    inc<499>::value;
    inc<500>::value;

    return 0;
}
```

```
1>        1 ms  CreateRecipeFile                           1 calls
1>        1 ms  PrepareForBuild                            1 calls
1>        7 ms  AssignTargetPaths                          1 calls
1>       63 ms  Link                                       1 calls
1>      197 ms  ClCompile                                  1 calls
1>
1>Build succeeded.
1>    0 Warning(s)
1>    0 Error(s)
1>
1>Time Elapsed 00:00:00.28
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

- In order to increase the efficiency of the proof, we also give a second method of displaying compile time, which indicates the time reduction of the following compile time.

- The first time compilation

The compilation time: 0.4s

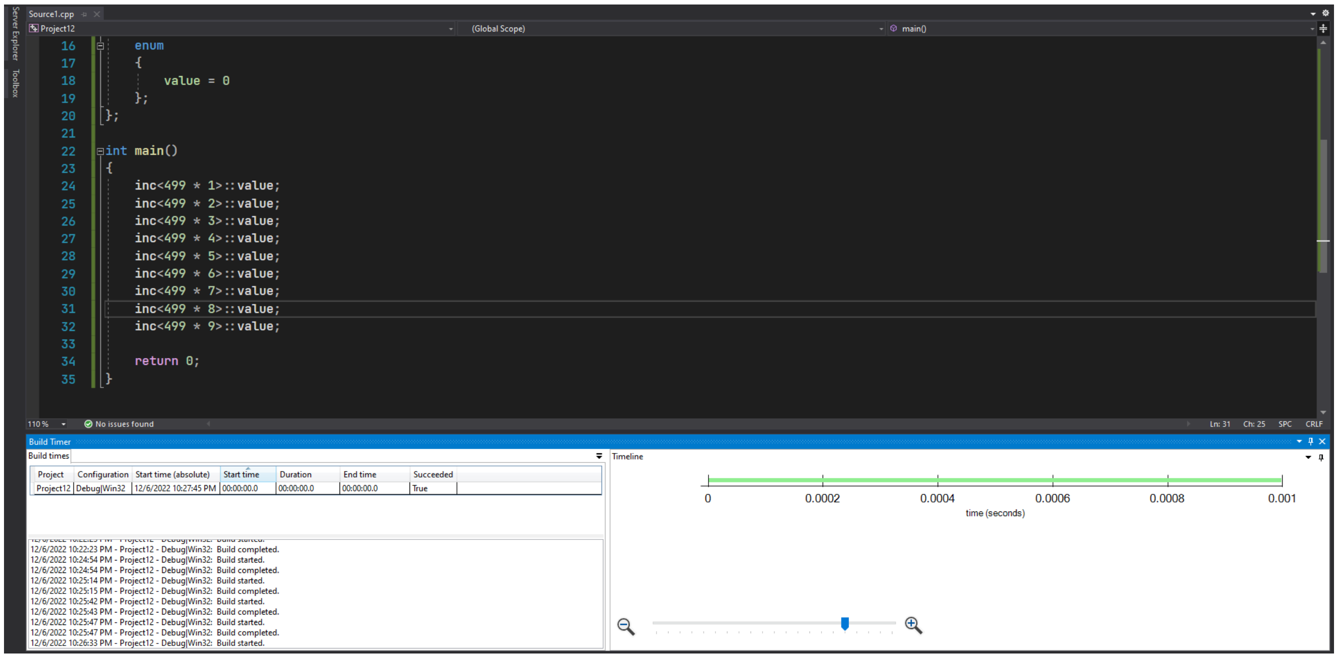


```
    enum
    {
        value = 0
    };
};

int main()
{
    inc<499 * 1>::value;
    inc<499 * 2>::value;
    inc<499 * 3>::value;
    inc<499 * 4>::value;
    inc<499 * 5>::value;
    inc<499 * 6>::value;
    inc<499 * 7>::value;
    inc<499 * 8>::value;
    inc<499 * 9>::value;

    return 0;
}
```

Project	Configuration	Start time (absolute)	Start time	Duration	End time	Succeeded
Project12	Debug|Win32	12/6/2022 10:26:33 PM	00:00:00.0	00:00:00.3	00:00:00.3	True

```
12/6/2022 10:22:23 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:24:54 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:24:54 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:25:14 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:25:15 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:25:42 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:25:43 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:25:47 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:25:47 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:26:33 PM - Project12 - Debug|Win32:  Build started.
```

- The second time, picture shows the reduction of compile time
  The compilation time: 0.001



```
    enum
    {
        value = 0
    };
};

int main()
{
    inc<499 * 1>::value;
    inc<499 * 2>::value;
    inc<499 * 3>::value;
    inc<499 * 4>::value;
    inc<499 * 5>::value;
    inc<499 * 6>::value;
    inc<499 * 7>::value;
    inc<499 * 8>::value;
    inc<499 * 9>::value;

    return 0;
}
```

Project	Configuration	Start time (absolute)	Start time	Duration	End time	Succeeded
Project12	Debug|Win32	12/6/2022 10:27:45 PM	00:00:00.0	00:00:00.0	00:00:00.0	True

```
12/6/2022 10:22:23 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:24:54 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:24:54 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:25:14 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:25:15 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:25:42 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:25:43 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:25:47 PM - Project12 - Debug|Win32:  Build started.
12/6/2022 10:25:47 PM - Project12 - Debug|Win32:  Build completed.
12/6/2022 10:26:33 PM - Project12 - Debug|Win32:  Build started.
```

→ The programming don't need to pre-compile, just use the available.

- There is other way to replace enum → typedef ifthenelse.

  – Example of enum

```
template <int N, int LO = 1, int HI = N>
class Sqrt2 {
public:
  // compute the midpoint, rounded up
  enum { mid = (LO + HI + 1) / 2 };
  // search a not too large value in a halved interval
  enum {
    result = (N < mid * mid) ? Sqrt2<N, LO, mid - 1>::result : Sqrt2<N, mid, HI>::result
  };
};

```

  – Example of ifthenelse

```
template<int N, int LO = 1, int HI = N>
class Sqrt {
public:
  // compute the midpoint, rounded up
  enum { mid = (LO + HI + 1) / 2 };
  // search a not too large value in a halved interval
  typedef typename IfThenElse<(N < mid * mid),
    Sqrt<N, LO, mid - 1>,
    Sqrt<N, mid, HI> >::ResultT
    SubT;
  enum { result = SubT::result };
};

```

– Because enum's not only instantiates the templates in the positive branch, but also those in the negative branch. When the whole process is examined in detail, we find that dozens of instantiations end up being generated. The total number is: $2 * N$

– Instead of typedef ifthenelse only instantiates the templates in the positve branch. The total number is: $log_2(N)$

8 Metaprogramming library (since C++11)

C++ provides metaprogramming facilities, such as type traits, compile-time rational arithmetic, and compile-time integer sequences.

Almost functions is used to check whether the given type is the appropriate type or not (for example, *std::is_pointer* is use to check the pointer type). Their parameters are the single *T (Trait class)* ones and check if T is the right type. Then, the return value of each one is boolean (true if it is the right type and reverse). These functions' syntax are shown below.

1. These functions below are used to check whether the type is "function name" type:
   - std::is_pointer
   - std::is_union
   - std::is_class
   - std::is_floating_point
   - std::is_enum
   - std::is_integral
2. Syntax:
```
        template <class T> struct is_"function name"

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

4. Illustration:

   Take *std::is_pointer* as an example, the illustration of the program is given below:

```
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

```

9 References:

1. C++ Template: The complete guild - David Vandevoorde, Nicolai M. Josuttis - Chaper 17: Metaprogramming
2. C++ Template metaprogramming: concepts, tools, techniques, and techniques from Boost and Beyond - David Abrahams, Aleksey Gurtovoy
3. Definition of metaprogramming
4. Definition of template metaprogramming
5. A gentle introduction to template metaprogramming
6. Template metaprogramming - Benefits and drawbacks of template metaprogramming
7. Template metaprogramming in C
8. Benefits and drawbacks of metaprogramming
9. Tips for making TMP code more expressive
10. How to write a metaprogramming
11. Why MPL