# 6

# Optimizing Code with Metaprogramming

We discussed the optimizing techniques using lazy evaluation in the previous chapter, and used the delaying process, caching technique, and memoization to make our code run fast. In this chapter, we will optimize the code using **metaprogramming**, where we will create a code that will create more code. The topics we will discuss in this chapter are as follows:

- Introduction to metaprogramming
- The part that builds the template metaprogramming
- Refactoring flow control into template metaprogramming
- Running the code in the compile-time execution
- The advantages and disadvantages of template metaprogramming

## Introduction to metaprogramming

The simplest way to say this is that metaprogramming is a technique that creates a code by using a code. Implementing metaprogramming, we write a computer program that manipulates the other programs and treats them as its data. In addition, templates are a compile-time mechanism in C++ that is **Turing-complete**, which means any computation expressible by a computer program can be computed, in some form, by a template metaprogram before runtime. It also uses recursion a lot and has immutable variables. So, in metaprogramming, we create code that will run when the code is compiled.

# Preprocessing the code using a macro

To start our discussion on metaprogramming, let's go back to the era when the ANSI C programming language was a popular language. For simplicity, we used the C preprocessor by creating a macro. The C parameterized macro is also known as **metafunctions**, and is one of the examples of metaprogramming. Consider the following parameterized macro:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

Since the C++ programming language has a drawback compatibility to the C language, we can compile the preceding macro using our C++ compiler. Let's create the code to consume the preceding macro, which will be as follows:

```
/* macro.cpp */
#include <iostream>

using namespace std;

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

auto main() -> int
{
  cout << "[macro.cpp]" << endl;

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

  // Consuming the MAX macro
  // and assign the result to z variable
  int z = MAX(x,y);

  // Displaying the result
  cout << "Max number of " << x << " and " << y;
  cout << " is " << z << endl;

  return 0;
}
```

As we can see in the preceding `macro.cpp` code, we pass two arguments to the `MAX` macro since it is a parameterized macro, which means the parameter can be obtained from the users. If we run the preceding code, we should see the following output on the console:


As we discussed at the beginning of this chapter, metaprogramming is a code that will run in compile time. By using a macro in the preceding code, we can demonstrate there's a new code generated from the `MAX` macro. The preprocessor will parse the macro in compile time and bring the new code. In compile time, the compiler modifies the code as follows:

```
auto main() -> int
{
  // same code
  // ...
  int z = (((a) > (b)) ? (a) : (b)); // <-- Notice this section

  // same code
  // ...

  return 0;
}
```

Besides a one line macro preprocessor, we can also generate a multiline macro metafunction. To achieve this, we can use the backslash character at the end of the line. Let's suppose we need to swap the two values. We can create a parameterized macro named `SWAP` and consume it like the following code:

```
/* macroswap.cpp */
#include <iostream>

using namespace std;

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

auto main() -> int
{
```

```
    cout << "[macroswap.cpp]" << endl;

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

    // Displaying original variable value
    cout << "before swapping" << endl;
    cout << "x = " << x << ", y = " << y ;
    cout << endl << endl;

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

    // Displaying swapped variable value
    cout << "after swapping" << endl;
    cout << "x = " << x << ", y = " << y;
    cout << endl;

    return 0;
  }
```

As we can see in the preceding code, we will create a multiline preprocessor macro and use backslash characters at the end of each line. Each time we invoke the `SWAP` parameterized macro, it will then be replaced with the implementation of the macro. We will see the following output on the console if we run the preceding code:


Now we have a basic understanding of the metaprogramming, especially in metafunction, we can move further in the next topics.

TIP

We use parenthesis for each variable in every implementation of the macro preprocessor because the preprocessor is simply replacing our code with the implementation of the macro. Let's suppose we have the following macro:
`MULTIPLY(a,b) (a * b)`
It won't be a problem if we pass the number as the parameters. However, if we pass an operation as the argument, a problem will occur. For instance, if we use the `MULTIPLY` macro as follows:
`MULTIPLY(x+2,y+5);`
Then the compiler will replace it as `(x+2*y+5)`. This happens because the macro just replaces the `a` variable with the `x + 2` expression and the `b` variable with the `y + 5` expression, with any additional parentheses. And because the order of multiplication is higher than addition, we will have got the result as follows:
`(x+2y+5)`
And that is not what we expect. As a result, the best approach is to use parenthesis in each variable of the parameter.

# Dissecting template metaprogramming in the Standard Library

We discussed the Standard Library in `Chapter 1`, *Diving into Modern C++*, and dealt with it in the previous chapter too. 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.

Let's pick up one of the classes in the Standard Library--the `Array` class. In the `Array` class, we can define a data type for it. When we instance the array, the compiler actually generates the code for an array of the data type we define. Now, let's try to build a simple `Array` template implementation as follows:

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

Then, we instance the `char` and `int` arrays as follows:

```
Array<char> arrChar;
Array<int> arrInt;
```

What the compiler does is it creates these two implementations of the template based on the data type we define. Although we won't see this in the code, the compiler actually creates the following code:

```
class ArrayChar
{
  char element;
};

class ArrayInt
{
  int element;
};

ArrayChar arrChar;
ArrayInt arrInt;
```

As we can see in the preceding code snippet, the template metaprogramming is a code that creates another code in compile time.

# Building the template metaprogramming

Before we go further in the template metaprogramming discussion, it's better if we discuss the skeleton that builds the template metaprogramming. There are four factors that form the template metaprogramming--**type**, **value**, **branch**, and **recursion**. In this topic, we will dig into the factors that form the template.

# Adding a value to the variable in the template

At the beginning of this chapter, we discussed the concept of metafunction when we talked about the macro preprocessor. In the macro preprocessor, we explicitly manipulate the source code; in this case, the macro (metafunction) manipulates the source code. In contrast, we work with types in C++ template metaprogramming. This means the metafunction is a function that works with types. So, the better approach to use template metaprogramming is working with type parameters only when possible. 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. Because we will code with types, the named values are `typedef`, as we can see in the following code snippet:

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

By using the preceding code, we store the `int` type to the `valueDataType` alias name so we can access the data type using the `valueDataType` variable. If we need to store a value instead of the data type to the variable, we can use `enum` so it will be the data member of the `enum` itself. Let's take a look at the following code snippet if we want to store the value:

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

Based on the preceding code snippet, we can now access the `value` variable to fetch its value.

# Mapping a function to the input parameters

We can add the variable to the template metaprogramming. Now, what we have to do next is retrieve the user parameters and map them to a function. Let's suppose we want to develop a `Multiplexer` function that will multiply two values and we have to use the template metaprogramming. The following code snippet can be used to solve this problem:

```
template<int A, int B>
struct Multiplexer
```

```
{
  enum
  {
    result = A * B
  };
};
```

As we can see in the preceding code snippet, the template requires two arguments, `A` and `B`, from the user, and it will use them to get the value of `result` variable by multiplying these two parameters. We can access the result variable using the following code:

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

If we run the preceding code snippet, the `i` variable will store `6` since it will calculate `2` times `3`.

# Choosing the correct process based on the condition

When we have more than one function, we have to choose one over the others based on certain conditions. We can construct the conditional branch by providing two alternative specializations of the `template` class, as shown here:

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

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

As we can see in the preceding `template` code, we have two templates that have `X` and `A`/`B` as their type. When the template has only a single type, that is, `typename X`, it means that the two types (`CheckingType <X, X>`) we compare are exactly the same. Otherwise, these two data types are different. The following code snippet can be used to consume the two preceding templates:

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

As we can see in the preceding code snippet, we try to compare the `UnknownType` data type with the `int` type. The `UnknownType` data type might be coming from the other process. Then, we can decide the next process we want to run by comparing these two types using templates.

> Up to here, you might wonder how template multiprogramming will help us make code optimization. Soon we will use the template metaprogramming to optimize code. However, we need to discuss other things that will solidify our knowledge in template multiprogramming. For now, please be patient and keep reading.

# Repeating the process recursively

We have successfully added value and data type to the template, then created a branch to decide the next process based on the current condition. Another thing we have to consider in the basic template is repeating the process. However, since the variable in the template is immutable, we cannot iterate the sequence. Instead, we have to recur the process as we discussed in `Chapter 4`, *Repeating Method Invocation Using Recursive Algorithm*.

Let's suppose we are developing a template to calculate the factorial value. The first thing we have to do is develop a general template that passes the `I` value to the function as follows:

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

```
    };
  };
```

As we can see in the preceding code, we can obtain the value of the factorial by running the following code:

```
Factorial<I>::value;
```

In the preceding code, `I` is an integer number.
Next, we have to develop a template to ensure that it doesn't end up with an infinite loop. We can create the following template that passes zero (`0`) as a parameter to it:

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

Now we have a pair of templates that will generate the value of the factorial in compile time. The following is a sample code to get the value of `Factorial(10)` in compile time:

```
int main()
{
  int fact10 = Factorial<10>::value;
}
```

If we run the preceding code, we will get `3628800` as a result of the factorial of `10`.

# Selecting a type in compile-time

As we discussed in the preceding topic, `type` is a basic part of a template. However, we can select a certain type based on the input from the user. Let's create a template that can decide what type should be used in the variable. The following `types.cpp` code will show the implementation of the template:

```
/* types.cpp */
#include <iostream>

using namespace std;

// Defining a data type
// in template
```

```
template<typename T>
struct datatype
{
  using type = T;
};

auto main() -> int
{
  cout << "[types.cpp]" << endl;

  // Selecting a data type in compile time
  using t = typename datatype<int>::type;

  // Using the selected data type
  t myVar = 123;

  // Displaying the selected data type
  cout << "myVar = " << myVar;

  return 0;
}
```

As we can see in the preceding code, we have a template named `datatype`. This template can be used to select the `type` we pass to it. We can use the `using` keyword to assign a variable to a `type`. From the preceding `types.cpp` code, we will assign a `t` variable to `type` from the `datatype` template. The `t` variable now will be `int` since we passed the `int` data type to the template.

We can also create a code to select the correct data type based on the current condition. We will have an `IfElseDataType` template that takes three arguments which are `predicate`, the data type when the `predicate` parameter is true, and the data type when the `predicate` parameter is false. The code will look as follows:

```
/* selectingtype.cpp */
#include <iostream>

using namespace std;

// Defining IfElseDataType template
template<
  bool predicate,
  typename TrueType,
  typename FalseType>
  struct IfElseDataType
  {
  };
```

```
// Defining template for TRUE condition
// passed to 'predicate' parameter
template<
  typename TrueType,
  typename FalseType>
  struct IfElseDataType<
   true,
   TrueType,
   FalseType>
   {
     typedef TrueType type;
   };

// Defining template for FALSE condition
// passed to 'predicate' parameter
template<
  typename TrueType,
  typename FalseType>
  struct IfElseDataType<
  false,
  TrueType,
  FalseType>
  {
     typedef FalseType type;
  };

auto main() -> int
{
  cout << "[types.cpp]" << endl;

  // Consuming template and passing
  // 'SHRT_MAX == 2147483647'
  // It will be FALSE
  // since the maximum value of short
  // is 32767
  // so the data type for myVar
  // will be 'int'
  IfElseDataType<
    SHRT_MAX == 2147483647,
    short,
    int>::type myVar;

  // Assigning myVar to maximum value
  // of 'short' type
  myVar = 2147483647;

  // Displaying the data type of myVar
  cout << "myVar has type ";
```

```
        cout << typeid(myVar).name() << endl;

        return 0;
    }
```

Now, by having the `IfElseDataType` template, we can select the correct type to the variable based on the condition we have. Let's suppose we want to assign `2147483647` to a variable so we can check if it's a short number. If so, `myVar` will be of type `short`, otherwise, it will be `int`. Moreover, since the maximum value of `short` type is `32767`, by giving the predicate as `SHRT_MAX == 2147483647` will be resulting `FALSE`. Therefore, the type of `myVar` will be an `int` type, as we can see in the following output that will appear on the console:


# Flow control with template metaprogramming

Code flow is an important aspect in coding a program. In many programming languages, they have an `if-else`, `switch`, and `do-while` statement to arrange the flow of the code. Now, let's refactor the usual flow of code to become a template-based flow. We will start by using the `if-else` statement, followed by the `switch` statement, and finally ending with the `do-while` statement, all in templates.

## Deciding the next process by the current condition

Now it's time to use the template as we discussed previously. Let's suppose we have two functions that we have to choose by a certain condition. What we usually do is use the `if-else` statement as follows:

```
/* condition.cpp */
#include <iostream>

using namespace std;
```

```
// Function that will run
// if the condition is TRUE
void TrueStatement()
{
  cout << "True Statement is run." << endl;
}

// Function that will run
// if the condition is FALSE
void FalseStatement()
{
  cout << "False Statement is run." << endl;
}

auto main() -> int
{
  cout << "[condition.cpp]" << endl;

  // Choosing the function
  // based on the condition
  if (2 + 3 == 5)
    TrueStatement();
  else
    FalseStatement();

  return 0;
}
```

As we can see in the preceding code, we have two functions--`TrueStatement()` and `FalseStatement()`. We also have a condition in the code--`2 + 3 == 5`. And since the condition is `TRUE`, then the `TrueStatement()` function will be run as we can see in the following screenshot:


Now, let's refactor the preceding `condition.cpp` code. We will create three templates here. First, the template initialization that inputs the condition as follows:

```
template<bool predicate> class IfElse
```

Then, we create two templates for each condition--`TRUE` or `FALSE`. The name will be as follows:

```
template<> class IfElse<true>
template<> class IfElse<false>
```

Each template in the preceding code snippet will run the functions we have created before--the `TrueStatement()` and `FalseStatement()` functions. And we will get the complete code as the following `conditionmeta.cpp` code:

```
/* conditionmeta.cpp */
#include <iostream>

using namespace std;

// Function that will run
// if the condition is TRUE
void TrueStatement()
{
  cout << "True Statement is run." << endl;
}

// Function that will run
// if the condition is FALSE
void FalseStatement()
{
  cout << "False Statement is run." << endl;
}

// Defining IfElse template
template<bool predicate>
class IfElse
{
};

// Defining template for TRUE condition
// passed to 'predicate' parameter
template<>
class IfElse<true>
{
  public:
    static inline void func()
    {
      TrueStatement();
    }
};

// Defining template for FALSE condition
```

```
// passed to 'predicate' parameter
template<>
class IfElse<false>
{
  public:
    static inline void func()
    {
      FalseStatement();
    }
};

auto main() -> int
{
  cout << "[conditionmeta.cpp]" << endl;

  // Consuming IfElse template
  IfElse<(2 + 3 == 5)>::func();

  return 0;
}
```

As we can see, we put the condition on the bracket of the `IfElse` template, then call the `func()` method inside the template. If we run the `conditionmeta.cpp` code, we will get the exact same output such as the `condition.cpp` code, as shown here:

```
Command Prompt
[conditionmeta.cpp]
True Statement is run.
```

We now have the `if-else` statement to flow our code in the template metaprogramming.

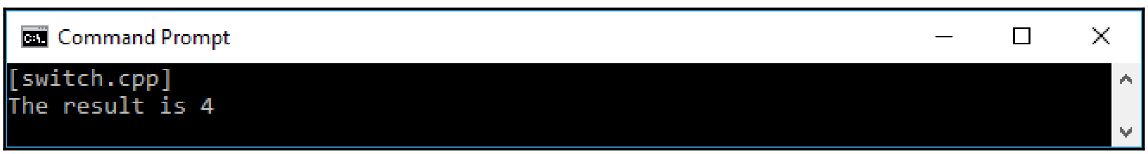
# Selecting the correct statement

In C++ programming, and other programming languages as well, we use the `switch` statement to select a certain process based on the value we give to the `switch` statement. If the value matches with the one of the switch case, it will run the process under that case. Let's take a look at the following `switch.cpp` code that implements the `switch` statement:

```
/* switch.cpp */
#include <iostream>

using namespace std;
```

```
// Function to find out
// the square of an int
int Square(int a)
{
  return a * a;
}

auto main() -> int
{
  cout << "[switch.cpp]" << endl;

  // Initializing two int variables
  int input = 2;
  int output = 0;

  // Passing the correct argument
  // to the function
  switch (input)
  {
    case 1:
        output = Square(1);
        break;
    case 2:
        output = Square(2);
        break;
    default:
        output = Square(0);
        break;
  }

  // Displaying the result
  cout << "The result is " << output << endl;

  return 0;
}
```

As we can see in the preceding code, we have a function named `Square()` that takes an argument. The argument we pass to it is based on the value that we give to the switch statement. Since the value we pass to switch is `2`, the `Square(2)` method will be run. The following screenshot is what we will see on the console screen:



To refactor the `switch.cpp` code to template metaprogramming, we have to create three templates that consist of the function we plan to run. First, we will create the initialization template to retrieve the value from the user, as follows:

```
template<int val> class SwitchTemplate
```

The preceding initialization template will also be used for the default value. Next, we will add two templates for each possible value as follows:

```
template<> class SwitchTemplate<1>
template<> class SwitchTemplate<2>
```

Each preceding template will run the `Square()` function and pass the argument based on the value of the template. The complete code is written as follows:

```
/* switchmeta.cpp */
#include <iostream>

using namespace std;

// Function to find out
// the square of an int
int Square(int a)
{
  return a * a;
}

// Defining template for
// default output
// for any input value
template<int val>
class SwitchTemplate
{
  public:
    static inline int func()
    {
      return Square(0);
    }
};

// Defining template for
// specific input value
// 'val' = 1
template<>
class SwitchTemplate<1>
{
   public:
     static inline int func()
```

```
        {
          return Square(1);
        }
};

// Defining template for
// specific input value
// 'val' = 2
template<>
class SwitchTemplate<2>
{
   public:
     static inline int func()
     {
        return Square(2);
     }
};

auto main() -> int
{
  cout << "[switchmeta.cpp]" << endl;

  // Defining a constant variable
  const int i = 2;

  // Consuming the SwitchTemplate template
  int output = SwitchTemplate<i>::func();

  // Displaying the result
  cout << "The result is " << output << endl;

  return 0;
}
```

As we can see, we do the same as `conditionmeta.cpp`--we call the `func()` method inside the template to run the selected function. The value for this `switch-case` condition is the template we put in the angle bracket. If we run the preceding `switchmeta.cpp` code, we will see the following output on the console:

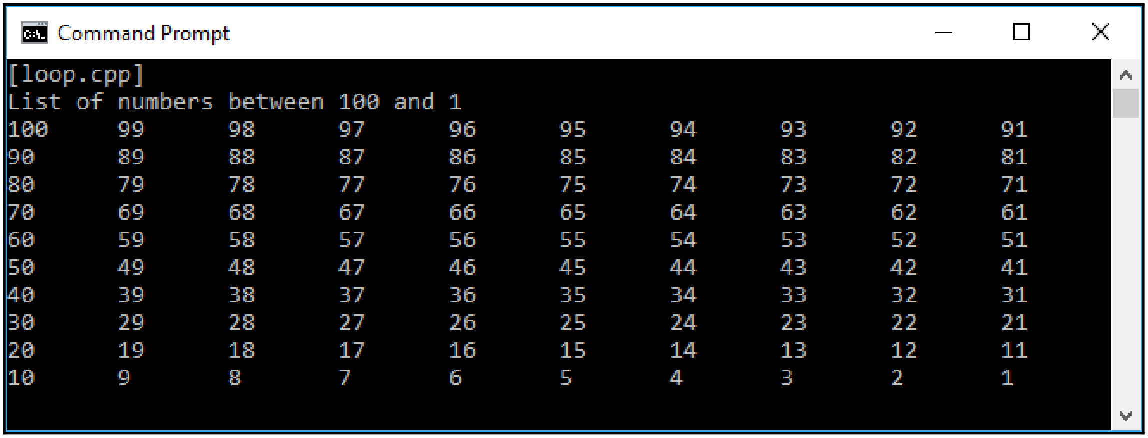
As we can see in the preceding screenshot, we've got the exact same output for switchmeta.cpp code as compared to the switch.cpp code. Thus, we have successfully refactored the switch.cpp code into the template metaprogramming.

# Looping the process

We usually use the do-while loop when we iterate something. Let's suppose we need to print certain numbers until it reaches zero (0). The code is as follows:

```
/* loop.cpp */
#include <iostream>

using namespace std;

// Function for printing
// given number
void PrintNumber(int i)
{
  cout << i << "\t";
}

auto main() -> int
{
  cout << "[loop.cpp]" << endl;

  // Initializing an int variable
  // marking as maximum number
  int i = 100;

  // Looping to print out
  // the numbers below i variable
  cout << "List of numbers between 100 and 1";
  cout << endl;
  do
  {
    PrintNumber(i);
  }
  while (--i > 0);
  cout << endl;

  return 0;
}
```

As we can see in the preceding code, we will print the number `100`, decrease its value, and print again. It will always run until the number reaches zero (`0`). The output on the console should be as follows:




Now, let's refactor it to the template metaprogramming. Here, we need only two templates to achieve the `do-while` loop in template metaprogramming. First, we will create the following template:

```
template<int limit> class DoWhile
```

The limit in the preceding code is the value that is passed to the `do-while` loop. And, to not make the loop become an infinite loop, we have to design the `DoWhile` template when it has reached zero (`0`), as shown here:

```
template<> class DoWhile<0>
```

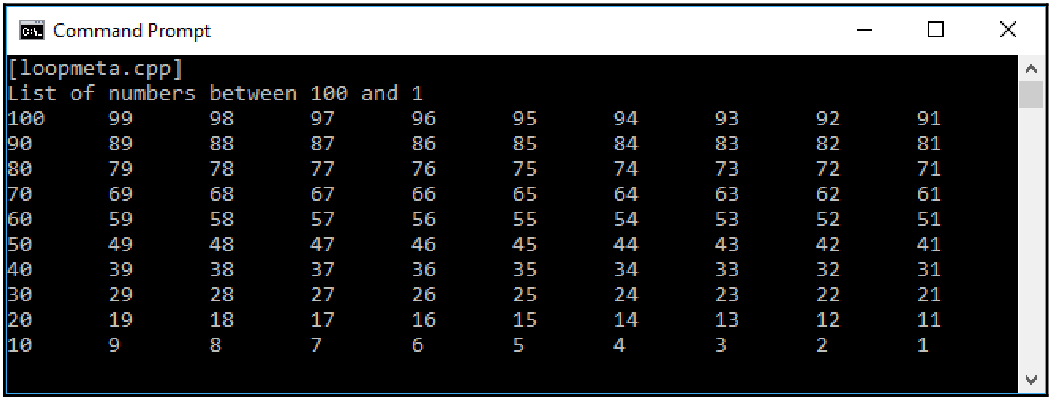
The preceding template will do nothing since it's used only to break the loop. The complete refactoring of the `do-while` loop is like the following `loopmeta.cpp` code:

```
/* loopmeta.cpp */
#include <iostream>

using namespace std;

// Function for printing
// given number
void PrintNumber(int i)
{
  cout << i << "\t";
}
```

```
// Defining template for printing number
// passing to its 'limit' parameter
// It's only run
// if the 'limit' has not been reached
template<int limit>
class DoWhile
{
   private:
     enum
     {
       run = (limit-1) != 0
     };

   public:
     static inline void func()
     {
       PrintNumber(limit);
       DoWhile<run == true ? (limit-1) : 0>
        ::func();
     }
};

// Defining template for doing nothing
// when the 'limit' reaches 0
template<>
class DoWhile<0>
{
  public:
    static inline void func()
    {
    }
};

auto main() -> int
{
  cout << "[loopmeta.cpp]" << endl;

  // Defining a constant variable
  const int i = 100;

  // Looping to print out
  // the numbers below i variable
  // by consuming the DoWhile
  cout << "List of numbers between 100 and 1";
  cout << endl;
  DoWhile<i>::func();
  cout << endl;
```

```
    return 0;
}
```

We then call the `func()` method inside the template to run our desired function. And, if we run the code, we will see the following output on the screen:




Again, we have successfully refactored the `loop.cpp` code into `loopmeta.cpp` code since both have the exact same output.

# Executing the code in compile-time

As we discussed earlier, template metaprogramming will run the code in compile-time by creating a new code. Now, let's see how we can get the compile-time constant and generate a compile-time class in this section.

# Getting a compile-time constant

To retrieve a compile-time constant, let's create a code that has the template for a Fibonacci algorithm in it. We will consume the template so the compiler will provide the value in compile time. The code should be as follows:

```
/* fibonaccimeta.cpp */
#include <iostream>

using namespace std;

// Defining Fibonacci template
// to calculate the Fibonacci sequence
```

```
template <int number>
struct Fibonacci
{
  enum
  {
    value =
        Fibonacci<number - 1>::value +
        Fibonacci<number - 2>::value
  };
};

// Defining template for
// specific input value
// 'number' = 1
template <>
struct Fibonacci<1>
{
  enum
  {
    value = 1
  };
};

// Defining template for
// specific input value
// 'number' = 0
template <>
struct Fibonacci<0>
{
  enum
  {
    value = 0
  };
};

auto main() -> int
{
  cout << "[fibonaccimeta.cpp]" << endl;

  // Displaying the compile-time constant
  cout << "Getting compile-time constant:";
  cout << endl;
  cout << "Fibonacci(25) = ";
  cout << Fibonacci<25>::value;
  cout << endl;

  return 0;
}
```

As we can see in the preceding code, the value variable in the Fibonacci template will provide a compile-time constant. And if we run the preceding code, we will see the following output on the console screen:


Now, we have `75025` that is generated by the compiler as a compile-time constant.

## Generating the class using a compile-time class generation

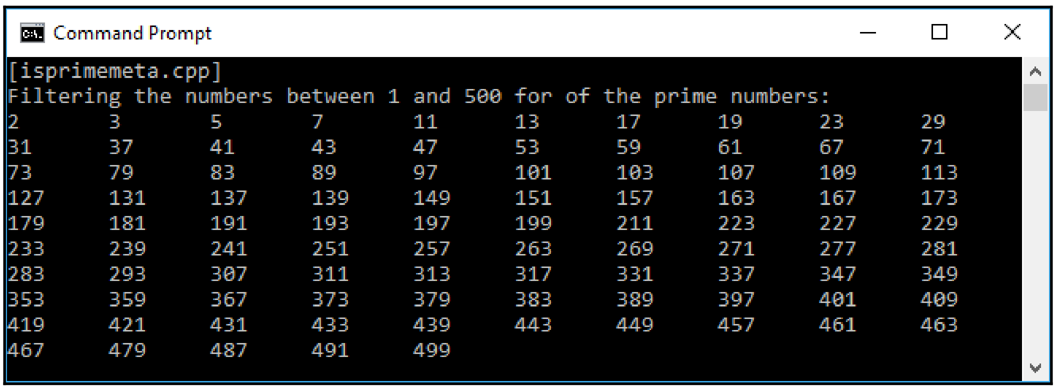
Besides the generation of a compile-time constant, we will also generate the class in compile time. Let's suppose we have a template to find out the prime number in the range `0` to `X`. The following `isprimemeta.cpp` code will explain the implementation of the template metaprogramming to find the prime number:

```
/* isprimemeta.cpp */
#include <iostream>

using namespace std;

// Defining template that decide
// whether or not the passed argument
// is a prime number
template <
  int lastNumber,
  int secondLastNumber>
class IsPrime
{
  public:
    enum
    {
      primeNumber = (
        (lastNumber % secondLastNumber) &&
        IsPrime<lastNumber, secondLastNumber - 1>
            ::primeNumber)
    };
 };
```

```
// Defining template for checking
// the number passed to the 'number' parameter
// is a prime number
template <int number>
class IsPrime<number, 1>
{
  public:
    enum
    {
      primeNumber = 1
    };
};

// Defining template to print out
// the passed argument is it's a prime number
template <int number>
class PrimeNumberPrinter
{
  public:
    PrimeNumberPrinter<number - 1> printer;

  enum
  {
    primeNumber = IsPrime<number, number - 1>
        ::primeNumber
  };

  void func()
  {
    printer.func();

    if (primeNumber)
    {
        cout << number << "\t";
    }
  }
};

// Defining template to just ignoring the number
// we pass 1 as argument to the parameter
// since 1 is not prime number
template<>
class PrimeNumberPrinter<1>
{
  public:
    enum
    {
      primeNumber = 0
```

```
        };

        void func()
        {
        }
    };

    int main()
    {
      cout << "[isprimemeta.cpp]" << endl;

      // Displaying the prime numbers between 1 and 500
      cout << "Filtering the numbers between 1 and 500 ";
      cout << "for of the prime numbers:" << endl;

      // Consuming PrimeNumberPrinter template
      PrimeNumberPrinter<500> printer;

      // invoking func() method from the template
      printer.func();

      cout << endl;
      return 0;
    }
```

There are two kinds of templates with different roles--the **prime checker**, that ensures the number that is passed is a prime number, and the **printer**, that displays the prime number to the console. The compiler then generates the class in compile-time when the code accesses `PrimeNumberPrinter<500> printer` and `printer.func()`. And when we run the preceding `isprimemeta.cpp` code, we will see the following output on the console screen:



Since we pass `500` to the template, we will get the prime number from `0` to `500`. The preceding output has proven that the compiler has successfully generated a compile-time class so we can get the correct value.

# Benefits and drawbacks of metaprogramming

After our discussion about template metaprogramming, the following are the advantages we derive:

- Template metaprogramming has no side effect since it is immutable, so we cannot modify an existing type
- There is better code readability compared to code that does not implement metaprogramming
- It reduces repetition of the code

Although we can gain benefits from template metaprogramming, there are several disadvantages, which are as follows:

- The syntax is quite complex.
- The compilation time takes longer since we now execute code during compile-time.
- The compiler can optimize the generated code much better and perform inlining, for instance, the C `qsort()` function and the C++ `sort` template. In C, the `qsort()` function takes a pointer to a comparison function, so there will be one copy of the `qsort` code that is not inlined. It will make a call through the pointer to the comparison routine. In C++, `std::sort` is a template, and it can take a `functor` object as a comparator. There is a different copy of `std::sort` for each different type used as a comparator. If we use a `functor` class with an overloaded `operator()` function, the call to the comparator can easily be inlined into this copy of `std::sort`.

# Summary

Metaprogramming, especially template metaprogramming, creates new code for us automatically so we don't need to write a lot of code in our source. By using template metaprogramming, we can refactor the flow control of our code as well as run the code in compile-time execution.

In the next chapter, we will talk about concurrency techniques that will bring a responsive enhancement to the application that we build. We can run the processes in our code simultaneously using the parallelism technique.