Template metaprogramming

From Wikipedia, the free encyclopedia

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 include compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time execution. The technique is used by a number of languages, the best-known being C++, but also Curl, D, and XL.

Template metaprogramming was, in a sense, discovered accidentally.[1]

Some other languages support similar, if not more powerful compile-time facilities (such as Lisp macros), but those are outside the scope of this article.

Contents

Components of template metaprogramming

The use of templates as a metaprogramming technique requires two distinct operations: a template must be defined, and a defined template must be instantiated. The template definition describes the generic form of the generated source code, and the instantiation causes a specific set of source code to be generated from the generic form in the template.

Template metaprogramming is Turing-complete, meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram.[2]

Templates are different from *macros*. A macro, which is also a compile-time language feature, generates code in-line using text manipulation and substitution. Macro systems often have limited compile-time process flow abilities and usually lack awareness of the semantics and type system of their companion language (an exception should be made with Lisp's macros, which are written in Lisp itself and involve manipulation and substitution of Lisp code represented as data structures as opposed to text).

Template metaprograms have no mutable variables— that is, no variable can change value once it has been initialized, therefore template metaprogramming can be seen as a form of functional programming. In fact many template implementations implement flow control only through recursion, as seen in the example below.

Using template metaprogramming

Though the syntax of template metaprogramming is usually very different from the programming language it is used with, it has practical uses. Some common reasons to use templates are to implement generic programming (avoiding sections of code which are similar except for some minor variations) or to perform automatic compile-time optimization such as doing something once at compile time rather than every time the program is run — for instance, by having the compiler unroll loops to eliminate jumps and loop count decrements whenever the program is executed.

Compile-time class generation

What exactly "programming at compile-time" means can be illustrated with an example of a factorial function, which in non-template C++ can be written using recursion as follows:

```
unsigned int factorial(unsigned int n) {
    return n == 0 ? 1 : n * factorial(n - 1);
}

// Usage examples:
// factorial(0) would yield 1;
// factorial(4) would yield 24.
```

The code above will execute at run time to determine the factorial value of the literals 4 and 0. By using template metaprogramming and template specialization to provide the ending condition for the recursion, the factorials used in the program—ignoring any factorial not used—can be calculated at compile time by this code:

```
template <unsigned int n>
struct factorial {
    enum { value = n * factorial<n - 1>::value };
};

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

// Usage examples:
// factorial<0>::value would yield 1;
// factorial<4>::value would yield 24.
```

The code above calculates the factorial value of the literals 4 and 0 at compile time and uses the result as if they were precalculated constants. To be able to use templates in this manner, the compiler must know the value of its parameters at compile time, which has the natural precondition that factorial<X>::value can only be used if X is known at compile time. In other words, X must be a constant literal or a constant expression.

In C++11, constexpr, a way to let the compiler execute simple constant expressions, was added. Using constexpr, one can use the usual recursive factorial definition.[3]

Compile-time code optimization

The factorial example above is one example of compile-time code optimization in that all factorials used by the program are pre-compiled and injected as numeric constants at compilation, saving both run-time overhead and memory footprint. It is, however, a relatively minor optimization.

As another, more significant, example of compile-time loop unrolling, template metaprogramming can be used to create length-*n* vector classes (where *n* is known at compile time). The benefit over a more traditional length-*n* vector is that the loops can be unrolled, resulting in very optimized code. As an example, consider the addition operator. A length-*n* vector addition might be written as

```
template <int length>
Vector<length>& Vector<length>::operator+=(const Vector<length>& rhs)
{
    for (int i = 0; i < length; ++i)
        value[i] += rhs.value[i];
    return *this;
}
```

When the compiler instantiates the function template defined above, the following code may be produced:

```
template <>
Vector<2>& Vector<2>::operator+=(const Vector<2>& rhs)
{
    value[0] += rhs.value[0];
    value[1] += rhs.value[1];
    return *this;
}
```

The compiler's optimizer should be able to unroll the `for` loop because the template parameter `length` is a constant at compile time.

However, take caution as this may cause code bloat as separate unrolled code will be generated for each 'N'(vector size) you instantiate with.

Static polymorphism

Polymorphism is a common standard programming facility where derived objects can be used as instances of their base object but where the derived objects' methods will be invoked, as in this code

```
class Base
{
public:
    virtual void method() { std::cout << "Base"; }
    virtual ~Base() {}
};

class Derived : public Base
{
public:
    virtual void method() { std::cout << "Derived"; }
};

int main()
{
    Base *pBase = new Derived;
    pBase->method(); //outputs "Derived"
```

```
    delete pBase;
    return 0;
}
```

where all invocations of virtual methods will be those of the most-derived class. This *dynamically polymorphic* behaviour is (typically) obtained by the creation of virtual look-up tables for classes with virtual methods, tables that are traversed at run time to identify the method to be invoked. Thus, *run-time polymorphism* necessarily entails execution overhead (though on modern architectures the overhead is small).

However, in many cases the polymorphic behaviour needed is invariant and can be determined at compile time. Then the Curiously Recurring Template Pattern (CRTP) can be used to achieve static polymorphism, which is an imitation of polymorphism in programming code but which is resolved at compile time and thus does away with run-time virtual-table lookups. For example:

```
template <class Derived>
struct base
{
    void interface()
    {
        // ...
        static_cast<Derived*>(this)->implementation();
        // ...
    }
};

struct derived : base<derived>
{
    void implementation()
    {
        // ...
    }
};
```

Here the base class template will take advantage of the fact that member function bodies are not instantiated until after their declarations, and it will use members of the derived class within its own member functions, via the use of a static_cast, thus at compilation generating an object composition with polymorphic characteristics. As an example of real-world usage, the CRTP is used in the Boost iterator library.[4]

Another similar use is the "Barton–Nackman trick", sometimes referred to as "restricted template expansion", where common functionality can be placed in a base class that is used not as a contract but as a necessary component to enforce conformant behaviour while minimising code redundancy.

Benefits and drawbacks of template metaprogramming

Compile-time versus execution-time tradeoff
: If a great deal of template metaprogramming is used, compilation may become slow; section 14.7.1 [temp.inst] of the current standard defines the circumstances under which templates are implicitly instantiated. Defining a template does not imply that it will be instantiated, and instantiating a class template does not cause its member definitions to be instantiated. Depending on the style of use, templates may compile either faster or slower than hand-rolled code.

Generic programming
: 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.

Readability
: With respect to C++, the syntax and idioms of template metaprogramming are esoteric compared to conventional C++ programming, and template metaprograms can be very difficult to understand. [5][6]

See also

- Substitution failure is not an error (SFINAE)
- Metaprogramming
- Preprocessor
- Parametric polymorphism
- Expression templates
- Variadic Templates
- Compile time function execution

References

1. See History of TMP on Wikibooks
2. Veldhuizen, Todd L. "C++ Templates are Turing Complete".
3. http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html

4. http://www.boost.org/libs/iterator/doc/iterator_facade.html
5. Czarnecki, K.; O'Donnell, J.; Striegnitz, J.; Taha, Walid Mohamed (2004). "DSL implementation in metaocaml, template haskell, and C++" (PDF). University of Waterloo, University of Glasgow, Research Centre Julich, Rice University. "*C++ Template Metaprogramming suffers from a number of limitations, including portability problems due to compiler limitations (although this has significantly improved in the last few years), lack of debugging support or IO during template instantiation, long compilation times, long and incomprehensible errors, poor readability of the code, and poor error reporting.*"
6. Sheard, Tim; Jones, Simon Peyton (2002). "Template Meta-programming for Haskell" (PDF). ACM 1-58113-415-0/01/0009. "*Robinson's provocative paper identifies C++ templates as a major, albeit accidental, success of the C++ language design. Despite the extremely baroque nature of template meta-programming, templates are used in fascinating ways that extend beyond the wildest dreams of the language designers. Perhaps surprisingly, in view of the fact that templates are functional programs, functional programmers have been slow to capitalize on C++'s success*"

- Eisenecker, Ulrich W. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. ISBN 0-201-30977-7.
- Alexandrescu, Andrei. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley. ISBN 3-8266-1347-3.
- Abrahams, David; Gurtovoy, Aleksey. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley. ISBN 0-321-22725-5.
- Vandervoorde, David; Josuttis, Nicolai M. *C++ Templates: The Complete Guide*. Addison-Wesley. ISBN 0-201-73484-2.
- Clavel, Manuel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. ISBN 1-57586-238-7.

External links

- "The Boost Metaprogramming Library (Boost MPL)".
- "The Spirit Library". (built using template-metaprogramming)
- "The Boost Lambda library". (use STL algorithms easily)
- Veldhuizen, Todd (May 1995). "Using C++ template metaprograms". *C++ Report*. 7 (4): 36–43. Archived from the original on 2009-03-04.
- "Template Haskell". (type-safe metaprogramming in Haskell)
- Bright, Walter. "Templates Revisited". (template metaprogramming in the D programming language)
- Koskinen, Johannes. "Metaprogramming in C++" (PDF).
- Attardi, Giuseppe; Cisternino, Antonio. "Reflection support by means of template metaprogramming" (PDF).
- Burton, Michael C.; Griswold, William G.; McCulloch, Andrew D.; Huber, Gary A. "Static data structures". CiteSeerX 10.1.1.14.5881.
- Amjad, Zeeshan. "Template Meta Programming and Number Theory".
- Amjad, Zeeshan. "Template Meta Programming and Number Theory: Part 2".
- "A library for LISP-style programming in C++".