This article first appeared in the May 1995 issue of C++ Report. It has been reprinted in the book "*C++ Gems*" edited by Stanley Lippman.

T. Veldhuizen, "Using C++ template metaprograms," *C++ Report* Vol. 7 No. 4 (May 1995), pp. 36-43.

C++ Report (ISSN 1040-6042) is published nine times per year by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1995 SIGS Publications Inc.

Back to the Blitz++ Home Page
Back to Todd's Home Page

Template Metaprograms

Todd Veldhuizen

Introduction

Compile-time programs

The introduction of templates to C++ added a facility whereby the compiler can act as an interpreter. This makes it possible to write programs in a subset of C++ which are interpreted at compile time. Language features such as for loops and if statements can be replaced by template specialization and recursion. The first examples of these techniques were written by Erwin Unruh and circulated among members of the ANSI/ISO C++ standardization committee [1]. These programs didn't have to be executed -- they generated their output at compile time as warning messages. For example, one program generated warning messages containing prime numbers when compiled.

Here's a simple example which generates factorials at compile time:

```
template<int N>
class Factorial {
public:
    enum { value = N * Factorial<N-1>::value };
};

class Factorial<1> {
public:
    enum { value = 1 };
};
```

Using this class, the value N! is accessible at compile time as Factorial<N>::value. How does this work? When Factorial<N> is instantiated, the compiler needs Factorial<N-1> in order to assign the enum 'value'. So it instantiates Factorial<N-1>, which in turn requires Factorial<N-2>, requiring Factorial<N-3>, and so on until Factorial<1> is reached, where template specialization is used to end the recursion. The compiler effectively performs a for loop to evaluate N! at compile time.

Although this technique might seem like just a cute C++ trick, it becomes powerful when combined with normal C++ code. In this hybrid approach, source code contains two programs: the normal C++ run-time program, and a template metaprogram which runs at compile time. Template metaprograms can generate useful code when interpreted by the compiler, such as a massively inlined algorithm -- that is, an implementation of an algorithm which works for a specific input size, and has its loops unrolled. This results in large speed increases for many applications.

Here is a simple template metaprogram "recipe" for a bubble sort algorithm.

An example: Bubble Sort

Although bubble sort is a very inefficient algorithm for large arrays, it's quite reasonable for small N. It's also the simplest sorting algorithm. This makes it a good example to illustrate how template metaprograms can be used to generate specialized algorithms. Here's a typical bubble sort implementation, to sort an array of ints:

```
inline void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}


void bubbleSort(int* data, int N)
{
    for (int i = N - 1; i > 0; --i)
    {
        for (int j = 0; j < i; ++j)
        {
            if (data[j] > data[j+1])
                swap(data[j], data[j+1]);
        }
    }
}
```

A specialized version of bubble sort for N=3 might look like this:

```
inline void bubbleSort3(int* data)
{
    int temp;

    if (data[0] > data[1])
    { temp = data[0]; data[0] = data[1]; data[1] = temp; }
    if (data[1] > data[2])
    { temp = data[1]; data[1] = data[2]; data[2] = temp; }
    if (data[0] > data[1])
    { temp = data[0]; data[0] = data[1]; data[1] = temp; }
}
```

In order to generate an inlined bubble sort such as the above, it seems that we'll have to unwind two loops. We can reduce the number of loops to one, by using a recursive version of bubble sort:

```
void bubbleSort(int* data, int N)
{
    for (int j = 0; j < N - 1; ++j)
    {
        if (data[j] > data[j+1])
            swap(data[j], data[j+1]);
    }

    if (N > 2)
```

```
        bubbleSort(data, N-1);
}
```

Now the sort consists of a loop, and a recursive call to itself. This structure is simple to implement using some template classes:

```
template<int N>
class IntBubbleSort {
public:
    static inline void sort(int* data)
    {
        IntBubbleSortLoop<N-1,0>::loop(data);
        IntBubbleSort<N-1>::sort(data);
    }
};


class IntBubbleSort<1> {
public:
    static inline void sort(int* data)
    { }
};
```

To sort an array of N integers, we invoke IntBubbleSort<N>::sort(int* data). This routine calls a function IntBubbleSortLoop<N-1,0>::loop(data), which will replace the for loop in j, then makes a recursive call to itself. A template specialization for N=1 is provided to end the recursive calls. We can manually expand this for N=4 to see the effect:

```
static inline void IntBubbleSort<4>::sort(int* data)
{
    IntBubbleSortLoop<3,0>::loop(data);
    IntBubbleSortLoop<2,0>::loop(data);
    IntBubbleSortLoop<1,0>::loop(data);
}
```

The first template argument in the IntBubbleSortLoop classes (3,2,1) is the value of i in the original version of bubbleSort(), so it makes sense to call this argument I. The second template parameter will take the role of j in the loop, so we'll call it J. Now we need to write the IntBubbleSortLoop class. It needs to loop from J=0 to J=I-2, comparing elements data[J] and data[J+1] and swapping them if necessary:

```
template<int I, int J>
class IntBubbleSortLoop {

private:
    enum { go = (J <= I-2) };

public:
    static inline void loop(int* data)
    {
        IntSwap<J,J+1>::compareAndSwap(data);
        IntBubbleSortLoop<go ? I : 0, go ? (J+1) : 0>::loop(data);
    }
};


class IntBubbleSortLoop<0,0> {

public:
    static inline void loop(int*)
    { }
};
```

Writing a base case for this recursion is a little more difficult, since we have two variables. The solution is to make both template parameters revert to 0 when the base case is reached. This is accomplished by storing the loop flag in an enumerative type (go), and using a conditional expression operator (?:) to force the template parameters to 0 when ’go’ is false. The class IntSwap<I,J> will perform the task of swapping data[I] and data[J] if necessary. Once again, we can manually expand IntBubbleSort<4>::sort(data) to see what is being generated:

```
static inline void IntBubbleSort<4>::sort(int* data)
{
    IntSwap<0,1>::compareAndSwap(data);
    IntSwap<1,2>::compareAndSwap(data);
    IntSwap<2,3>::compareAndSwap(data);
    IntSwap<0,1>::compareAndSwap(data);
    IntSwap<1,2>::compareAndSwap(data);
    IntSwap<0,1>::compareAndSwap(data);
}
```

The last remaining definition is the IntSwap<I,J>::compareAndSwap() routine:

```
template<int I, int J>
class IntSwap {

public:
    static inline void compareAndSwap(int* data)
    {
        if (data[I] > data[J])
            swap(data[I], data[J]);
    }
};
```

The swap() routine is the same as before:

```
inline void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

It’s easy to see that this results in code equivalent to

```
static inline void IntBubbleSort<4>::sort(int* data)
{
    if (data[0] > data[1]) swap(data[0], data[1]);
    if (data[1] > data[2]) swap(data[1], data[2]);
    if (data[2] > data[3]) swap(data[2], data[3]);
    if (data[0] > data[1]) swap(data[0], data[1]);
    if (data[1] > data[2]) swap(data[1], data[2]);
    if (data[0] > data[1]) swap(data[0], data[1]);
}
```

In the next section, the speed of this version is compared to the original version of bubbleSort().

Practical Issues

Performance

The goal of algorithm specialization is to improve performance. Figure 3 shows benchmarks comparing the time required to sort arrays of different sizes, using the specialized bubble sort versus a call to the bubbleSort() routine. These benchmarks were obtained on an 80486/66Mhz machine running Borland C++ V4.0. For small arrays (3-5 elements), the inlined version is 7 to 2.5 times faster than the bubbleSort() routine. As the array size gets larger, the advantage levels off to roughly 1.6.

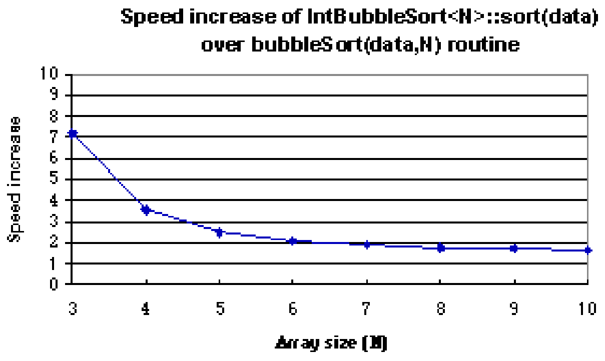

Figure 3. Performance increase achieved by specializing the bubble sort algorithm.

The curve in Figure 3 is the general shape of the performance increase for algorithm specialization. For very small input sizes, the advantage is very good since the overhead of function calls and setting up stack variables is avoided. As the input size gets larger, the performance increase settles out to some constant value which depends on the nature of the algorithm and the processor for which code is generated. In most situations, this value will be greater than 1, since the compiler can generate faster code if array indices are known at compile time. Also, in superscalar architectures, the long code bursts generated by inlining can run faster than code containing loops, since the cost of failed branch predictions is avoided.

In theory, unwinding loops can result in a performance loss for processors with instruction caches. If an algorithm contains loops, the cache hit rate may be much higher for the non-specialized version the first time it is invoked. However, in practice, most algorithms for which specialization is useful are invoked many times, and hence are loaded into the cache.

The price of specialized code is that it takes longer to compile, and generates larger programs. There are less tangible costs for the developer: the explosion of a simple algorithm into scores of cryptic template classes can cause serious debugging and maintenance problems. This leads to the question: is there a better way of working with template metaprograms? The next section presents a better representation.

A condensed representation

In essence, template metaprograms encode an algorithm as a set of production rules. For example, the specialized bubble sort can be summarized by this grammar, where the template classes are non-terminal symbols, and C++ code blocks are terminal symbols:

```
IntBubbleSort<N>:
        IntBubbleSortLoop<N-1,0>  IntBubbleSort<N-1>
```

```
IntBubbleSort<1>:
        (nil)
IntBubbleSortLoop<I,J>:
        IntSwap<J,J+1> IntBubbleSortLoop<(J+2<=I) ? I : 0, (J+2<=I) ? (J+1) : 0>
IntBubbleSortLoop<0,0>:
        (nil)
IntSwap<I,J>:
        if (data[I] > data[J]) swap(data[I],data[J]);
```

Although this representation is far from elegant, it is more compact and easier to work with than the corresponding class implementations. When designing template metaprograms, it's useful to avoid thinking about details such as class declarations, and just concentrate on the underlying production rules.

Implementing control flow structures

Writing algorithms as production rules imposes annoying constraints, such as requiring the use of recursion to implement loops. Though primitive, production rules are powerful enough to implement all C++ control flow structures, with the exception of goto. This section provides some sample implementations.

If/if-else statements

An easy way to code an if-else statement is to use a template class with a bool parameter, which can be specialized for the true and false cases. If your compiler does not yet support the ANSI C++ bool type, then an integer template parameter works just as well.

C++ version	Template metaprogram version

`if (condition)`
`   statement1;`
`else`
`   statement2;`
	`// Class declarations`
	`template<bool C>`
	`class _name { };`
	
	`class _name<true> {`
	`public:`
	`    static inline void f()`
	`    { statement1; }         // true case`
	`};`
	
	`class _name<false> {`
	`public:`
	`    static inline void f()`
	`    { statement2; }         // false case`
	`};`
	
	`// Replacement for 'if/else' statement:`
	`_name<condition>::f();`

The template metaprogram version generates either statement1 or statement2, depending on whether condition is true. Note that since condition is used as a template parameter, it must be known at compile time. Arguments can be passed to the f() function as either template parameters of _name, or function arguments of f().

Switch statements

Switch statements can be implemented using specialization, if the cases are selected by the value of an integral type:

C++ version

```
int i;

switch(i)
{
    case value1:
        statement1;
        break;

    case value2:
        statement2;
        break;

    default:
        default-statement;
        break;
}
```

Template metaprogram version

```
// Class declarations
template<int I>
class _name {
public:
    static inline void f()
    { default-statement; }
};

class _name<value1> {
public:
    static inline void f()
    { statement1; }
};

class _name<value2> {
public:
    static inline void f()
    { statement2; }
};

// Replacement for switch(i) statement
_name<I>::f();
```

The template metaprogram version generates one of statement1, statement2, or default- statement depending on the value of I. Again, since I is used as a template argument, its value must be known at compile time.

Loops

Loops are implemented with template recursion. The use of the ’go’ enumerative value avoids repeating the condition if there are several template parameters:

C++ version	Template metaprogram version

`int i = N;`
 	
`do {`
`    statement;`
`} while (--i > 0);`
	`// Class declarations`
	`template<int I>`
	`class _name {`
	`private:`
	`    enum { go = (I-1) != 0 };`
	
	`public:`
	`    static inline void f()`
	`    {`
	`         statement;`
	`         _name<go ? (I-1) : 0>::f();`
	`    }`
	`};`
	
	`// Specialization provides base case for`
	`// recursion`
	`class _name<0> {`
	`public:`
	`    static inline void f()`
	`    { }`
	`};`
	
	`// Equivalent loop code`
	`_name<N>::f();`

The template metaprogram generates statement N times. Of course, the code generated by statement can vary depending on the loop index I. Similar implementations are possible for while and for loops. The conditional expression operator (?:) can be replaced by multiplication, which is trickier but more clear visually (i.e. _name<(I- 1)*go> rather than name_<go ? (I-1) : 0>).

Temporary variables

In C++, temporary variables are used to store the result of a computation which one wishes to reuse, or to simplify a complex expression by naming subexpressions. Enumerative types can be used in an analogous way for template metaprograms, although they are limited to integral values. For example, to count the number of bits set in the lowest nibble of an integer, a C++ implementation might be:

```
int countBits(int N)
{
    int bit3 = (N & 0x08) ? 1 : 0,
        bit2 = (N & 0x04) ? 1 : 0,
        bit1 = (N & 0x02) ? 1 : 0,
        bit0 = (N & 0x01) ? 1 : 0;

    return bit0+bit1+bit2+bit3;
}

int i = countBits(13);
```

Writing a template metaprogram version of this function, the argument N is passed as a template parameter, and the four temporary variables (bit0,bit1,bit2,bit3) are replaced by enumerative types:

```
template<int N>
class countBits {
    enum {
           bit3 = (N & 0x08) ? 1 : 0,
           bit2 = (N & 0x04) ? 1 : 0,
            bit1 = (N & 0x02) ? 1 : 0,
            bit0 = (N & 0x01) ? 1 : 0 };

public:
    enum { nbits = bit0+bit1+bit2+bit3 };
};

int i = countBits<13>::nbits;
```

Compile-time functions

Many C++ optimizers will use partial evaluation to simplify expressions containing known values to a single result. This makes it possible to write compile-time versions of library functions such as sin(), cos(), log(), etc. For example, to write a sine function which will be evaluated at compile time, we can use a series expansion:

sin x = x - x^3/3! + x^5/5! - x^7/7! + ...

A C implementation of this series might be

```
// Calculate sin(x) using j terms
float sine(float x, int j)
{
    float val = 1;

    for (int k = j - 1; k >= 0; --k)
        val = 1 - x*x/(2*k+2)/(2*k+3)*val;

    return x * val;
}
```

Using our production rule grammar analogy, we can design the corresponding template classes needed to evaluate sin x using J=10 terms of the series. Letting x = 2*Pi*I/N, which permits us to pass x using two integer template parameters (I and N), we can write

```
Sine<N,I>:
        x * SineSeries<N,I,10,0>
SineSeries<N,I,J,K>:
        1-x*x/(2*K+2)/(2*K+3) *  SineSeries<N*go, I*go,J*go, (K+1)*go>
SineSeries<0,0,0,0>:
        1;
```

Where go is the result of (K+1 != J). Here are the corresponding class implementations:

```
template<int N, int I>
class Sine {
public:
    static inline float sin()
    {
        return (I*2*M_PI/N) * SineSeries<N,I,10,0>::accumulate();
    }
};


// Compute J terms in the series expansion.  K is the loop variable.
template<int N, int I, int J, int K>
class SineSeries {
public:
    enum { go = (K+1 != J) };
    static inline float accumulate()
    {
        return 1-(I*2*M_PI/N)*(I*2*M_PI/N)/(2*K+2)/(2*K+3) *
            SineSeries<N*go,I*go,J*go,(K+1)*go>::accumulate();
    }
};


// Specialization to terminate loop
class SineSeries<0,0,0,0> {
public:
    static inline float accumulate()
    { return 1; }
};
```

The redeeming quality of that cryptic implementation is that a line of code such as

```
float f = Sine<32,5>::sin();
```

gets compiled into this single 80486 assembly statement by Borland C++:

```
mov     dword ptr [bp-4],large 03F7A2DDEh
```

The literal 03F7A2DDEh represents the floating point value 0.83147, which is the sine of 2*Pi*I/N. The sine function is evaluated at compile time, and the result is stored as part of the processor instruction. This is very useful in applications where sine and cosine values are needed, such as computing a Fast Fourier Transform [2]. Rather than retrieving sine and cosine values from lookup tables, they can be evaluated using compile-time functions and stored in the instruction stream. Although the implementation of an inlined FFT server is too complex to describe in detail here, it serves as an illustration that template metaprograms are not limited to simple applications such as sorting. An FFT server implemented with template metaprograms does much of its work at compile time, such as computing sine and cosine values, and determining the reordering needed for the input array. It can use template specialization to squeeze out optimization for special cases, such as replacing

```
y = a * cos(0) + b * sin(0);
```

with

```
y = a;
```

The result is a massively inlined FFT server containing no loops or function calls, which runs at roughly three times the speed of an optimized non-specialized routine.

References

[1] E. Unruh, "Prime number computation," ANSI X3J16-94-0075/ISO WG21-462.

[2] W. Press et al., Numerical Recipes in C, Cambridge University Press, 1992.

Web Site

More information on these techniques is available from the Blitz++ Project Home Page, at "http://monet.uwaterloo.ca/blitz/".

How to reach me:

tveldhui@monet.uwaterloo.ca

Todd Veldhuizen
Dept. of Systems Design Engineering
University of Waterloo, Waterloo, Ontario
Canada. N2L 3G1
Tel: (519) 885-1211 ext. 6087
Fax: (519) 746-3077

Last Revision: Oct. 3 1995