Program specialization and metaprogramming

Laboratory work No. 1

Introduction into C++ template metaprogramming

1. Aims

To get acquainted with the basics of template metaprogramming in C++.

2. Tasks

Learn how to do develop generic functions and classes using C++ template metaprogramming. Learn how to produce efficient C++ code using template specialization.

3. Work

A. 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, making the programs extremely efficient. Language features such as for loops and if statements can be replaced by template specialization and recursion techniques.

Launch Visual Studio and create C++ Console Application project.

Add the following classes to the C++ program source file:

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

template<>
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>, 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. Therefore, for example, calling `Factorial<4>::value` is equivalent to calling a function that simply returns 24.

To print the result, add the following code line into the main function:

```
std::cout << Factorial<5>::value;
```

To see the result, select Build -> Build Solution and run the program be selecting Debug -> Start Without Debugging.

What is the result?

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. 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 integer numbers:

```
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);
    }
};

template<>
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.

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);
    }
};

template<>
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.

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;
}
```

To test the sort template metaprogram, for example, add the following code to your main function:

```
    int array[4] = { 9,2,7,6 };
    IntBubbleSort<4>::sort(array);
    for (int i = 0; i < 4; i++)
       std::cout << array[i] << '\n';

    end;
```

What is the result?

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?

Template metaprograms encode an algorithm as a set of production rules. 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.

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 { };`
	
	`template<>`
	`class _name<true> {`

	public:
	    static inline void f()
	    { statement1; } // true case
	};
	
	template<>
	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	Template metaprogram version

int i;
 	
switch(i)
{
    case value1:
        statement1;
        break;
 	
    case value2:
        statement2;
        break;
 	
    default:
        default-statement;
        break;
}
	// Class declarations
	template<int I>
	class _name {
	public:
	    static inline void f()
	    { default-statement; }
	};
	
	template<>
	class _name<value1> {
	public:
	    static inline void f()
	    { statement1; }
	};
	
	template<>
	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>).

```
//generic metafunction calls (inherits from) it self
template<int IIn, int ISum = 1>
struct Factorial : Factorial<IIn - 1, IIn * ISum> {
};
//specialized metafunction has a value and does not inherit
template<int ISum> //take ISum as a wild card
struct Factorial<1, ISum> {
    enum { value = ISum };
};

    int i = Factorial<4>::value;
```

What is the result of this program?

To summarize, we instantiate a class template Factorial with a parameter of 4. The compiler instantiates Factorial<4,1> using the default value for the second parameter since we didn't give it one. In order for the compiler to instantiate Factorial<4,1> it must first instantiate its base class Factorial<3,4> which instantiates Factorial<2,12> which instantiates Factorial<1,24> which matches the specialization thus terminating recursion and initializing the member enum value to 24.

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, in a C++ template implementation, 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;
```

What is the result of this program?

B. 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:

$$\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

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 template metaprogramming, 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 the corresponding class implementations:

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

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

Note: in order to use M_PI constant, add the following code in the beginning of your program:

```
#define _USE_MATH_DEFINES
#include <math.h>
```

A line of code such as

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

gets compiled into this single floating point value of 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. Rather than retrieving sine and cosine values from lookup tables, they can be evaluated using compile-time functions and stored in the instruction stream. For example, FFT 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 input array.

C. Dot product of vectors

Implement the composite as a class template using structural information as template arguments. This structural information is the dimension of the vectors. Remember what we did for the calculation of the factorial and the square root: the former function argument in the runtime implementation became a template argument in the compile-time implementation. We will be doing a similar thing here: the vectors' dimension is passed as an argument to the evaluation function in the object-oriented approach; we will pass it as a template argument in the compile-time implementation. Hence the dimension of the vectors will be become a non-type template argument of the composite class.

```
template <size_t N, class T>
class DotProduct  {
public:
  static T eval(T* a, T* b)
  { return  DotProduct<1,T>::eval(a,b)
          + DotProduct<N-1,T>::eval(a+1,b+1);
  }
```

```
};
template <class T>
class DotProduct<1,T> {
public:
  static T eval(T* a, T* b)
  { return (*a)*(*b); }
};
```

The leaf will be implemented as a specialization of the composite class template for dimension N = 1. As before replace the runtime recursion by a compile-time recursion: replace the recursive invocation of the virtual evaluation function by recursive template instantiation of a static evaluation function.

```
template <size_t N, class T>
inline T dot(T* a, T* b)
{
        return DotProduct<N,T>::eval(a,b);
}
```

Test the program by adding the following code to the main() function:

```
int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
cout << dot<4>(a,b);
```

What is the result of program execution?

4. Challenge

Perform time measurements, what is the execution time of template metaprogram versions of code vs implementations using c++ with different template parameters. Use clock() function or other to measure CPU time.

Present and discuss the results in your report.

5. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?