Federal University of Santa Catarina
Real-Time Operating Systems

Static Metaprogramming in C++

Prof. Dr. Giovani Gracioli
giovani@lisha.ufsc.br
http://www.lisha.ufsc.br/Giovani

What is Static Metaprogramming? (1)

- In linguistics, a metalanguage is defined as follow
  - “any language or symbolic system used to discuss, describe, or analyze another language or symbolic system” [Czarnecki]
- A program that manipulates another program is clearly an instance of metaprogramming
- We focus on generative metaprograms
  - programs manipulating and generating other program
- Generative metaprogram = algorithm + program representation

What is Static Metaprogramming? (2)

- Static metaprograms
  - These metaprograms run before the load time of the code they manipulate
- Examples
  - Compilers and pre-processors
  - Manipulate representations of input programs in order to transform them into other languages

Template Metaprogramming

- A practicable approach to metaprogramming in C++ is template metaprogramming
- Template metaprograms consist of class templates operating on numbers and/or types as data
- Algorithms are expressed using template recursion as a looping construct and class template specialization as a conditional construct

Example: factorial (1)

```
template<int n> struct Factorial
{
      enum { RET =
      Factorial<n-1>::RET * n };
};

//the following template
//specialization terminates the
//recursion
template<> struct Factorial<0>
 {
      enum { RET = 1 };
};

void main() {
cout << Factorial<7>::RET << endl; //prints 5040
}
```

- Factorial<7> is instanciated at compile time
  - It determines the valoe of Factorial<7>::RET
- The generated code inside main is equal to: cout << 5040 << endl;

Example: factorial (2)

- We can regard Factorial<> as a function which is evaluated at compile time
- This particular function takes one number as its parameter and returns another in its RET member (RET is an abbreviation for RETURN; we use this name to mimic the return statement in a programming language)
- By encoding data as types, we can actually use (or abuse) the compiler as a processor for interpreting metaprograms
- Factorial<> is a metafunction since, at compilation time, it computes constant data of a program which has not been generated yet

Example: generic types

```
int sum (int a, int b)
{
  return a+b;
}

double sum (double a, double b)
{
  return a+b;
}

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

```
template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

Non-type template arguments

- The template parameters can not only include types introduced by class or typename, but can also include expressions of a particular type:

```
template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

Metafunction: if statement (1)

- Another example of a metafunction would be a function which returns a type, especially a class type
- Since a class type can represent computation, such a metafunction actually manipulates representations of computation

Metafunction: if statement (2)

```
template<bool cond, class ThenType, class ElseType>
struct IF {
      typedef ThenType RET;
}

template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType> {
      typedef ElseType RET;
};
```

- The if metafunction takes a boolean and two types as parameter and returns a type
- If statement: it has a condition parameter, a “then” parameter, and an “else” parameter
- If the condition is true, it returns ThenType in RET
- If the condition is false, it returns ElseType in RET. Thus, this metafunction can be viewed as a meta-control statement

Using the meta IF

- Meta IF can be used to define a type according to a condition
- Example

  typedef IF<variable, type1, type2>::RET type;

- If variable is true, type is defined as type1
- If variable is false, type is defined as type2

Trait class

- Templates providing information about other types are referred to as traits templates
- A base Trait class and several specializations
- More: http://www.cantrip.org/traits.html

```
template <class Imp> struct Traits {
      static const bool enabled = false;
};

template <> struct Traits<Component> {
      static const bool enabled = true;
};

void Component::method() {
      if(Traits<Component>::enabled) {
            ......
      }
}
```

Metaprograms and Code Generation (1)

- So far, we used templates to perform computations at compile time
- We can also arrange metafunctions into metaprograms that generate code
- Example: each type define the static inline method print()
- Consider the following statement:
  - IF< (1<2), Type1, Type2>::RET::print();
- What is the output?

```
struct Type1 {
    static void print() {
    cout << "Type1" << endl;
    }
};
struct Type2 {
    static void print() {
    cout << "Type2" << endl;
    }
};
```

Metaprograms and Code Generation (2)

- The previous statement compiles into machine code which is equivalent to the machine code obtained by compiling the following statements:
  - cout << "Type1" << endl;
- The reason is that print() is declared as a static inline method and the compiler can optimize away any overhead associated with the method call
- This was just a very simple example, but in general you can imagine a metafunction that takes some parameters and does arbitrary complex computation in order to select some type providing the right method:

```
SomeMetafunction</* takes some parameters here*/>::RET::executeSomeCode();
```

Example: loop unrolling

```
class Test {
public:
    template <int i>
    class LOOP {
    public:
        static inline void EXEC() {
            cout << “i = “ << i << endl;
            LOOP<i - 1>::EXEC();
        }
    };
     Test() {
        LOOP<10>::EXEC();
     }
};
template <>
inline void Test::LOOP<0>::EXEC() { }
```

```
int main(void)
{
    Test t;
    return 0;
}
```

Meta List of Types

- Available at moodle.ufsc.br
- How to use:

```
template <> struct Traits<Sensor_Manager>
{
    typedef LIST<Class1, Class2> LIST;
};

class Sensor_Manager
{
public:
    typedef Traits<Sensor_Manager>::LIST LIST;
    template <int i> class LOOP {
        static inline void EXEC() {
            //LIST::Get<i-1>::Result
        }
     };
};
```

Metaprogramming Limitations

Complexity limit	**The complexity of metaprograms is limited by the limits of current C++ compilers in handling very deeply nested templates**
Debugging support	**There is no debugging support**
Error report	**Inadequate**
Programming effort	**Due to the lack of debugging support, error-prone syntax (e.g. lots of angle brackets), and current compiler limits, template metaprograms may require significant programming effort. On the other hand, the code to be generated can be represented as easy-to-configure class templates**
Readbility of meta code	**Low**
Compilation speed	**Larger metaprograms (with recursion) may have unacceptable compilation times. Template metaprograms are interpreted.**
Portability/availability	**Potentially wide available, but better support for the C++ standard is required. The same applies to portability.**
Performance of generated code	**Comparable to manually written code. The complexity of optimizations is limited by the complexity limits of template metaprograms.**

Conclusion

- We can view C++ as a two-level language. Two-level languages contain static code (which is evaluated at compile-time) and dynamic code (which is compiled, and later executed at run-time)
- Static metaprogramming adds configurability to the software, which is very interesting mainly for embedded systems

Homework 2: generic list

- Available at moodle.ufsc.br
- Implementation of a singly linked list using templates (Simple_List)
- TODO: implement the methods of a Simple_Ordered_List, which extends Simple_List

References

- Generative Programming - Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Krzysztof Czarnecki. PhD Thesis. 1998
- http://www.cantrip.org/traits.html
- http://www.cs.ucdavis.edu/~devanbu/teaching/260/meta.pdf