Generic Programming with Generic Components

System construction from generic components using templates

Literature:
Especially, Chapters 7, 9 and 10 on Static Metaprogramming in C++.

Recall: Static vs. Dynamic Metaprogramming

<table>
<thead>
<tr>
<th>Metaprogramming</th>
<th>Static metaprogramming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspect/configure/modify the (own) code</td>
<td>Configuration of program components, done at compile time</td>
</tr>
<tr>
<td></td>
<td>Evaluated by the compiler before code generation (no run-time overhead)</td>
</tr>
<tr>
<td></td>
<td>Based on simple macro expansion (C preprocessor) or grammar extensions (BETA) or type expressions (C++ templates) or hook binding (COMPOST) ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic metaprogramming</th>
<th>Evaluated at run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>By the run-time system (--&gt; run-time overhead)</td>
<td>e.g., Java using the Java Reflection API, CORBA DII, ...</td>
</tr>
</tbody>
</table>

Generic Components

- A **generic component** is a template from which other components can be generated
- A **generic class** is a special case, in which types are parametric.
- However, all language constructs could be parametric
  - If the language has a metamodel, by which the parameters are typed.
  - Example: COMPOST
- Generic components rely on **bind** operations that bind the template parameter with a **value** (parameterization)
  - The result is called the **extent**

Binding Templates with Static Values

Static Value

- e.g. a constant, type name ...

Static Value

Static Value

extent (woven code after binding)

Generic Algorithm

Static Value
Generic Components (Templates)
Bind at Compile Time

New Fragment Component

Value
(Fragment)

Generic Fragment
(Template)

Value
(Generic
Fragment)


Templates in C++ (1)

- **Templates with type parameters:**
  int, short, char, ...

- **Specializations override defaults**

```cpp
template<class T>
void swap(T &a, T &b)
{  T c = a;
   a = b;
   b = c;
}
```

```cpp
template<class T, int size>
class Vector
{   T data[size];
    // ...
};
```

```cpp
template<class T>
void swap(Vector<T>& a, Vector<T>& b)
{  a.swap(b);
}
```

Templates in C++ (2)

- **Parameterized inheritance:**

```cpp
template<class Superclass>
class SomeClass : public Superclass
{   //...
}
```

```cpp
template Vector<Node, 10, 10> MyMatrix;
```

**Definition:**

- **Templates** in C++ allow parameterization of expressions over types and compile-time constants, evaluated at compile time (similar to macro expansion).
- **Template Metaprogramming**

  - **Disadvantage:** Leads to unreadable programs, since the template concept is being over-used.
  - **Advantage:** Uses standard tooling.

Template Metaprogramming

- **Template Metaprogramming** [Czarnecki/Eisenecker]

- C++ has **templates**
  - i.e., parameterized expressions over types and compile-time constants,
  - evaluated at compile time (similar to macro expansion)
  - declarative rather than imperative

- Disadvantage: Leads to unreadable programs, since the template concept is being over-used

- Advantage: Uses standard tooling
Representing Metainformation with Traits

- Store metainformation for (dynamic) objects in type variables, using templates
- Such type characteristics are called traits.

**Example:** For a statically allocated vector, advertise its size and element type

```cpp
template <class ElementType_, int size_> class Vector {

  public:
    // traits class
    struct Config {
      typedef ElementType_ ElementType;
      enum { size = size_ };  // type characteristics
    }

  private:
    ElementType_ elems [ size_ ];

  //...
};
```

Query size by

```cpp```
Vector :: Config :: size
```

**Useful:** Allows to retrieve element type information (metainformation about parameter type) at runtime (i.e., reflection), which is otherwise not possible (static information is removed).

Static Metaprogramming in C++ (1)

**Example:** recursive factorial function

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

**Evaluation at compile time** possible for statically known n:

```cpp```
cout << "factorial(7)= " << factorial(7) << endl;
```

Static Metaprogramming in C++ (2)

**Example:** recursive factorial function

```cpp```
cout << "factorial(7)= " << factorial(7) << endl;
```

Use of the name RET mimics the effect of a return statement of a conventional function.

```cpp```
Factorial<.> is a template metafunction.
```

Static Metaprogramming in C++ (3)

**Static if:**

```cpp```
```
// specialization for condition==false:
template<class Then, class Else>
struct IF<false, Then, Else>
{ typedef Else RET;  // the type of i is int!
  //...
};
```

Static if:
Static Metaprogramming in C++ (4)

- **Compile-time Lists**

In Lisp: `(cons 1 (cons 2 (cons 3 (cons 9 nil))))` creates list `(1 2 3 9)`
Simulate this with C++ templates:
```
Cons<1, Cons<2, Cons<3, Cons<9, End> >>
```

// tag marking the end of a list:
const int endValue = ~(~0u >> 1);  // initialize with the smallest int

struct End
{
    enum { head = endValue }
    typedef End Tail;
};

template <int head_, class Tail_ = End>
struct Cons
{
    enum { head = head_ }
    typedef Tail_ Tail;
};

Static Metaprogramming in C++ (5)

- Expansion at compile time (in the compiler's intermediate representation):
```
typedef Cons<1, Cons<2, End> > list1;

typedef struct Cons
{
    enum { head = 1 }
    typedef struct Cons
    {
        enum { head = 2 }
        typedef End Tail;
    } Tail;
} list1;
```

Compute statically the length of a compile-time list:
```
template <class List>
struct Length
{
    enum { Ret = Length <typename List::Tail> :: Ret + 1 };
};

// stop the recursion if we arrived at End:
template <>
struct Length <End>
{
    enum { RET = 0 };
};

typedef Cons<1, Cons<2, End> > list1;

cout << Length < list1 > :: Ret << endl;
```

Static Metaprogramming in C++ (6)

- Expansion at compile time (in the compiler's intermediate representation):
```
Example: cout << Length<list1>::RET << endl;  // prints 2
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET
Length < Cons<1, Cons<2, End >>> :: Ret
Length < End > :: RET + 1
```

```
struct Length
{
    enum { RET = Length < Cons<2, End > :: Ret + 1 };
};
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```

```
Length < list1 > :: RET
Length < Cons<1, Cons<2, End >>> :: RET + 1
```
### Static Metaprogramming in C++ (8)

```cpp
// Implementation of a statically evaluated SWITCH as struct type
// `tag`: parameter type that characterizes the case tag
// `aCase`: parameter type that characterizes the case block
struct SWITCH {
  template <int Tag, class aCase>
  struct SWITCH {
    typedef aCase::next nextCase;
    enum {
      tag = aCase::tag,
      nextTag = nextCase::tag
    }
    typedef IF<(found != 0),
      aCase::statement,
      nextSwitch::RET> ::RET;
    typedef IF<(tag == Tag || tag == DEFAULT), // Tag test, is statically evaluated
      nextSwitch::RET,                        ::RET
      nextSwitch;                        ::RET;
  };
};

// Example using the static SWITCH construct:
switch ( 3*a+1 ) {
  case 2: pr(a); break;
  case 4: exit();
  case 6: foo(b); break;
  default: bar(); break;
}
```

### Static Metaprogramming in C++ (9)

- Example using the static SWITCH construct:

  ```cpp
  struct A { static void execute() { cout << "A" << endl; });
  struct B { static void execute() { cout << "B" << endl; };
  struct D { static void execute() { cout << "Default" << endl; };
  
  // ...SWITCH < (1 + 1 – 2 ),
  CASE < 1, A,
  CASE < 2, B,
  CASE < DEFAULT, D > >> :: RET :: execute();                      // prints "Default"
  ```

### C++ templates as Composition System

- **Component model**
  - Source code components
  - Generic components
  - Composition interfaces with declared slots

- **Composition technique**
  - Composition operator: bind (parameterize)

- **Composition language**
  - Simple combination of the composition operators
Other Approaches to Generic Programming:
BETA, GenVoca

- [BETA-DEF]
  Object-Oriented Programming in the BETA Programming Language.
  Addison-Wesley, 1993.
  Freely available on the web: http://www.daimi.au.dk/~beta/

- [BETA-ENV]
  Great book on BETA and its environment.

- Ole Lehmann Madsen. The Mjölnir BETA fragment system. In [BETA-ENV].

  Fourth Int. Conference on Software Reuse, April 23-26, 1996, Orlando Florida.
  IEEE Computer Society Press, pages 166-175

- Ole Lehrmann Madsen. The Mjölnir BETA fragment system. In [BETA-ENV].


Java Generics

- Since JDK 1.5 (2004)
  - Based on Pizza [M. Odersky, 1997]
- Simulate part of C++ Templates functionality (generic types and methods)
  on top of existing JVM
- Generic types are NOT expanded at compile time
  but resolved at run time
- List<Integer> is compiled to List of Object in the JVM bytecode;
  Compiler inserts type casts automatically at accesses
  (dynamic downcasts – runtime checking overhead!)
- There is only one version of the code for List.
- The type parameter is essentially thrown away after compilation
  and NOT stored in the metadata.
  - getClass() only gives List, not List<Integer>

+ Java Generics are type-safe.
+ No extension of JVM needed
- Inefficient compared to C++ Templates

C# Generics

- C# Generics are also resolved at runtime (load time), as with Java Generics
- But C# compiler + IL/CLR keep the information of the type parameter
  + Reflection works
    Yields List<Integer> in the above example
  + No dynamic downcast necessary

Differences between C++ Templates and Java / C# Generics