Design Patterns in C++

Template metaprogramming

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

April 6, 2011

Outline

1. C++ Template meta-programming
2. Techniques
3. Detecting inheritance relationship
4. Lists of types

Templates as code generators

- A template is a way to generate code. The following program generates three functions

```
template<typename T>
void swap(T& a, T&b) {
    T t = a;
    a = b;
    b = t;
}
int main() {
    char z = 1, w = 2;
    int x = 3, y = 4;
    double r = 5, s = 6;

    swap(z, w);
    swap(z, y);
    swap(r, s);
}
```

Templates as code generators

- A template is a way to generate code. The following program generates three functions

```
template<typename T>
void swap(T& a, T&b) {
    T t = a;
    a = b;
    b = t;
}
int main() {
    char z = 1, w = 2;
    int x = 3, y = 4;
    double r = 5, s = 6;

    swap(z, w);
    swap(z, y);
    swap(r, s);
}
```

```
void swap(char &a, char &b) {
    char t = a;
    a = b;
    b = t;
}
void swap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
}
void swap(double &a, double &b) {
    double t = a;
    a = b;
    b = t;
}
```

Template members

- It is also possible to make one member function a template

```
class MyClass {
    ...
public:
    template<class V>
    void accept(V &v) {
        v.visit(*this);
    }
};
```

- The member function is not compiled if it is not used
- Important restriction: cannot make template virtual methods

Incomplete instantiation

- Remember that template classes are not generated if not used.
- This is also valid for members of template classes

```
template <class T>
class MyClass {
    T *obj;
public:
    MyClass(T *p) : obj(p) {}

    void call_fun() {
        obj->fun();
    }

    T* getPtr() { return ptr; }
};

MyClass<int> my(new int(6));
cout << my.getPtr() << endl;
```

- In the code above, function `call_fun()` is never called, so the code is correct (it would give error on a `int` type!)

Template template parameters

- It is possible to require a parameter to be a template:

```
template <template <class Created> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget> {
    ...
public:
    void fun() {
        Widget *w = create();
    }
};

template <class T>
class DynamicCreation {
public:
    T* create() { return new T; }
};
```

Template template parameters

- template parameter `Created` can be entirely omitted

```
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget> {
    ...
public:
    void fun() {
        Widget *w = create();
    }
};
```

Specialization

- Another important feature is template specialization
- Given a template with one or more parameters:
  - we can provide special versions of the template where some of the parameter is assigned specific types

```
// general version
template<class T> class MyClass {
   ...
};

// special version with T = int
template <> class MyClass<int> {
};

MyClass<double> m;
// uses the general version

MyClass<int> n;
// uses the special version
```

Partial specialization

- When we have more than one type parameter, we can specialize a subset of the types

```
// general version
template<typename T, typename U> class A { ... };

// specialization on T
template<typename U> class A<int, U> { ... };

// specialization on both
template<> class A<int, int> { ... };

A<double, double> a; // uses general version
A<int, double> b;    // uses first specialization
A<int, int> a;       // uses second specialization
```

Problems

- Be careful with partial template specialization

```
// general version
template<typename T, typename U> class A { ... };

// specialization on T
template<typename U> class A<int, U> { ... };

// specialization on U
template<typename T> class A<T, int> { ... };

A<int, int> a;        // Error! ambiguos
```

Restrictions on partial specialization

- Partial template specialization does not apply to functions

```
template <class T, class U> T fun(U obj) {...}

// the following is illegal
template <class U> void fun<void, U>(U obj) {...}

// the following is legal (overloading)
template <class T> T fun(Window obj) {...}
```

Integral template parameters

- The parameters of a template can be constant numbers instead of types
  - By constant I mean that they are known at compile time

```
template <unsigned n>
struct Num {
    static const unsigned num = n;
};

Num<5> x;
Num<8> y;

cout << x.num << endl;
cout << y.num << endl;
cout << "sizeof(Num<5>) = " << sizeof(x) << endl;
```

- **Note 1**: the output of last instruction is 1

Integral template parameters

- The parameters of a template can be constant numbers instead of types
  - By constant I mean that they are known at compile time

```
template <unsigned n>
struct Num {
    static const unsigned num = n;
};

Num<5> x;
Num<8> y;

cout << x.num << endl;
cout << y.num << endl;
cout << "sizeof(Num<5>) = " << sizeof(x) << endl;
```

- **Note 1**: the output of last instruction is 1
- **Note 2**: we can also make calculation with n, but of course they must be solved at run-time

Specialization on numbers

- We can also specialize on certain numbers

```
template <unsigned n>
struct VarNum {
    void p() { cout << "Number is " << n << endl; }
};

template <>
struct VarNum<0> {
    void p() { cout << "N is zero!!" << endl; }
};
```

Factorial

- The “code generation engine” that is embedded in the C++ compiler can actually make some simple calculation, using recursion
- The key trick is combine specialization with integral parameters
  - This is calculation made by the compiler, not at run-time!
- Here is how to compute the factorial

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

template <>
struct Fact<1> {
    enum { value = 1 };
};

cout << Fact<5>::value << endl;
```

- See the code

Very simple types

- C++ does not require much for defining a type
- The following ones are all different types

```
struct Simple {};
struct AnotherSimple {};
struct YetAnotherSimple {};
```

- Of course, you cannot do much with such types, except telling the compiler that a certain type exist
- at run time there is no code associated with it
- `sizeof(Simple)` returns `1`

Defining types

- Look at this code

```
template<typename T>
struct Cont {
    typedef T MyType;
};

Cont<int>::MyType anInteger = 2;
Cont<double>::MyType aDouble = 0.5;
```

- `Cont` does not contain anything, only the definition of another type, which is only used by the compiler, but not at run-time
- `Cont<int>` is not different from Simple

Outline

1. C++ Template meta-programming
2. Techniques
3. Detecting inheritance relationship
4. Lists of types

Mapping integers to types

- It is sometimes necessary generate new types from integers. Here is how to do it:

```
template<int v>
struct Int2Type
{
    enum {value = v};
};
```

- This template generates a new type for each integral number it is passed
- we will use this in the following to do a “compile-time dispatch”
  - You want to select one of several functions to be called, and you want to do this selection at compile time, depending on a compile-time constant

Run-Time vs. Compile time dispatching

- Run-time dispatching is something that depends on the values of certain variables that are only known at run-time
  - It can be performed by using `if-then-else` or `switch-case` statements
  - It can also be performed using virtual functions and dynamic binding
  - the cost of doing this is often negligible
- of course, you can also do run-time dispatching based on compile-time constants
  - however this is not always possible

Selection example

- Suppose you are designing a container

```
template<class T> MyContainer { ... };
```

- You want to provide the ability to copy objects of type `MyContainer` by copying all contained objects
- to duplicated contained objects, you can call the copy constructor, but this may not be always available
  - Suppose instead that class `T` provides a virtual `clone()` method
- you would like your container to be generic, i.e. a container that can be used both for objects with copy constructors (non-polymorphic), and objects with clone (polymorphic)
- therefore, the choice of calling one of the two functions is based on a compile-time boolean constant `isPolymorphic`

Run-time selection?

- The following code does not work:

```
template <class T, bool isPolymorphic>
class MyContainer {
    ...
    void function() {
        T *p = ...;
        if (isPolymorphic) {
            T *p2 = p->clone();
            ...
        } else {
            T *p2 = new T(*p);
            ...
        }
    }
};
```

- why?

Compile-time dispatch

- Solution:

```
template<class T, bool isPolymorphic>
class MyContainer {
private:
    void function(T* p, Int2Type<true>) {
        T *p2 = p->clone();
        ...
    }
    void function(T *p, Int2Type<false>) {
        T *p2 = new T(*p);
        ...
    }
public:
    void function(T *p) {
        function(p, Int2Type<isPolymorphic>());
    }
};
```

- The basic trick is that only one between the two private `function` will be compiled, the right one!

Type to Type

- Unfortunately, it is not possible to apply partial template specialization to functions
  - You can simulate this with overload
- example:

```
// creates a class by invoking the constructor,
// which takes one argument
template<class T, class U>
T *create(const U &arg) {
    return new T(arg);
}
```

- Now, suppose you want to specialize it by using the same function on objects of class Widget, whose constructor takes two arguments, the second one must always be -1

Special version

- Wrong:

```
template<class T, class U>
T *create(const U &arg) {
    return new T(arg);
}

template<class U>
Widget *create<Widget, U>(const U &arg) {
    return new Widget(arg, -1);
}
```

Special version

- Wrong:

```
template<class T, class U>
T *create(const U &arg) {
    return new T(arg);
}

template<class U>
Widget *create<Widget, U>(const U &arg) {
    return new Widget(arg, -1);
}
```

- Correct:

```
template<class T, class U>
T *create(const U &arg, T) {
    return new T(arg);
}

template<class U>
Widget *create(const U &arg, Widget) {
    return new Widget(arg, -1);
}
```

Reducing overhead

- To eliminate the overhead of creating a dummy parameter to pass by value, we can use the following “trick”:

```
template<typename T>
struct Type2Type {
    typedef T OriginalType;
};

template<class T, class U>
T *create(const U &arg, Type2Type<T>) {
    return new T(arg);
}

template<class U>
Widget *create(const U &arg, Type2Type<Widget>) {
    return new Widget(arg, -1);
}

string *ps = create("Hello", Type2Type<string>());
Widget *pw = create("Hello", Type2Type<Widget>());
```

Selecting types

- Suppose that in `MyContainer` we want to store objects if the type is non-polymorphic, and pointers to objects if the type is polymorphic
- for simplicity, suppose that internally we use a vector
- what type should we give to the vector? `T` or `T*`?
- We can make use of a “select” structure

```
template <bool flag, typename T, typename U>
struct Select {
    typedef T Result;
};
template<typename T, typename U>
struct Select<false, T, U> {
    typedef U Result;
};
```

Selecting the type

```
template <typename T, bool isPolymorphic>
class MyContainer {
    typedef typename Select<isPolymorphic, T*, T>::Result ValueType;
    ...
    std::vector<ValueType> objects;
    ...
};
```

Outline

1. C++ Template meta-programming
2. Techniques
3. Detecting inheritance relationship
4. Lists of types

How to detect type relationship

- Sometimes it is important to be able to understand if two generic types T and U are in a relationship of inheritance
  - for example, we would like to know if U derives from T
  - this may be useful sometimes when we have to check and enforce some properties on types in our generic algorithms at compile time
  - also, some algorithms may be optimized if an inheritance relationship exists
- inheritance is a special case of convertibility between pointers
- there are only three cases in which we can automatically convert `U*` to `T*`;
  - `T` and `U` are the same case
  - `T` is a public base class of `U`
  - `T` is `void` (because any pointer can be converted to `void *`)
- So, now the problem is how to detect if an automatic conversion exists at compile time

Using sizeof

- `sizeof()` is a compile time mechanism of C/C++ to compute the size of a type
- since it is done at compile time, its argument is not actually executed at run-time, but it is only evaluated
  - For example, if the argument contains a function call, it does not actually invoke the function, but it only checks its type

```
int returnInt(); // not implemented
sizeof(returnInt()); // equivalent to sizeof(int)
```

- `sizeof()` returns an integer that we can use in comparisons, so let's define two types of different size

```
typedef char Small;
class Big { char dummy[2]; };
```

Conversion

```
template <typename T, Typename U>
class Conversion {
    typedef char Small;
    class Big { char dummy[2]; };
    static Small Test(U);
    static Big Test(...);
    static T MakeT();
public:
    enum { exists = sizeof(Test(MakeT())) == sizeof(Small) };
};
```

- We try pass an object of type `T` to a function that takes a `U` or anything else;
  - If the conversion exist, the first version is used which returns a `Small`

Last step

- We can define (also in the previous code) an additional enum value `sameType` and provide the following specialization

```
template<typename T>
class Conversion {
public:
    enum { exists = true, sameType = true; }
};

#define SUPERSUBCLASS(T, U) \
    (Conversion<const U*, const T*>::exists && \
     !Conversion<const T*, void *>::sameType)

#define SUPERSUBCLASS_STRICT(T, U) \
   (SUPERSUBCLASS(T, U) && \
    !Conversion<const U*, const T*>::sameType)
```

Outline

1. C++ Template meta-programming
2. Techniques
3. Detecting inheritance relationship
4. Lists of types

TypeList definition

- A type list is a simple list of any number of types (from 1 to any)
- At compile time the user instantiates the typelist with the desired types
- however, C++ does not support a variable number of template arguments
- Therefore, we have to simulate it
- the basic structure is very simple:

```
struct NullType {};

template<typename H, typename T>
struct TypeList {
    typedef H Head;
    typedef T Tail;
};
```

Defining typelists

- A few examples:

```
// two types, int and double
TypeList
<
  int,
  TypeList<double, NullType>
> intDoubleVar;

// three types, int, double and char
TypeList
<
    int,
    TypeList
    <
        double,
        TypeList<char, NullType>
    >
> intDoubleCharVar;
```

Obtaining types

- It is annoying to carry on a list in expanded form
- We can use defines to simplify the writing a little bit

```
#define TYPELIST_1(t1)          TypeList<t1, NullType>
#define TYPELIST_2(t1, t2)      TypeList<t1, TYPELIST_1(t2) >
#define TYPELIST_3(t1, t2, t3) TypeList<t1, TYPELIST_2(t2, t3) >
...

TYPELIST_3(int, double, char) intDoubleCharVar;
```

- Remember: the size of variable `intDoubleCharVar` is ...

Obtaining types

- It is annoying to carry on a list in expanded form
- We can use defines to simplify the writing a little bit

```
#define TYPELIST_1(t1)          TypeList<t1, NullType>
#define TYPELIST_2(t1, t2)      TypeList<t1, TYPELIST_1(t2) >
#define TYPELIST_3(t1, t2, t3) TypeList<t1, TYPELIST_2(t2, t3) >
...

TYPELIST_3(int, double, char) intDoubleCharVar;
```

- Remember: the size of variable `intDoubleCharVar` is ...1!!
- so, what is the purpose of this?
  - a typelist is not a “useful” class, because it does not contain code
  - it is only useful to help the compiler to generate code
  - it is a trick to pass “types” instead of “values”

How to use it

- A type list can be used to obtain the various types in the list

```
typedef TYPELIST_3(int, double, char) ThreeTypes;

ThreeTypes::Head              param1; // this will be an int
ThreeTypes::Tail::Head        param2; // this will be a double
ThreeTypes::Tail::Tail::Head param3; // this will be a char
```

- It is quite ugly, but it works: I now defined three variables with the right type
- Almost there, we are missing a few steps

Example

- Suppose I need to write the code for a structure that must contain three data members
- however, I do not know “a-priori” the type of these members
- I would like to write generic, template code that instantiates the structure with the right types

```
template <typename T>
struct MyStruct {
    T::Head              param1;
    T::Tail::Head        param2;
    T::Tail::Tail::Head param3;
};

MyStruct< TYPELIST_3(int, double, char) > s;
s.param1 = 5;
s.param2 = 0.75;
s.param3 = 'a';
```

How to simplify type extraction

```
template<typename H, typename T, unsigned index>
struct TypeAt {
    typedef typename TypeAt_<typename T::Head,
                             typename T::Tail,
                             index-1>::Result Result;
};

// specialization for index == 0
template <typename H, typename T>
struct TypeAt<H, T, 0> {
    typedef H Result;
};

// if it reaches the end of the list with index > 0...
template <typename H, unsigned index>
struct TypeAt<H, NullType, index> {
    typedef NullType Result;
};

// special case: the last element
template <typename H>
struct TypeAt<H, NullType, 0> {
    typedef H Result;
};

#define TYPEAT(L, i) typename TypeAt<typename L::Head, \
                                     typename L::Tail, \
                                     i                 \
                                     >::Result
```

How to use it

- Quite difficult to understand
  - However, once it works, it's already there, you do not need to understand how it works every time
  - you only need to know how to use it

```
template<typename L>
struct MyStruct {
    TYPEAT(L, 0) param1;
    TYPEAT(L, 1) param2;
    TYPEAT(L, 2) param3;
};

...
MyStruct< TYPELIST_3(int, double, char) > s;
s.param1 = 1;
s.param2 = 0.75;
s.param3 = 'c';
```

Count elements in the typelist

```
template<typename H, typename T>
struct TypeListSize {
   enum { Result = TypeListSize<T::Head,
                                T::Tail,
                                index-1>::Result + 1 };
};

template<typename H>
struct TypeListSize<H, NullType> {
   enum { Result = 1 };
};

template<typename T>
struct TypeListSize<NullType, T> {
   enum { Result = 0 };
};

#define TYPELISTSIZE(L) TypeListSize<L::Head, L::Tail>::Result
```