C++ Templates

Rémi Delcos

March 29, 2004

1 Introduction

The origin of templates in C++ goes back to the need to build a standard library. A major issue with the languages was the lack of a general facility to produce container classes. Two different approaches for providing such classes were considered [1]:

1. The Smalltalk way of relying on dynamic typing and inheritance (notably having a single root class hierarchy).

2. The Clu approach of using static typing and a facility for arguments of type *type*.

The goal was to define a mechanism that is type-safe, has close to ideal run-time and space requirements and has a convenient notation. Portability and reasonably efficient compilation and linkage were the main constraints. Building on the Clu approach, the resulting C++ templates mechanism not only solved the container problem nicely but opened the door to a wealth of new programming techniques.

2 Templates

Templates are classes or functions that have been written with one or more types yet not defined [2]. When the template is used, the missing types are passed, explicitly or implicitly, as *template parameters*. A template with a given set of parameters is a *specialization*. When a specialization is used, for example as the type in a variable declaration, it is *instantiated*. Templates are the basis for *generic programming* or *programming with types*.

2.1 Class Template Definition

A class template's declaration begins with the keyword `template` followed by one or more arguments listed in angle brackets. The definition is similar to the definition of a regular class except that the undefined types are represented by the template's formal parameters. A template class can be part of an inheritance hierarchy. Here is the class template definition for a stack:

```
template<typename T> class stack {
public:
  //...
  void push(T item) {array[top++] = item;}
  T    pop() {return array[--top];}
};
```

2.2 Template Instantiation

When a template is used, and all its parameters are provided, it can be instantiated. The compiler then generates a new class, indistinguishable from a hand written class. Instantiation is usually done automatically by the language implementation. The declaration of a pointer or reference to a template class doesn't cause instantiation.

```
stack<int>   s;   // implicit specialization for int instantiated
stack<long>* sp;  // no instantiation
```

Errors that are not related to template arguments are checked at the point of definition. Error related to arguments can be detected at the earliest at the point of use also called the *point of (first) instantiation*. However, the language implementation is allowed to postpone error checking until the program is linked. As only the member functions that are actually used are instantiated, a template can contain member functions that are legal only with some template parameters.

Because each different set of template arguments causes a new instantiation, using templates can lead to code replication or "code bloat". Several techniques are available to the programmer for curbing code bloat, including factoring out code that is common to all template instantiations. In addition language implementations can coalesce template instantiations that might share the same code (for example `stack<int*>` and `stack<char*>`).

2.3 Template Parameters

Template parameters can be type parameters, parameters of ordinary types or template template parameters. Type parameters are preceded by either `typename` or `class`. Both keywords have the same meaning in this context. Ordinary type parameters must be compile time constants. Like function parameters, template parameters are allowed default values.

```
template<typename T=int, int size=10> class stack {
  //...
};

stack<> s1;          // stack<int, 10>
stack<long> s2;      // stack<long, 10>
stack<long, 20> s3; // stack<long, 20>
```

Here is an example of template template parameter usage:

```
template<typename T> class point {
  //...
};

template <template <typename T> class P> class rect {
  P<short> top_left;
  P<short> bottom_right;
  //...
};

rect<point> r;  // instantiation
```

2.4 Explicit Specialization

If a template's definition is not suitable or optimal for certain parameters, it is possible to redefine the template for specific types. This is called *user-defined specialization* or *explicit specialization*. A template can also be *partially specialized* i.e. only part of the parameters are fixed. A partially specialized template remains a template and needs the remaining arguments for being instantiated. Explicit specializations must be declared before their first use.

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

// explicitly specialized for int and void*
template<> class A<int, void*> {
  //...
};

// partially specialized for int
template<typename U> class A<int, U> {
  //...
};

// partially specialized for int and any pointer type
template<typename U> class A<int, U*> {
  //...
};
```

2.5 Function Templates

In addition to classes, also free functions and class member functions can be turned into templates. For example, a generic `max()` function can be defined as follows:

```
template<typename T> T max(T arg1, T arg2) {
  return arg1 > arg2 ? arg1 : arg2;
}
```

Contrary to class templates, function templates can deduce type and non-type arguments from a call. Types can be stated explicitly if the compiler does not instantiate the proper function using the types from type deduction.

```
int i1 = 1;
int i2 = 2;
long l = 3;
int result = max(i1, i2);    // max<int> instantiated
result = max(i1, l);         // error, no max(int, long)
result = max<long>(i1, l);   // ok, max(long, long)
```

Instantiated template function are subject to the same rules as ordinary functions. Especially they can participate with ordinary functions to overloading resolution.

2.6 Source Code Organization

Template declarations and definitions are almost always stored in header files. These headers are included in the source files that use the templates. Because templates are often natural candidates for inlining, this model seems all the more fitting. In some cases, for example with commercial libraries, it can be desirable to provide only the template declarations as source code and have the definitions as compiled binaries. To this end C++ defines the `export` keyword which basically means "accessible from another translation unit".

```
// file1.cpp
  export template<typename T> twice(T t) { return t+t; }

// file2.cpp
  template<typename T> twice(T t);  // declaration only
  int f(int i) { return twice(i); }
```

Unfortunately `export` is currently available only in one commercial compiler (Comeau 4.3.x which uses the EDG 3.x front end), has proven to be very hard to implement, and doesn't actually solve the problems it was created to solve [3].

2.7 Comparing Templates, Inheritance, and Composition

Templates can be seen as a way to re-use code along inheritance and composition. In C++ inheritance can express either a IS-A relationship (public inheritance) or a re-use of implementation (private inheritance). Composition expresses the relation HAS-A. Private inheritance and composition can be considered as solutions to similar problems. A template is a family of implementations whose members differ by the types used in their shared code.

Both inheritance and class templates work by creating new types. While inheriting creates subtypes, classes generated from class templates have no relationship type-wise. If inheritance and virtual functions allow *run-time polymorphism*, templates can be said to provide *compile-time polymorphism*.

Inheritance and composition are possible without the compiler having access to the class definition. When instantiating templates both the decla-

ration and the definition of the class template are needed (`export` notwithstanding).

Unlike inheritance which incurs a run-time penalty due to virtual function dispatching, templates have no inherent run-time cost. If templates are inlined, they can reveal further optimization opportunities leading to implementations that match the best hand written code while maintaining high level structure and type safety.

3 Template Meta-programming

Meta-programming means building programs that create other programs. Although originally templates were not designed for meta-programming, its has been discovered that they allow the use of powerful meta-programming techniques. It has even been shown that that any partial recursive function can be computed at compile time using techniques based on templates [4]. C++ can viewed as a two level languages were the first level consists of template programs that generates code for the second, conventional level. Here is a simple meta-program that instantiates templates recursively to generate factorials. Template specialization is used to end the recursion:

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

void main() {
  std::cout << Factorial<5>::value; //  prints 120
}
```

If-else constructs can be expressed as follows:

```
template<bool C> class Condition { };

class Condition <true>  {
  public: static inline void f() { statement1; }
}
```

```
class Condition <false>  {
  public: static inline void f() { statement2; }
}

void main() {
  // if condition==true generates statement1 else statement2
  Condition <condition>::f();
}
```

4 Notable Template Libraries

The template mechanism was added to C++ as a tool for building libraries. Originally it was meant mainly for defining container classes and generic functions. With the discovery of template meta-programming completely new kinds of application became possible.

4.1 STL

The most widely used C++ template library is the STL which is part of the C++ standard library. It offers containers (strings, lists, vectors, sets, maps), algorithms in the form of function templates and utilities.

4.2 Blitz++

One of the challenges of programing languages is to offer high level constructs to allow working on the proper abstraction level while maintaining good performance when needed. In the domain of numerical computing Fortran has been unsurpassed due to its highly optimized libraries. Blitz++ is a C++ library for scientific computing that using template techniques achieves performances on par with Fortran [5].

4.3 Spirit

Spirit [6], part of the Boost libraries, is a recursive-descent parser generator framework. Traditionally grammars have been expressed in a separated language which has been then translated into C/C++ code for inclusion into the target application. Spirit allows the programmer to write the grammar directly in C++, using a notation that approximated the Extended Backus-Normal Form (EBNF). This EBNF grammar of a simple calculator:

```
group      ::= '(' expression ')'
factor     ::= integer | group
term       ::= factor (('*' factor) | ('/' factor))*
expression ::= term (('+' term) | ('-' term))*
```

can be expressed in C++ using Spirit:

```
group      = '(' >> expression >> ')';
factor     = integer | group;
term       = factor >> *(('*' >> factor) | ('/' >> factor));
expression = term >> *(('+' >> term) | ('-' >> term))*;
```

The production rule `expression` is an object that has a member function `parse`.

4.4 FC++

FC++ brings Haskell like functional programming facilities to C++. FC++'s implementation relies heavily on templates and the C++ types system [7].

References

[1] Bjarne Stroustroup, *The Design and Evolution of C++*, Addison Wesley, 1994

[2] Bjarne Stroustroup, *The C++ Programming Language, Special Edition*, Addison Wesley, 2000

[3] Herb Sutter, Tom Plum, *Why We Cant Afford Export*, http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf

[4] Martin Böhme, Bodo Manthey, *The Computational Power of C++*, http://www.tomasoberg.com/pdf/compchem_030207.pdf

[5] *Blitz++*, http://www.oonumerics.org/blitz/

[6] *Spirit*, http://spirit.sourceforge.net/

[7] *FC++*, http://www.cc.gatech.edu/ yannis/fc++/