Understanding C++ Templates, Part 1

Chuck Allison
Associate Professor
Utah Valley University
Orem, Utah
chuck@freshsources.com

Agenda
(Parts 1 and 2)

- Executive Overview
- Template Parameters
- Function Template Issues
- Template Specialization
- A Generic Programming Session
- Member Templates
- Template Idioms
- Templates and friends
- Templates in C++0x (scattered throughout)

Executive Overview

- Generic Programming
- Compile-time Code Generation
- Structural Conformance (“Implicit Interfaces”)
- Template Terminology Overview

Generic Programming

- Evolved from design of algorithms and data structures
- Goal: to write *type-transparent* code
  - vs. **Object** mega-hierarchy; **void*** in C
- C++ prefers *type-safe* generics
  - Statically-checked
  - Allows efficient use of built-in types
    - No runtime class overhead
    - No need for autoboxing

Compile-time Code Generation

- Separate code versions are automatically generated
- On-demand code generation
  - Implicit and explicit
- Compile-time programming
  - Selective code generation
- *Potential* for Code Bloat
  - Not usually a problem

```
template<class T> class
Stack {
  T* data;
  size_t count;
public:
  void push(const T& t);
  // etc.
};

// Explicit instantiation
Stack<int> s;
```

What Gets Generated?

- If using a *function template*, the *requested function* code is instantiated
- If using a *class template*, the requested version of the *class definition* is instantiated
  - But *definitions* are generated only for the member functions *actually used*

On-Demand Instantiation

```
class X {
public:
  void f() {}
};

class Y {
public:
  void g() {}
};

template<typename T>
class Z {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};
```

```
int main() {
  Z<X> zx;
  zx.a();  // Z<X>::b() not attempted
  Z<Y> zy;
  zy.b();  // Z<Y>::a() not attempted
}
```

Question: *When* are the names **f** and **g** looked up by the compiler?

Where Should Template Code Reside?

- Usually totally in *header files*
- The template code must be available during instantiation
  - This is the *Inclusion Model* of template compilation
- There is a way to separate function template *declarations* from their *definitions* in the Classic Way
  - **export** and the *Separation Model*
  - It is fragile, and not widely supported or used
  - *Concepts* in C++0x may supplant this

Structural Conformance
("Implicit Interfaces" or "Compile-time polymorphism")

- aka "Duck Typing"
  - If it quacks like a duck, …
- The *status quo* for dynamically-typed languages
  - Perl, Python, PHP, Ruby…
- C++ *statically* verifies that generic types support required operations

```
template<class T>
const T& min(const T& a,
             const T& b)
{
  return (a < b) ? a : b;
}

…

  min(i, j) // T deduced
```

Template Terminology Overview

- Template *Parameter*
  - Names inside <>'s after the **template** keyword (<**T**>)
- Template *Argument*
  - Names inside <>'s in a specialization (<**int**>)
- Template *Specialization*
  - What results when you give a template some arguments (**Stack<int>**)
- Template *Instantiation*
  - The class or function generated for a given *complete set* of template arguments

Template Executive Summary

- Templates are instructions for **compile-time** code generation
- They enable **type-safe**, type-transparent code
- Template parameters constitute **implicit interfaces**
  - "Duck Typing"
- Classic OOP uses *explicit interfaces* and *runtime polymorphism*
  - Templates and OOP complement each other

So much for the Easy Stuff!

The Real Presentation begins now…

Template Parameters

- 3 Kinds…
- Type parameters
  - the most common (vector<**int**>)
- Non-type
  - compile-time integral values
- Templates
  - “template template parameters”

“Non-type” Template Parameters

- Must be compile-time constant values
  - usually **integral** expressions
  - can also be addresses of static objects or functions
  - should probably be called “value parameters”
- Often used to embed member arrays on the runtime stack
  - (see next slide)
- Often used in Template Metaprogramming
  - compile-time expressions governing code generation

std::bitset
(from STLPort)

```
template<size_t _Nw>
struct _Base_bitset {
  typedef unsigned long _WordT;

  _WordT _M_w[_Nw]; // Can go on stack
  . . .
};

template<size_t _Nb>
class bitset : public
               _Base_bitset<__BITSET_WORDS(_Nb) >
{ . . . };
```

Default Template Arguments

- Similar to default function arguments
  - if missing, the defaults are supplied
  - only allowed in *class templates*

- ```
  template<class T = int, size_t N = 100>
  class FixedStack {
      T data[N];
       …
  };
  FixedStack<> s1;          // = <int, 100>
  FixedStack<float> s2; // = <float,100>
  ```
  ```

The Container **std::vector**

- ```
  template<class T,
    class Allocator = std::allocator<T> >
  class vector;
  ```
- Note how the second template parameter uses the first
- Note the space between the '>'s
  - Fixed in C++0x
  ```

Template Template Parameters

- Templates are not *really* types
  - They are *instructions* for generating types or functions
- If you plan on using a template parameter itself *as a template*, the compiler needs to know
  - otherwise it won't let you do template things with it
  - You often don't *name* its parameters
  - Only *class templates* can be used this way
- Examples follow

A Simple Expandable Sequence

(*will be used as a template argument on next slide)*

```
// A simple, expandable sequence (like std::vector)
template<class T>
class Array {
      . . .
public:
   Array();
   ~Array();
   void push_back(const T& t);
   void pop_back();
   T* begin();
   T* end();
};
```

Passing **Array** as a Template Arg.

```
template<class T, template<class> class Seq>
class Container {
  Seq<T> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container; // Pass template
  . . .
}
```

Template-Template Parameters and Default Arguments

- Template template *arguments* must match their receiving template template parameters *exactly*
  - Including any *default* arguments
- Example on next slide

```
template<class T, size_t N = 10>  // change
class Array {...};

template<class T,
         template<class, size_t = 10> class Seq>
class Container {
  Seq<T> seq;  // Default used
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container;
  . . .
}
```

Passing Standard Sequence Templates as Template Arguments

- **std::allocator<T>** is part of standard container template declarations
  - It is a default second argument
- Example on next slide
  - Modifies **Container** to repeat **std::allocator<T>** default argument

```
template<class T,
         template<class U, class = allocator<U> >
         class Seq>
class Container {
  Seq<T> seq; // Default of allocator<T> applied
public:
  void push_back(const T& t) { seq.push_back(t); }
  typename Seq<T>::iterator begin() {return seq.begin();}
  typename Seq<T>::iterator end() {return seq.end();}
};

int main() {
  // Use a vector
  Container<int, vector> vContainer; // Could use deque
}
```

Function Template Issues

- Type Deduction of Arguments
- Function template overloading
- Partial Ordering of Function Templates

Type Deduction in Function Templates

- Under most circumstances, the compiler *deduces* type parameters from the arguments in a call:
  - the corresponding specialization is instantiated automatically
  - but standard conversions do *not* apply!
- You can use a *fully-qualified call syntax* if you want to:
  int x = min<int>(a, b); // vs. **min(a, b);**
- Sometimes you *have* to:
  - when the arguments are different types (**min(1.0, 2)**)
  - when the template argument is a *return type only*, and therefore cannot be deduced by the arguments

Useful String Conversion Function Templates

```
// StringConv.h
#include <string>
#include <sstream>

template<class T> T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}

template<class T> std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
```

```
int main() {
  // Implicit Type Deduction
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"" << endl;
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"" << endl;
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"" << endl;
  cout << endl;

  // Explicit Function Template Specialization
  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString<complex<float> >(string("(1.0,2.0)"));
  cout << "c == " << c << endl;
}
```

Another Example
implicit_cast (takes 2 type arguments)

```
template <class U, class T>
U implicit_cast(const T& t) {
    return t;
}
```

- Makes a legal implicit conversion visible in code
- Must specify return type (**U**)
  - Can't be deduced (**s = implicit_cast<string>(x);**)
- Can omit **T**
  - But only because we put it *last* in the parameter list
  - Couldn't say **implicit_cast<?, string>** (no wildcards)

Function Templates and Default Arguments

- Can even use template parameters in default function arguments:

```
template<class T>
T init(const T& t = T()) {    // zero initialization
    return t;
}

int main() {
    cout << init<int>() << endl; // 0
    cout << init(5) << endl;     // 5
}
```

Zero Initialization

- Introduced to allow “default construction” of built-in types in templates:

```
template<class T>
class Foo {
    T data;
public:
    Foo() : data() {} // T could be int
…
};
```

Function Template Overloading

- You can define multiple functions and function templates with the same name
- The “best match” will be used
  - Can force using a template with “<>”
- You can overload a function template by having a different function “signature” (including template parameters)

```
template<class T>
const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
double min(double x, double y) {
  return (x < y) ? x : y;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(1, 2) << endl;       // 1: 1 (template)
  cout << min(1.0, 2.0) << endl;   // 2: 1 (double)
  cout << min(1, 2.0) << endl;     // 3: 1 (double)
  cout << min(s1, s2) << endl;     // 4: knights who
                                   //    (const char*)
  cout << min<>(s1, s2) << endl;   // 5: say "Ni-!"
                                   //    (template)
}
```

A Template may be the Best Match

- Given the template:
  template<class T> void f(T, T);
- And the plain function: **void f(long, int);**
- The best match for **f(1,2)** is…
  - The template
    - No conversions required
- But **f(1, '2')** will match the plain function

Partial Ordering of Function Templates

- Plain functions are often considered better than templates
  - Why generate another function when an existing one will suffice?
- Some templates are better than others
  - Considered more “specialized” if matches more combinations of arguments types than another…

```
// Overloaded templates
template<class T> void f(T) {
  cout << "T" << endl;
}

template<class T> void f(T*) {
  cout << "T*" << endl;
}

template<class T> void f(const T*) {
  cout << "const T*" << endl;
}

int main() {
  f(0);             // T
  int i = 0;
  f(&i);            // T*
  const int j = 0;
  f(&j);            // const T*
}
```

Things to Remember…
About Function Templates

- Arguments that can be deduced, will be
  - The rest you must provide (place them *first*)
- Standard Conversions do *not* apply when using unqualified calls to function templates
  - *Except*: pointer decay, const/volatile
- Function Templates can be overloaded
  - Ordinary functions trump *equivalent* function templates

Template Specialization

- A template by nature is a *generalization*
- It becomes *specialized* for a particular use when the actual template arguments are provided
- An instantiation is therefore a *specialization*
- But there is another type of "specialization"…

Explicit Specialization

- What if you want special “one-off” behavior for certain combinations of template arguments?
  - Common case: **T = char***
- You can provide *custom code* for such cases
  - both *full* or *partial* specializations for class templates
  - the compiler will use your explicit versions instead of what the “primary template” would have instantiated
  - Note: *no* automatic instantiation is needed
- Full specialization uses the **template<>** syntax

Full Class Specialization

Example

```
// A Primary Template
template<class T>
class Foo {
public:
    void f();      // member function decls…
    void g();
};

template<class T> // member function defs…
void Foo<T>::f() {
    cout << "f using primary template\n";
}

template<class T>
void Foo<T>::g() {
    cout << "g using primary template\n";
}
```

Full Class Specialization

*Specializes on **int***

```
// A Full Specialization
template<>
class Foo<int> {
public:
    void f();
    void g();
};

// NOTE: No template keyword here
// (It's a "real function"; place in .cpp file)
void Foo<int>::f() {
    cout << "f using int specialization\n";
}

void Foo<int>::g() {
    cout << "g using int specialization\n";
}
```

Full Class Specialization

Example Usage

```
int main() {
    Foo<char> c;
    Foo<int> i;
    c.f();
    c.g();
    i.f();
    i.g();
}

/* Output:
f using primary template
g using primary template
f using int specialization
g using int specialization
*/
```

Explicit Specialization of Function Templates

- *Full specializations* are allowed, but you can always just provide a plain function to do the job
  - Plain overloading is sufficient and easier to grok
- *Partial specialization* of function templates is not currently allowed
  - Function template overloading covers most needs
- Full function template specialization example on next slide

A Full Function Template Specialization

```
// The primary template
template<class T> const T& min(const T& a, const T& b) {
   return (a < b) ? a : b;
}

// An explicit specialization of the min template
template<>
const char* const &
min<const char*>(const char* const& a,
                 const char* const& b) {
   return (strcmp(a, b) < 0) ? a : b;
}

int main() {
   const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
   cout << min(s1, s2) << endl;   // uses specialization
}
```

Function Template Specialization
Example courtesy Sutter, Abrahams, Dimov

```
template<class T> // (a) a base template
void f( T );

template<class T> // (b) another, overloads (a)
void f( T* );

template<>        // (c) explicit specialization of (b)
void f<>(int*);

// ...
int *p;
f( p );           // calls (c) [why???]
```

Swap b) and c) – Surprise!
Example courtesy Sutter, Abrahams, Dimov

```
template<class T> // (a) same base template as before
void f( T );

template<>        // (c) explicit specialization of (a)
void f<>(int*);

template<class T> // (b) another base template, overloads (a)
void f( T* );

// ...
int *p;
f( p );           // calls (b)!
```

Analysis

- Specializations of function templates are *not* considered during overload resolution
  - Only after a primary template is found are specializations are considered
- Moral: Don't Specialize Function Templates!
  - *except* as directed 3 slides from now for member functions of class templates

Explicit Specialization and Default Arguments

- Example: **vector<bool>**
  - packs bits, like **bitset** (but is dynamically sized)
- **vector** is defined as:

```
template<class T, class Allocator = allocator<T> >
class vector {...};
```

- **vector<bool>** *could* be defined as:

```
template<> class vector<bool> // default arg used
{…};
```

  - That would be a full/total (explicit) specialization
  - Note how the default allocator argument applies to the specialization

About Explicit Class Template Specialization

- There is little direct relationship between a primary template and its specializations
  - Except *default arguments* apply
- A specialization does not have to provide all members
- Member function signatures don't have to match!
  - But they should be *call-compatible*
- You can also specialize just *selected member functions*!
  - But this is *not* a *class* specialization!

Specializing Selected Member Functions

- You can do a full specialization of selected member functions, if desired
- The other member functions in the primary template are still available
  - (See next slide)
- *All types* of members can be selectively specialized
  - Static members, Member templates,…

Selective Member Function Spec.

```
template<>   // required now!
void Foo<int>::g() {
    cout << "g using int specialization\n";
}

int main() {
    Foo<char> c;
    Foo<int> i;
    c.f();
    c.g();
    i.f();
    i.g();
}
```

```
f using primary template
g using primary template
f using primary template
g int specialization
```

Compile-time Computation
(from David Abrahams)

```
template<unsigned long N>
struct binary {
   static const unsigned int value =
      binary<N/10>::value << 1 | N%10;
};

// A (full/total) specialization to stop the recursion.
template<>
struct binary<0> {
   static const unsigned int value = 0;
};

int main() {
   cout << binary<101>::value << endl;           // 5
   cout << binary<1001101110>::value << endl;    // 622
   cout << binary<11100111>::value << endl;      // 231
}
```

Partial Specialization of Class Templates

- Can specialize on a *subset* of template arguments
  - leaving the rest unspecified
  - can also specialize on: “pointer-ness”, “const-ness”, type equality, and much more
- **vector<bool>** is actually a partial specialization
  - It specializes the data type, but leaves the allocator type “open”:

```
template<class Allocator>
class vector<bool, Allocator> {…};
```

- The “most specialized, most restricted” match is preferred
  - (See next slide)

```
template<class T, class U> class C {
public:
   void f() { cout << "Primary Template\n"; }
};
template<class U> class C<int, U> {
public:
   void f() { cout << "T == int\n"; }
};
template<class T> class C<T, double> {
public:
   void f() { cout << "U == double\n"; }
};
template<class T, class U> class C<T*, U> {
public:
   void f() { cout << "T* used\n"; }
};
template<class T, class U> class C<T, U*> {
public:
   void f() { cout << "U* used\n"; }
};
```

```
template<class T, class U> class C<T*, U*> {
public:
    void f() { cout << "T* and U* used\n"; }
};
template<class T> class C<T, T> {
public:
    void f() { cout << "T == U\n"; }
};

int main() {
  C<float, int>().f();     // 1: Primary template
  C<int, float>().f();     // 2: T == int
  C<float, double>().f();  // 3: U == double
  C<float, float>().f();   // 4: T == U
  C<float*, float>().f();  // 5: T* used [T is float]
  C<float, float*>().f();  // 6: U* used [U is float]
  C<float*, int*>().f();   // 7: T* and U* used [float, int]
```

```
// The following are ambiguous:
//    8: C<int, int>().f();
//    9: C<double, double>().f();
//   10: C<float*, float*>().f();
//   11: C<int, int*>().f();
//   12: C<int*, int*>().f();
}
```

Type Tricks with Specialization

- Discover whether two type parameters are the same type
  - Example: **IsSame.cpp**
- Discover whether a template parameter is a template
  - Example: **IsTemplate.cpp**
- (These examples are adapted from Steve Dewhurst; see Boost.type_traits for more)

```
// isSame.cpp
template<class T1, class T2>  // primary
struct IsSame {
   enum { result = false };
};

template<class T>             // partial spec.
struct IsSame<T, T> {
   enum { result = true };
};

int main() {
   // Evaluated at compile time!
   const bool q1 = IsSame<int, int>::result;
   const bool q2 = IsSame<int, double>::result;
   cout << q1 << endl;  // 1
   cout << q2 << endl;  // 0
}
```

```
// IsTemplate.cpp
// The Primary template
template<class T>
struct IsTemplate {
   enum { numargs = 0 };
};

// Partial spec. for templates with 1 type parm
template<template<class> class X, class T>
struct IsTemplate< X<T> > {
   enum { numargs = 1 };
   typedef T type;   // extract T!
};

// Partial spec. for templates with 2 type parms
template<template<class, class> class X, class T1,
         class T2>
struct IsTemplate< X<T1, T2> > {
   enum { numargs = 2 };
   typedef T1 type1;
   typedef T2 type2;
};
```

```
// A test template
template<class T>
struct Foo {
   T t;
};

int main() {
   typedef IsTemplate<int> Int;
   if (Int::numargs == 0)
      cout << "int is not a template\n";

   typedef IsTemplate< Foo<double> > Foo_t;
   if (Foo_t::numargs) {
      cout << "Foo is a template with "
           << Foo_t::numargs << " argument(s)\n";
      cout << "Its argument here is "
           << typeid(Foo_t::type).name() << endl;
   }
}
```

```
    typedef IsTemplate< vector<int> > IntVec;
    if (IntVec::numargs == 2) {
        cout << "vector is a template with " << IntVec::numargs
             << " argument(s)\n";
        cout << "Its first argument here is "
             << typeid(IntVec::type1).name() << endl;
        cout << "Its second argument here is "
             << typeid(IntVec::type2).name() << endl;
    }
}

/* Output:
int is not a template
Foo is a template with 1 argument(s)
Its argument here is double
vector is a template with 2 argument(s)
Its first argument here is int
Its second argument here is std::allocator<>
*/
```

```
// A Print function for standard sequences
template<typename T,
         template<typename U,
                  typename = allocator<U> >
         class Seq>
void printSeq(const Seq<T>& seq) {
  for(typename Seq<T>::const_iterator b = seq.begin();
       b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Process a vector
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Process a list
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
}
```

C++0x Version
Using **`auto`**

```
template<typename T,
         template<typename U,
                  typename = allocator<U> >
         class Seq>
void printSeq(Seq<T>& seq) {
  for(auto b = seq.begin(); b != seq.end();)
     cout << *b++ << endl;
}
```

A Slightly More Idiomatic Version

Uses **std::copy**

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include <vector>
using namespace std;

template<typename T,
         template<typename U,
                  typename = allocator<U> >
         class Seq>
void printSeq(const Seq<T>& seq) {
  copy(seq.begin(),seq.end(),ostream_iterator<T>(cout,
                                                 "\n"));
}
```

Question

- Can we write a version of **printSeq** that will work for both standard sequences and arrays?
- Well, sure, but we'll have to take two *iterator* arguments instead:

  template<class Iter> void printSeq(Iter b, Iter e);

- But… how do we infer **T** from the iterator type?
  - Required to instantiate **ostream_iterator**

iterator_traits
An application of partial specialization

- A technique for endowing naked pointers with what other iterators have:
  - difference_type
  - value_type
  - pointer
  - reference
  - iterator_category

Implementing iterator_traits

```
// Primary template
template<class Iterator> struct iterator_traits {
  typedef typename Iterator::difference_type difference_type;
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::pointer pointer;
  typedef typename Iterator::reference reference;
  typedef typename Iterator::iterator_category
    iterator_category;
};

// Partial specialization for pointer types
template<class T> struct iterator_traits<T*> {
  typedef ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef random_access_iterator_tag iterator_category;
};
```

Using iterator_traits

```
// Print any standard sequence or an array
template<class Iter>
void printSeq(Iter b, Iter e) {
  typedef typename iterator_traits<Iter>::value_type T;
  copy(b, e, ostream_iterator<T>(cout, "\n"));
}
```

What Have We Covered?

- Template Parameters
- Type Deduction in Function Templates
- Function Template Overloading
- Explicit Specialization
  - Full, partial
- **typename**
- **iterator_traits**

End of Part 1