Understanding C++ Templates, Part 2

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

Agenda
As time allows…

- A Generic Programming Session
- Member Templates
- Template Idioms
- Templates and Friends

A Generic Programming Session

- Task: Write a class that simulates the composition of an arbitrary number of functions
- $\mathbf{f_1(f_2(\ldots f_n(x)\ldots))}$
- Strategy:
  - Hold the function pointers in some sequence container
  - Apply them in reverse order, starting with $f_n(x)$

Aside: About **std::accumulate**

- Two overloads:
- **accumulate(beg,end,init)**
  - Computes **init+x_1+x_2+…+x_n**
- **accumulate(beg,end,init,f)**
  - Computes f(…(f(f(init,x_1),x_2),…,x_n)
  - "Fold left" (**foldl**) in ML and Haskell
- Our **x**'s will be the functions, and our **f** will *apply* each **x** in turn to the running result…

A Generic Programming Session

Version 1 – vector of f(double)

```
typedef double (*Fun)(double);

class Composer {
private:
    vector<Fun> funs;
    static double apply(double sofar, double (*f)(double)) {
        return f(sofar);
    }
public:
    Composer(vector<Fun>& fs) : funs(fs) {
        reverse(funs.begin(), funs.end());
    }
    double operator()(double x) {
        return accumulate(funs.begin(), funs.end(), x, apply);
    }
};
```

A Generic Programming Session

Version 1 - Test

```
double f(double x) {
    return x*x;
}
double g(double x) {
    return x+1;
}
double h(double x) {
    return x/3.0;
}

int main() {
    vector<Fun> funs;
    funs.push_back(h);
    funs.push_back(f);
    funs.push_back(g);
    Composer comp(funs);    // h(f(g(x)))
    cout << comp(2.0);      // 3
}
```

A Generic Programming Session

Version 2 – vector of f(T)

```
template<class>                  // Primary template declaration
class Composer;

template<class T>                // A Partial Specialization
class Composer<T (*)(T)> {
private:
    typedef T (*FunType)(T);
    vector<FunType> funs;
    static T apply(T sofar, T f(T)) {
        return f(sofar);
    }
public:
    Composer(const vector<FunType>& fs) : funs(fs) {
        reverse(funs.begin(), funs.end());
    }
    T operator()(T x) {
        return accumulate(funs.begin(), funs.end(), x, apply);
    }
};
```

A Generic Programming Session

Version 2 - Test

```
string f(string s) { return s + "s"; }
string g(string s) { return "the" + s; }
string h(string s) { return " " + s; }

int main() {
    typedef string (*Fun)(string);
    vector<Fun> funs;
    funs.push_back(f);
    funs.push_back(g);
    funs.push_back(h);
    Composer<Fun> comp(funs);
    cout << comp("boy") << endl; // "the boys"
}
```

A Generic Programming Session

*Version 3 – Generalize the Sequence via **Iterators***

```
template<class T, class Iter>
class Composer {
private:
    typedef std::reverse_iterator<Iter> RevIter;
    RevIter beg, end;       // Iterators, not a container
    static T apply(T sofar, T f(T)) {
        return f(sofar);
    }
public:
    Composer(Iter b, Iter e) : beg(RevIter(e)), end(RevIter(b))
    {}
    T operator()(T x) {
        return accumulate(beg, end, x, apply);
    }
};
```

A Generic Programming Session

Version 3 – Test

```
int main() {
    typedef double (*Fun)(double);

    // Try a vector
    vector<Fun> funs;
    funs.push_back(h);
    funs.push_back(f);
    funs.push_back(g);
    Composer<double, vector<Fun>::iterator >
      comp(funs.begin(), funs.end());
    cout << comp(2.0) << endl;

    // Try an array
    Fun funs2[] = {h,f,g};
    Composer<double, Fun*> comp2(funs2, funs2+3);
    cout << comp(2.0) << endl;
}
```

A Generic Programming Session

Version 4 – Generalize the Callable Type

```
template<class T, class Iter>
class Composer {
private:
    typedef std::reverse_iterator<Iter> RevIter;
    RevIter beg, end;
    static T apply(T sofar, std::tr1::function<T(T)> f) {
        return f(sofar);
    }
public:
    Composer(Iter b, Iter e) : beg(RevIter(e)), end(RevIter(b))
    {}
    T operator()(T x) {
        return std::accumulate(beg, end, x, apply);
    }
};
```

A Generic Programming Session

Version 4 – Test with Function Objects

```
template<class T>
struct square {
    T operator()(T t) const {
        return t*t;
    }
};
template<class T>
struct plus1 {
    T operator()(T t) const {
        return t+1;
    }
};
template<class T>
struct div3 {
    T operator()(T t) const {
        return t/3.0;
    }
};
```

A Generic Programming Session

Version 4 – Test with Function Objects

```
int main() {
    std::list<Fun> funs3;
    funs3.push_back(div3<double>());
    funs3.push_back(square<double>());
    funs3.push_back(plus1<double>());
    double nums[] = {1.0,2.0,3.0};
    comp3(funs3.begin(), funs3.end());
    std::cout << comp3(2.0) << std::endl;  // 3
}
```

A Generic Programming Session

*Version 5 – Infer T from **tr1::function::result_type***

```
template<class Iter>
class Composer {
private:
    // Deduce function and return/argument types
    typedef typename iterator_traits<Iter>::value_type Fun;
    typedef typename Fun::result_type T;

    // Declare reverse iterators (see constructor for use)
    typedef reverse_iterator<Iter> RevIter;
    RevIter beg, end;

    // The function called by accumulate
    static T apply(T sofar, Fun f) {
        return f(sofar);
    }
```

A Generic Programming Session

Version 5 – Infer T from ***tr1::function::result_type***

```
public:
    Composer(Iter b, Iter e) : beg(RevIter(e)), end(RevIter(b))
    {}
    T operator()(T x) {
        return accumulate(beg, end, x, apply);  // Applicator
    }
};
```

A Generic Programming Session

Version 5 – Introduce a Helper Function

```
template<Class Iter>
Composer<Iter> compose(Iter b, Iter e) {
    return Composer<Iter>(b, e);
}
```

A Generic Programming Session

Version 5 – Test

```
int main() {
    double nums[] = {1.0, 2.0, 3.0};

    // Use an array
    typedef tr1::function<double(double)> Fun;
    Fun funs[] = {h,f,g};
    transform(nums, nums+3,
              ostream_iterator<double>(cout," "),
              compose(funs,funs+3));
    cout << endl;
}
```

A Generic Programming Session

*Version 5a – Use C++0x **auto***

```
int main() {
    double nums[] = {1.0, 2.0, 3.0};

    // Use an array
    typedef std::tr1::function<double(double)> Fun;
    Fun funs[] = {h,f,g};
    auto = compose(funs,funs+3));
    cout << comp(2.0) << endl;
}
```

Just for Grins…

Python Version

This one does it all!

```
def compose(*funs):
    return lambda x: reduce(lambda z,f: f(z),
                            reversed(funs), x)


>>> c = compose(h,f,g)
>>> print c(2.0)
3
```

D Version

```
T delegate(T) compose(T)(T function(T)[] funs) {
   T doit(T n) {
      T result = n;
      foreach_reverse (f; funs)
         result = f(result);
      return result;
   }
   return &doit;
}

   // Give it a whirl…
   int function(int)[] funs;
   funs ~= function int(int x){return x*x;};
   funs ~= function int(int x){return x+1;};
   auto c = compose(funs);
   writeln(c(3));    // 16
```

Question

- What kind of things can you define as members of a class?

Answer

- **Variables**
  - Data members; either static or non-static
- **Functions**
  - Member functions, either static and non-static
- **Types**
  - Nested classes or **typedef**s
- **Templates**
  - “Member Templates”
  - This is where the need for a special use of the **template** keyword arises

Member Templates

- Defining a template inside a class
  - An independent set of template parameters
- Very handy for *conversion constructors*:
  - **template<typename T> class complex {**
    public:
    template<class X> complex(const complex<X>&);
  - Inside of STL sequences:
    template <class InputIterator>
    deque(InputIterator first, InputIterator last,
    const Allocator& = allocator());
- Example on next slide

A Member Class Template

```
template<class T> class Outer {
public:
  template<class R> class Inner {
  public:
    void f();
  };
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Full Inner == " << typeid(*this).name()
       << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
}
```

Output from Previous Example

Outer == int
Inner == bool
Full Inner == Outer<int>::Inner<bool>

Member Function Templates

- Used in practice more than member *class* templates
- Cannot not be **virtual**
  - Because vtables are statically determined by the class *definition*

Member Function Templates
Example

- Example: **std::bitset::to_string**
  - `template <class charT, class traits, class Allocator> basic_string<charT, traits, Allocator> to_string() const;`
  - Because strings are themselves templates!
    - **char** vs. **wchar_t**
- So you must explicitly specialize a call to **bitset::to_string( )** to indicate the *return type*
- But compilers assume member functions are *not* templates!
  - You need to give them a *clue* about the '<'
  - Similar to why **typename** exists

Avoiding Parse Errors

- There are contexts in which a ‘<‘ will be interpreted as a *less-than operation*, instead of the *beginning of a template* argument list
- Example:
  - Convert a **bitset** to a **string:**
  - ```
    string s =
      // error:
      bs.to_string<char, char_traits<char>,
                   allocator<char> >();
    ```
    ```

The **template** keyword as a disambiguator

```
template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
  return bs.template to_string<charT,
                               std::char_traits<charT>,
                               std::allocator<charT> >();
}

int main() {
  bitset<10> bs;
  bs.set(1);
  bs.set(5);
  cout << bs << endl; // 0000100010
  string s = bitsetToString<char>(bs);
  cout << s << endl;  // 0000100010
}
```

Name Classification

- Names can be nicely dichotomized as:
  - *Unqualified*
    - x, f( )
  - *Qualified* (provides scope context)
    - A::x, x.f( ), p->f( )
- Names inside templates are also either:
  - *Dependent*
    - They depend on a template parameter: e.g., **T::iterator**
    - We used **typename** earlier for dependent *types*
  - *Non-dependent*

Template Name Lookup Issues

- Some things cannot be resolved when the compiler first encounters a template definition
  - Anything *dependent* on a template parameter
- The compiler must wait until *instantiation time* to resolve those issues
  - When the actual template arguments are known
- Hence, a 2-step template compilation process:
  - 1) Template Definition
  - 2) Template Instantiation

Two-phase Template Compilation

- Template definition time
  - The template code is parsed
  - Obvious syntax errors are caught
  - Non-dependent names are looked up in the context of the template definition (no waiting needed)
    - Unqualified names are often non-dependent
- Template instantiation time
  - Dependent names are resolved
  - Which specialization to use is determined
    - A key motivation for 2-phase lookup
- Examples follow

What Output Should Display?

```
void f(double) { cout << "f(double)" << endl; }

template<class T> class X {
public:
  void g() { f(1); }
};

void f(int) { cout << "f(int)" << endl; }

int main() {
  X<int>().g();
}
```

Dependent Base Classes

- Nondependent names are *not* looked up in base classes
  - Because a specialization found later may apply
- If a class template has a dependent base class, you want calls to inherited functions to be looked up in Phase 2
  - When specializations are resolved
- Calls to such functions must be *qualified*
  - This makes the name *dependent*
  - Otherwise they'll be looked up in Phase 1 and will fail
- See next slide

A Dependent Base Class

Flawed Version (won't compile)

```
template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(begin(),end(), comp);
  }
  const T& top() const { return front(); }
  void push(const T& x) {
    push_back(x);
    push_heap(begin(), end(), comp);
  }
  void pop() {
    pop_heap(begin(), end(), comp);
    pop_back();
  }
};
```

A Dependent Base Class

Correct Version

```
template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(this->begin(),this->end(), comp);
  }
  const T& top() const { return this->front(); }
  void push(const T& x) {
    this->push_back(x);
    push_heap(this->begin(),this->end(), comp);
  }
  void pop() {
    pop_heap(this->begin(),this->end(), comp);
    this->pop_back();
  }
};
```

Rule of Thumb

- When necessary, *clue* the compiler as to which names are dependent by *qualifying* them if necessary
  - They'll get looked up in Phase 2
- Take-away:
  - Non-dependent names are looked up *immediately*
    - i.e., in Phase I
  - But they won't be looked up in dependent base classes
  - "x" is not always the same as "this->x"!

`this->` vs. `Base<T>::`

- Either one suffices to make a name dependent
- But the latter explicitly identifies the scope
- If you want polymorphism, use **`this->`**

Template Programming Idioms

- Traits
- Policies
- Curiously Recurring Template Patterns (CRTP)
- Template Constraints
  - And a peek at C++0x Concepts

Traits

- Traits usually hold *data* for chosen specializations of a template
  - A nice code-factoring idiom
  - 1) Define a default primary template (could be empty)
  - 2) Fully specialize on selected template arguments of interest
- Examples:
  - Numeric Limits (from the Standard Library)
  - IEEE Limits (my stuff)

Numeric Limits

- Defined in the header <limits>
- Define key values for the different numeric types
  - max, min, digits10, etc.
- Uses a default primary template
- Overloaded for all numeric types
  - Some members don't apply to all types
  - e.g., epsilon( )
- See next slide

std::numeric_limits

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;
  static T min() throw(); // for float, etc.
  static T max() throw();
  static const int digits = 0;
  static const int digits10 = 0;
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();
...
```

Using numeric_limits

```
#include <limits>
#include <iostream>
using namespace std;

int main() {
    cout << numeric_limits<int>::min() << endl;
    cout << numeric_limits<int>::max() << endl;
    cout << numeric_limits<int>::epsilon() << endl;
    cout << numeric_limits<int>::is_signed << endl;
    cout << numeric_limits<int>::is_integer << endl;
    cout << numeric_limits<int>::digits << endl;
    cout << numeric_limits<float>::min() << endl;
    cout << numeric_limits<float>::max() << endl;
    cout << numeric_limits<float>::epsilon() << endl;
    cout << numeric_limits<float>::is_signed << endl;
    cout << numeric_limits<float>::is_integer << endl;
    cout << numeric_limits<float>::digits << endl;
}
```

IEEE Traits
(My Own Stuff)

- For numeric programming
- Encapsulate attributes of IEEE numbers
  - For 32-bit (float), 64-bit (double)
  - Traits: # of exponent bits, storage bias,…
- Used by my generic IEEE support functions
  - fraction(x), exponent(x), signbit(x), isinfinity(x),…
- Uses an empty primary template
  - There are no common default values

IEEE Traits
(continued)

```
// Primary template
template<typename T>
struct IEEE_traits;

template<>
struct IEEE_traits<float> {
    typedef unsigned int
            IType;
    enum {
        exp_bits = 8,
        frac_bits = 23,
        bias = 127
    };
};
```

```
template<>
struct IEEE_traits<double> {
    typedef unsigned long
            long IType;
    enum {
        exp_bits = 11,
        frac_bits = 52,
        bias = 1023
    };
};
```

Using IEEE_Traits

```
template<typename FType>
bool isinfinity(FType x) {
   return exponent(x) == IEEE_traits<FType>::bias+1 &&
          fraction(x) == 0;
}

template<typename FType>
bool isnan(FType x) {
   return exponent(x) == IEEE_traits<FType>::bias+1 &&
          fraction(x) != 0;
}
```

Policies

- Similar to traits, but the emphasis is on associating *functionality* with a template parameter
  - not data
- A compile-time application of the Strategy Design Pattern
- Perfected by Andrei Alexandrescu
  - *Modern C++ Design*

C++'s Container Adapters
Policies in Action

- **queue**, **stack**, **priority_queue**
- Implemented with an underlying *sequence* data structure
  - **vector**, **deque**, or **list** or roll your own
- You "glue them together" at compile time:
  - `queue<int, list<int> >`
  - Default storage "policy" is `deque<T>`

Allocators

- Recall the definition of std::vector:

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

- **`std::allocator<T>`** is a *memory management policy*
  - It uses **new** and **delete**
  - But you can provide your own custom allocator class
    - A pool allocator, say

Alexandrescu's Singleton

- Has 4 template parameters:
  - The class to "singleton-ize"
  - Storage Policy
  - Lifetime Policy
  - Threading Policy
- **Singleton<MyClass,CreateStatic,NoDestroy> x;**
  - Defaults to **SingleThreaded**

Counting Objects

- How can you track the number of current "live" objects of a class?

Counting Non-template objects

```
// This is C++ 101:
class CountedClass {
  static int count;
public:
  CountedClass() { ++count; }
  CountedClass(const CountedClass&) { ++count; }
  ~CountedClass() { --count; }
  static int getCount() { return count; }
};

int CountedClass::count = 0;
```

Observation

- The logic for counting objects is type-independent
- It would be a shame to replicate that code for each class to be counted
- How can we reuse the counting logic?

How *not* to share Counting Code

```
class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};
int Counted::count = 0;

// All derived classes share the same count!
class CountedClass : public Counted {};
class CountedClass2 : public Counted {};
```

The Solution

- We need a *separate*, *static* count for each class to be counted
- Therefore, we need to derive from a *different class* for each client class
- Hence, we need to use *both* inheritance (OOP) *and* templates (compile-time polymorphism)
- See next slide

A Template Counter Solution

```
template<class> class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted &) { ++count; }
  virtual ~Counted() { --count; }
  static int getCount() { return count; }
};
template<class T> int Counted<T>::count = 0;

// Curious class definitions!!!
class CountedClass : public Counted<CountedClass>
{};
class CountedClass2 : public Counted<CountedClass2>
{};
```

A Curiously Recurring Template Pattern (CRTP)

- A class, **T**, inherits from a template that specializes on **T**!
- **class T : public X<T> {…};**
- Only valid if the size of **X<T>** can be determined independently of **T**
  - i.e., during Phase 1

Singleton via CRTP

- Inherit “Singleton-ness”
- Uses Meyers’ static singleton object approach
  - Since nothing static is inherited, the size is known at template definition time
- Protected constructor, destructor
- Disables copy/assign
- See next slide

Singleton via CRTP

```
// Base class - encapsulates singleton-ness
template<class T> class Singleton {
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);
protected:
  Singleton() {}
  virtual ~Singleton() {}
public:
  static T& instance() {
    static T theInstance; // Meyers' Singleton
    return theInstance;
  }
};
```

Making a Class a Singleton

```
// A sample class to be made into a Singleton
class MyClass : public Singleton<MyClass> {
  int x;
protected:
  friend class Singleton<MyClass>; // to create it
  MyClass() { x = 0; }
public:
  void setValue(int n) { x = n; }
  int getValue() const { return x; }
};

int main() {
  MyClass& m = MyClass::instance();
  cout << m.getValue() << endl;
  m.setValue(1);
  cout << m.getValue() << endl;
}
```

Constraining Template Arguments

- You've probably noticed…
- If a template argument used with STL doesn't support a required operation, the error usually turns up deep inside STL's nether parts
  - Nigh impossible to decipher
- It is possible to place constraints on template arguments to better localize the error
- Example from Bjarne Stroustrup
  - Next slide

```
#include <string>
using namespace std;

// A constraint class (inspects T for operations)
template<typename T>
struct Comparable {
   static void constraint(T a, T b) {
      // Exercise required operations
      //(errors will point here)
      (void)(a < b);
      (void)(a <= b);
   }
   Comparable() {
      // Force instantiation of static function above
      void (*p)(T,T) = constraint;
   }
};

template<typename T>
class Subject : private Comparable<T>
{};
```

```
// A class that is not Comparable
struct Foo{};

int main() {
   Subject<int> s1;
   Subject<string> s2;
   Subject<Foo> s3;  // Causes errors below
}
```

```
Error E2093 f:\3370\constraints.cpp 9: 'operator<' not
implemented in type 'Foo' for arguments of the same type
in function Comparable<Foo>::constraint(Foo,Foo)
Error E2093 f:\3370\constraints.cpp 10: 'operator<=' not
implemented in type 'Foo' for arguments of the same type
in function Comparable<Foo>::constraint(Foo,Foo)
```

Template Concepts
Example from Bjarne Stroustrup

- C++0x will support template argument constraints
  - See Container and Comparable below
- Still under construction
- Will make obsolete *most* uses of traits and other template tricks

```
template<Container C, Comparable Cmp>
requires Assignable<Cmp::argument_type,
                    C::value_type> &&
         Callable<Cmp, C::value_type>
void sort(C& c, Cmp less);
```

Notation

- **`template<Concept Type>`** is shorthand for:

```
template<class Type> …
requires Concept<Type>
```

Defining Concepts
Example from Bjarne Stroustrup

```
concept Assignable<class T> {
    T& operator=(const& T);
}

concept Comparable<class T> {
    bool operator==(const T&, const T&);
    bool operator!=(const T& x, const T& y) {
        return !(x == y); // Synthesized default
    }
}
```

printSeq with Concepts
Uses Overloading

```
// This one won't work for arrays
template<Container C>
void printSeq(const C& seq) {
  copy(seq.begin(),seq.end(),
       ostream_iterator<typename C::value_type>(cout,"\n"));
}

// Pointers (decayed arrays) are Forward_Iterators, and
// value_type is provided (similar to iterator_traits)
// via a Concept Map ("specialization" of Forward_Iterator
// for pointers). Works for arrays and conforming sequences.

template<Forward_Iterator Iter>
void printSeq(Iter begin, Iter end) {
  copy(begin, end,
       ostream_iterator<typename Iter::value_type>(cout,"\n");
```

Using Concepts: Summary

- Allows readable, checkable specification of template constraints
  - *Not* via classes or explicit interface types
  - Work with built-in types too
  - Better compile-time error messages
- Completely separates the processing of template definitions vs. instantiations
- Useful in definitions *and* at points of instantiation
- Can be combined/refined
  - A type of “inheritance”

Templates and Friends

- If *not dependent* on any template parameter
  - then that function is a friend to all specializations of the host class template
  - not very useful (can't use T, hence can't use objects of the host class)
- If a template using the *same parameter* as the class
  - then only the specialization that *matches* the class specialization is a friend
- If a *member template* (with parameter U, say)
  - then any specialization (via U) of the friend function can access *any* class specialization

A Simple Class Template

- How can we add a stream inserter?
  - `ostream& operator<<(ostream&, const Box<T>&);`

```
template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
};
```

This Won't Work!

Templates Are Different

```
template<typename T>
class Box {
   T value;
public:
   Box(const T& t) { value = t; }
   friend ostream& operator<<(ostream&, const Box<T>&);
};

template<typename T>
ostream& operator<<(ostream& os, const Box<T>& b) {
   return os << b.value;
}
```

What's the Problem?

- The inserter is *not* a template
  - But it "should be" (it uses a template argument (T))
  - This is a problem since it's not a member function
- What do we want?
  - Our **operator<<( )** must be a template
    - To be associated with a specialization of Box
  - And we want a distinct specialization for each **T**
  - But we don't want a member template!
    - That would introduce another *independent* template parameter!

Solution

- There are *special rules* for making function templates friends to class templates
  - Use angle brackets in the friend declaration
    - Can be empty if the function parameters are sufficient to deduce the template arguments
  - But… the friend must have been *previously declared*
    - As a *template*
    - This also requires a forward declaration of Box
- See Next Slide

Friend Function Templates

```
// Forward declarations
template<class T> class Box;
template<class T> ostream& operator<<(ostream&,
                                      const Box<T>&);
template<class T>
class Box {
   T value;
public:
   Box(const T& t) { value = t; }
   friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
ostream& operator<<(ostream& os, const Box<T>& b)
{
   return os << b.value;
}
```

Another Approach

- "Making New Friends"
  - Dan Saks' term
- Define body of **op<<** *in situ* (i.e.,inside of **Box**)
  - "In Situ" is also Dan Saks' term
- Such an **op<<** is *not* a template!
  - No angle brackets are used
- A new overloaded function is created for each specialization of **Box**!
- See next slide

“Making New Friends”

```
template<typename T>
class Box {
   T value;
public:
   Box(const T& t) { value = t; }
   friend ostream& operator<<(ostream& os, const Box<T>& b)
   {
      return os << b.value;
   }
};
```

Friend Templates

- We can arrange for all specializations of a function template **f( )** to befriend all specializations of **Box**
- **Note**: Only *primary templates*, and *non-templates* can be declared as friends
  - That's okay…
  - Any specializations are friends automatically!

A Friend Template

```
template<class T>
class Box {
   T value;
public:
   Box(const T& t) { value = t; }
   template<class U>
   friend void f(const Box<T>& b, const U& u) {
       cout << "u: " << u << ", b: " << b.value << endl;
   }
};

int main() {
    Box<int> b(2);          // u: 1, b: 2
    f(b, 1.0);              // u: 5, b: 1.5
    Box<double> d(1.5);
    f(d, 5);
}
```

Whew!

- We've covered most topics you'll need to know
  - And then some?!?
- There is a cost in C++ for type-safe generic programming
  - But no real runtime cost!
- C++0x will be easier to use, more powerful