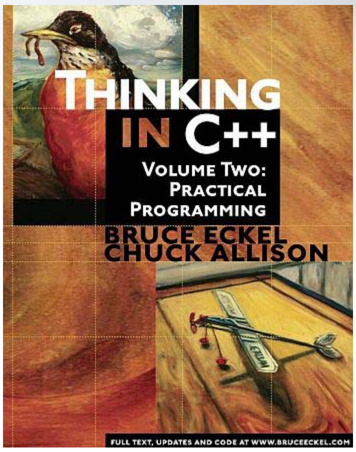
Templates in Depth

Prepared especially for **ACCU**
April 2004

From the Book…



Agenda
(We won’t have time to finish☹)

- Template Parameters
- Member Templates
- Function Template Issues
- Template Specialization
- Templates and Friends
- Template Idioms
- Template Metaprogramming
- Template Compilation Models

Template Parameters

- 3 Kinds…
- Type parameters
  – the most common (vector<int>)
- Non-type
  – compile-time integral values (**bitset<10>**, for example)
- Templates
  – “template template parameters”

Type Parameters

- The original motivation for templates
  - “Generic Programming”
- Container logic is independent of its containee’s type
- Containers of “T”:
  - vector<int> v; // **int** is a type argument
  - template<class T> // **T** is a type parameter

```
template<class T>
class vector {
    T* data;
     …
};
```

Template Instantiation

- When the compiler sees the declaration:
  vector<int> v;
  it automatically generates an “int” version of vector
  - int is substituted for T everywhere
    - just as if you had typed it that way
  - The “name” of the class is **vector<int>**
- **vector<int>**, a class, is an *instantiation* of the template “vector”
- Templates are therefore a *code generation facility*

Non-type Template Parameters

- Must be compile-time constant values
  - usually integral expressions
  - can also be addresses of static objects or functions
- Often used to place array data members on the stack
  - like with bitset
- See next slide

bitset

(from STLPort)

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

  _WordT _M_w[_Nw];
  . . .
};

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

Default Template Arguments

- Like default function arguments
  - If missing, the defaults are supplied
- template<class T = int, size_t N = 100>
  class FixedStack {
  T data[N];
  …
  };
  FixedStack<> s1; // same as Stack<int, 100>
  FixedStack<float> s2; // same as Stack<float,100>

The vector Container Declaration

- **template<class T,**
  class Allocator = allocator<T> >
  class vector;
- Note how the second parameter uses the first
- Note the space between the ‘>’s

Template Template Parameters

- 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
- Examples follow

A Simple Expandable Sequence

```
// A simple, expandable sequence
template<class T>
class Array {
  enum { INIT = 10 };
  T* data;
  size_t capacity;
  size_t count;
public:
  Array() {
    count = 0;
    data = new T[capacity = INIT];
  }
  ~Array() { delete [] data; }
  void push_back(const T& t) {...}
  void pop_back() {...}
  T* begin() { return data; }
  T* end() { return data + count; }
};
```

Passing Array as a template argument

```
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
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
}
```

Template Template Parameters and Default Arguments

- Must repeat the default argument when passing the template as an argument to another template
- Example on next slide

```
template<class T, size_t N = 10>
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;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
}
```

Passing Standard Sequence Templates as Template Arguments

- Must be aware of their default arguments
  - In particular, allocator<T>
- Example on next slide
  - Modifies **Container** to repeat allocator<T>

```
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;
  vContainer.push_back(1);
  vContainer.push_back(2);
  for(vector<int>::iterator p = vContainer.begin();
      p != vContainer.end(); ++p) {
    cout << *p << endl;
  }
```

```
  // Use a list
  Container<int, list> lContainer;
  lContainer.push_back(3);
  lContainer.push_back(4);
  for(list<int>::iterator p2 = lContainer.begin();
      p2 != lContainer.end(); ++p2) {
    cout << *p2 << endl;
  }
}
```

The typename keyword

- In certain contexts, you need to help the compiler know that an identifier represents a type
- In particular, when you want to use a *member type* from a *template parameter*
  - it assumes a name qualified by a template parameter refers to a *static member* (it has to assume *something*, since T is unknown)
  - The problem occurs when the name before the :: is a template
    - a "dependent" name
- Can also use **typename** instead of **class** in a template declaration

// A Print function for standard sequences

```
template<typename T,
         template<typename U,
                  typename = allocator<U> >
         class Seq>
void printSeq(Seq<T>& seq) {
  for(typename Seq<T>::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);
}
```

Applying a Function to a Sequence

- Many languages have an "apply" function
  - you pass it a function and a sequence (array, etc.)
  - the apply function applies the function to each element in the sequence
- C++ has **for_each( )**
- Pretend we don't know about for_each( )
  - Let's write an apply for objects
  - See next slide

An "apply" for member functions

```
template<class Cont, class PtrMemFun>
void apply(Cont& c, PtrMemFun f) {
  typename Cont::iterator it = c.begin();
  while(it != c.end()) {
    ((*it).*f)(); // Can't use it->*f
    ++it;
  }
}

class Z {
  int i;
public:
  Z(int ii) : i(ii) {}
  void g() { ++i; }
  friend ostream& operator<<(ostream& os, const Z& z) {
    return os << z.i;
  }
};
```

```
int main() {
  ostream_iterator<Z> out(cout, " ");
  vector<Z> vz;
  for(int i = 0; i < 10; i++)
    vz.push_back(Z(i));
  copy(vz.begin(), vz.end(), out);
  cout << endl;
  apply(vz, &Z::g); // Call apply()
  copy(vz.begin(), vz.end(), out);
}
```

Avoiding Parse Errors

- You’ve already seen the need to put a space between angle brackets: **template<class T, class Allocator = allocator<T> >** **class vector;**
- There are other contexts in which a ‘<‘ will be interpreted as a less-than operation, instead of the beginning of a template argument list
- The **template** keyword tells the compiler to assume that a template is being used instead of a less-than
- Example on next slide:
  - Converting a **bitset** to a **string** (needs to know the char type)
  - **bitset::to_string( )** needs a template context:

```
string s =
  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,
                               char_traits<charT>,
                               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
}
```

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**
  - Either nested classes or **typedef**s
- **Templates**
  - “Member Templates”

Member Types

Nested Classes

```
class bitset {
      class reference {
            …
      };
      reference operator[](size_t pos) {…}
};
```

If public, could say:

`bitset::reference…`

Member Types

Nested typedefs

```
template<class T,…>
class vector {
     typedef MyIteratorType iterator;
     …
};
```

Can say:

```
vector<int>::iterator p = v.begin();
```

Member Templates

- Can also nest *template definitions* inside a class
- Inside of a class template, too
- 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>

Latent Typing

- Templates use "latent typing"
  - They assume their arguments support the needed operations
  - If they don't the compiler complains
- Called latent typing because the enforcement of operations isn't through strong static typing
  - e.g., whether a template argument implements a particular interface
  - This allows non-class types such as ints to be used

Constraining Template Arguments

- 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)
```

Type Deduction in Function Templates

- Under most circumstances, the compiler deduces the argument types in a call to a function template
  - the proper version is generated automatically
- 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
  - when the template argument is a return type, and therefore cannot be deduced by the arguments
  - Example on next slide

String Conversion Function Templates

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

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

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

```
#include <complex>
#include <iostream>
#include "StringConv.h"
using namespace std;

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;
}
```

Function Template Overloading

- You can define functions with the same name as a function template
- The “best match” will be used
  - A regular function is preferred over a template
  - Can force using the template with “<>”
- You can also overload a function template by having a different number of arguments

```
template<typename 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)
}
```

Taking the Address of a Function Template Instantiation

- We like to pass functions as arguments
  - Why not pass a generated template function?
  - g(&f<int>);   // Okay if not overloaded
- Fine, unless it is overloaded
  - Which overload do you want? Ambiguity
- Examples on subsequent slides

This Won't Work -- Why?

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s("LOWER");
  transform(s.begin(), s.end(), s.begin(), tolower);
  cout << s << endl;
}
```

Answer

- <cctype> defines the single-argument version of tolower( )
- <locale> defines:
  template<class charT>
  charT tolower(charT, const locale&);
- The compiler doesn't know which one to use
- Is there a work-around?

Work-around #1

- Tell the compiler which version you want with a cast:
  transform(s.begin(), s.end(), s.begin(),
  static_cast<int (*)(int)>(tolower)
- This fails, however, if tolower( ) has C linkage
  - Some implementations do!
    - Dinkumware, for example
    - It’s a real feature
  - Transform has C++ linkage, of course

Work-around #2

- Wrap the call to transform( ) in a function that is in its own translation unit
  - Where <locale> is not included
  - Not portable
- See next slide

```
// StrTolower.cpp
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;
string strTolower(string s) {
  // Calls the single-arg tolower from <cctype>:
  transform(s.begin(), s.end(), s.begin(), tolower);
  return s;
}
```

```
// Client code
string strTolower(string);

int main() {
  string s("LOWER");
  cout << strTolower(s) << endl;
}
```

Work-around #3

- Remove the ambiguity with a wrapper template that calls the single-arg version explicitly:

```
#include <algorithm>
#include <cctype>
using namespace std;

template<class charT>
charT strToLower(charT c) {
  return tolower(c);
}
…
transform(s.begin(),s.end(),s.begin(),&strToLower<char>);
```

Partial Ordering of Function Templates

- With overloaded templates, there needs to be a way to choose the "best fit"
- Plain functions are always considered better than templates
  - why generate another function when an existing one will suffice?
- Some templates are better than others also
  - more "specialized" if matches more combinations of arguments types than another
  - Example follows

```
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*
}
```

Template Specialization

- A template by nature is a *generalization*
- It becomes specialized for a particular use when we specify the actual template arguments of interest
- A particular instantiation is therefore a *specialization*

Explicit Specialization

- The template facility specializes a template for your use when you instantiate it
  - but it uses the “primary” template to do it
- What if you want special “one-off” behavior for certain combinations of template parameters?
- You can provide custom code for specializations
  - both *full* or *partial* specializations for class templates
  - the compiler will use your versions instead of what it would have generated
- Full specialization uses the **template<>** syntax

Explicit Specialization of Function Templates

- It is done, but you can always just provide a plain function to do the job
- Nonetheless, see the next slide
  - And note correct use of **const**!

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;
  cout << min<>(s1, s2) << endl;
}
```

Explicit Specialization of Class Templates

- Example: vector<bool>
  - packs bits, like bitset does (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, allocator<bool> >
{…};
```

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”
    - equality
- 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” 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();
}
```

iterator_traits

- A technique for endowing naked pointers with:
  - difference_type
  - value_type
  - pointer
  - reference
  - iterator_category
- Uses partial specialization

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 print(Iter b, Iter e) {
typedef typename iterator_traits<Iter>::value_type T;
  copy(b, e, ostream_iterator<T>(cout, "\n"));
}
```

Preventing Template Code Bloat

- Templates generate distinct classes/functions for each unique combination of template arguments
  - A Good Thing
- Code overhead can be lessened for containers:
  - All pointer specializations can share code
  - Fully specialize on void*
  - Partially specialize on T*, using void*'s code
    - Each partial specialization is a thin veneer
  - See next slide

Sharing void*'s Code

```
template<class T> class Stack {
   // Defines the complete primary template
};

template<> class Stack<void *> {
   // Fully specializes Stack for pointers
   // (Can optimize copy semantics with memcpy)
};

// Partial specialization for pointers (small footprint)
template<class T>
class Stack<T*> : protected Stack<void *> {
   // Forwards all functions to Stack<void*>
};
```

Type Tricks with Specialization

- Discover whether types are the same
  - Example: IsSame.cpp
- Discover whether a type is a template
  - Example: IsTemplate.cpp
- (These examples are from Steve Dewhurst)
- More such neat stuff in Metaprogramming section

```cpp
// IsTemplate.cpp {-bor}
#include <iostream>
#include <typeinfo>
#include <vector>
using namespace std;

template<typename T>
struct IsTemplate {
   enum { numargs = 0 };
};

template<template<typename> class X, typename T>
struct IsTemplate< X<T> > {
   enum { numargs = 1 };
   typedef T type;   // extract T!
};

template<template<typename, typename> class X, typename T1,
typename T2>
struct IsTemplate< X<T1, T2> > {
   enum { numargs = 2 };
   typedef T1 type1;
   typedef T2 type2;
};
```

```cpp
// A test template
template<typename 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<>
*/
```

Name Taxonomy

- Names can be nicely dichotomized as:
  - Unqualified
    - x, f( )
  - Qualified (provides 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

Name Lookup Outside of Templates

- Normally found by searching enclosing scopes
  - Inside-out
  - Local, class, base class, namespace/global
- Special case:
  std::string s(…);
  std::cout << s << std::endl;
- How is **operator<<(ostream&, const string&)** found?
  - It wasn't explicitly requested

Argument-Dependent Lookup

- Also known as "Koenig" Lookup
- To find the scope of definition of a function, the namespace (or class) of each argument type is searched
- Motivated by operator overloading
- Since "s" was a std::string, and cout is a std::ostream, search for operator<<( ) in std
- Ambiguities can still result, of course
- Can disable ADL with parentheses: (f)( );

ADL "Gotcha"

- We have a class named Thesaurus (Chapter 7)
- There is an associated stream inserter for a Thesaurus Entry (class TEntry):
  - ostream& operator<<(ostream&, const TEntry&);
- In main( ) we call:
  std::copy(thesaurus.begin( ), thesaurus.end( ),
  ostream_iterator<TEntry>(cout, "\n"));
  - Which calls the operator<<( ) above
- All of these arguments are in std…
  - ADL kicks in and ignores the global namespace!

Work-around

```
namespace std {
   ostream& operator<<(ostream&  os, const TEntry& t)
   {
      os << t.first << ": ";
      copy(t.second.begin(), t.second.end(),
           ostream_iterator<string>(os, " "));
      return os;
   }
}
```

- Technically illegal!
- Every compiler supports it!
  - What choice do they have?!

Template Name Lookup Issues

- Some things cannot be resolved when the compiler first encounters a template definition
  - Anything depending on a template parameter
- The compiler must wait until instantiation time to resolve those issues
  - When template arguments are supplied
- 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
- Template instantiation time
  - Dependent names are resolved
  - Which specialization to use is determined
- 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();
}
```

Answer

- f( ) is a non-dependent name, so it is looked up when the template code is parsed
- f(double) is in scope at the time, so the call resolves to that function, not f(int)
- Most compilers do the wrong thing here
  - For historical reasons (the rules were decided on late in the game)

A Second Example

```
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();
  }
};
```

Discussion

- If the calls to the inherited member functions (**begin**, **end**, **front**, etc.) were *unqualified*, they would be looked up *early*
- Since the base class is dependent on a template parameter, the base class can't be searched at template definition time
  - Therefore, the inherited functions would not be found
- By qualifying the calls, the fact that the type of **this** (PQV) has a dependent base class causes lookup to be delayed until instantiation time.

Templates and Friends

- If a **friend** declaration inside of a class template is:
  - not dependent on any template parameter
    - then that function is a friend to all specializations of the host class template
  - a template using the same parameter as the class
    - then only the specialization that matches the class specialization is a friend
  - a member template (with parameter U, say)
    - then any flavor of the function can access any class version

```
// A Non-template
class Friendly {
  int i;
public:
  Friendly(int theInt) { i = theInt; }
  friend void f(const Friendly&); // Needs global def.
  void g() { f(*this); }
};

void h() {
  f(Friendly(1));  // Uses ADL
}

void f(const Friendly& fo) {  // Definition of friend
  cout << fo.i << endl;
}

int main() {
  h(); // Prints 1
  Friendly(2).g(); // Prints 2
}
```

Now Make Friendly a Template

- We want a separate friend for each specialization of Friendly
- One solution:
  - Make f( ) a template
  - But it's not as easy as you might think
  - See next slide

A friend Function Template

```
// Necessary forward declarations:
template<class T> class Friendly;
template<class T> void f(const Friendly<T>&);

template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f<>(const Friendly<T>&); // or <T>
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

template<class T> void f(const Friendly<T>& fo) {
  cout << fo.t << endl;
}
```

Another Approach

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

Making New Friends

```
template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f(const Friendly<T>& fo) { // in situ
    cout << fo.t << endl;
  }
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}
```

Friend Templates

- We can arrange for all specializations of f<>( ) to befriend all specializations of Friendly
- Use a member template:

```
template<class T> class Friendly {
  template<class U>
    friend void f<>(const Friendly<U>&);
  …
};
```

Template Programming Idioms

- Traits
- Policies
- The Curiously Recurring Template Pattern (CRTP)

Traits

- Traits usually hold *data* for chosen specializations of a template
  - **char_traits** contains member *functions*, but only to accommodate different overloads based on character type (strcmp( ) vs. wcscmp( )) – the functionality is basically the same for the two flavors of character
- Other examples follow

Character Traits

- Strings are templates
  - template<class charT, class traits = char_traits<charT>, class allocator = allocator<charT> >
    class basic_string;
  - typedef basic_string<char> string;
- char_traits allow you to customize character-based operations
  - Can introduce case-insensitive searching once and for all
  - Can process Unicode characters
- Example: ICompare.cpp, IWCompare.cpp

```
struct ichar_traits : char_traits<char> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(char c1st, char c2nd) {
    return toupper(c1st) == toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return !eq(c1st, c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return toupper(c1st) < toupper(c2nd);
  }
  static int compare(const char* str1,
    const char* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(tolower(*str1) < tolower(*str2))
        return -1;
      else if(tolower(*str1) > tolower(*str2))
        return 1;
      assert(tolower(*str1) == tolower(*str2));
      str1++; str2++; // Compare the other chars
    }
    return 0;
  }
```

```
static const char* find(const char* s1,
    size_t n, char c) {
    while(n-- > 0)
      if(toupper(*s1) == toupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const istring& s) {
  return os << string(s.c_str(), s.length());
}
#endif // ICHAR_TRAITS_H  ///:~
```

```
//: C03:ICompare.cpp
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first << endl;
  cout << second << endl;
  assert(first.compare(second) == 0);
  assert(first.find('h') == 1);
  assert(first.find('I') == 2);
  assert(first.find('x') == string::npos);
} ///:~
```

```
struct iwchar_traits : char_traits<wchar_t> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) == towupper(c2nd);
  }
  static bool ne(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) != towupper(c2nd);
  }
  static bool lt(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) < towupper(c2nd);
  }
  static int compare(const wchar_t* str1,
    const wchar_t* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(towlower(*str1) < towlower(*str2))
        return -1;
      else if(towlower(*str1) > towlower(*str2))
        return 1;
      assert(towlower(*str1) == towlower(*str2));
      str1++; str2++; // Compare the other wchar_ts
    }
    return 0;
  }
```

```
typedef basic_string<wchar_t, iwchar_traits> iwstring;

inline wostream& operator<<(wostream& os,
  const iwstring& s) {
  return os << wstring(s.c_str(), s.length());
}
#endif // IWCHAR_TRAITS_H  ///:~
```

```
//: C03:IWCompare.cpp
//{-g++}
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  iwstring wfirst = L"tHis";
  iwstring wsecond = L"ThIS";
  wcout << wfirst << endl;
  wcout << wsecond << endl;
  assert(wfirst.compare(wsecond) == 0);
  assert(wfirst.find('h') == 1);
  assert(wfirst.find('I') == 2);
  assert(wfirst.find('x') == wstring::npos);
} ///:~
```

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();
...
```

Traits at Bear Corner

```
// Item classes (traits of guests):

// Beverage types:
class Milk {};
class CondensedMilk {};

// Snack types:
class Honey {};
class Cookies {};

// Guest classes:
class Bear {};
class Boy {};
```

Traits at Bear Corner

```
// Primary traits template (could hold common types)
template<class Guest> class GuestTraits;

// Traits specializations for Guest types
template<> class GuestTraits<Bear> {
public:
  typedef CondensedMilk beverage_type;
  typedef Honey snack_type;
};

template<> class GuestTraits<Boy> {
public:
  typedef Milk beverage_type;
  typedef Cookies snack_type;
};
```

Traits at Bear Corner

```
// A custom traits class
class MixedUpTraits {
public:
  typedef Milk beverage_type;
  typedef Honey snack_type;
};

// The Guest template (uses a traits class)
template<class Guest, class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << "Entertaining " << theGuest
         << " serving " << bev
         << " and " << snack << endl;
  }
};
```

Traits at Bear Corner

```
int main() {
  Boy cr;
  BearCorner<Boy> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear> pc2(pb);
  pc2.entertain();
  BearCorner<Bear, MixedUpTraits> pc3(pb);
  pc3.entertain();
}
```

/* Output:
Entertaining Patrick serving Milk and Cookies
Entertaining Theodore serving Condensed Milk and Honey
Entertaining Theodore serving Milk and Honey
***/**

Policies

- Similar to traits, but the emphasis is on associating *functionality* with a template parameter
  - not data
- Pioneered by Andrei Alexandrescu
  - *Modern C++ Design*

Policies at Bear Corner

```
// Policy classes (require a static doAction() function):
class Feed {
public:
  static const char* doAction() { return "Feeding"; }
};

class Stuff {
public:
  static const char* doAction() { return "Stuffing"; }
};
```

Policies at Bear Corner

```
// The Guest template (uses a policy and a traits class)
template<class Guest, class Action,
         class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << Action::doAction() << " " << theGuest
         << " with " << bev
         << " and " << snack << endl;
  }
};
```

Policies at Bear Corner

```
int main() {
  Boy cr;
  BearCorner<Boy, Feed> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear, Stuff> pc2(pb);
  pc2.entertain();
}
```

Counting Objects

- Can use a static data member that tracks the object count
- Constructors increment
- Destructors decrement

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
- The non-template solution for code sharing is *inheritance*
- See next slide

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 count for each class to be counted
- We need to inherit from a *different class* for each client class
- Hence, we need to combine *both* inheritance and template (parametric) polymorphism
- See next slide

A Template Counter Solution

```
template<class T> class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted<T>&) { ++count; }
  ~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>
{};
```

The 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.

Singleton via CRTP

- Inherit Singleton-ness
- Uses Meyers' static singleton object approach
  - Since nothing non-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;
}
```

Template Metaprogramming

- Compile-time Computation!
  - whoa!
- Has been proven to be “Turing complete”
  - means that theoretically, you can do *anything* at compile time
  - reminiscent of functional programming
  - in practice, it’s used for custom code generation, compile-time assertions, and numerical libraries

The "Hello World" Examples for TMP

- Factorial.cpp
- Power.cpp
- Accumulate.cpp
- Loops are done by recursion
- Decisions done by the ternary operator ?: or by partial specialization
- Remember, only compile-time constants (ints, types) can be used

Compile-time Factorial

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

template<int n> struct Factorial {
  enum { val = Factorial<n-1>::val * n };
};

template<> struct Factorial<0> {
  enum { val = 1 };
};

int main() {
  cout << Factorial<12>::val << endl;
  // Prints 479001600
}
```

Compile-time Exponentiation

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

template<int N, int P> struct Power {
  enum { val = N * Power<N, P-1>::val };
};

template<int N> struct Power<N, 0> {
  enum { val = 1 };
};

int main() {
  cout << Power<2, 5>::val << endl;  // 32
}
```

Compile-time Accumulation

From Eisenecker and Czarnecki

```
// Accumulates the results of F(0)..F(n)
template<int n, template<int> class F> struct Accumulate
{
  enum { val = Accumulate<n-1, F>::val + F<n>::val };
};

// The stopping criterion (returns the value F(0))
template<template<int> class F> struct Accumulate<0, F>
{
  enum { val = F<0>::val };
};
```

```
// Various "functions":
template<int n> struct Identity {
  enum { val = n };
};

template<int n> struct Square {
  enum { val = n*n };
};

template<int n> struct Cube {
  enum { val = n*n*n };
};

int main() {
  cout << Accumulate<4, Identity>::val << endl; // 10
  cout << Accumulate<4, Square>::val << endl;   // 30
  cout << Accumulate<4, Cube>::val << endl;     // 100
}
```

Static Assertions

- Uses a combination of macro magic, local classes, and full specialization to obtain a “meaningful” message when a compile-time assertion fails
- More expressive than the widely used:

```
#define STATIC_ASSERT(x) \
     do { typedef int a[(x) ? 1 : -1]; } while(0)
```

Alexandrescu's Static Assertions

```
// A template and a specialization
template<bool> struct StaticCheck {
  StaticCheck(...);  // Universal implicit conversion
};

template<> struct StaticCheck<false> {};

// The macro (generates a local class)
#define STATIC_CHECK(expr, msg) {             \
  class Error_##msg {};                       \
  sizeof((StaticCheck<expr>(Error_##msg()))); \
}
```

Applying STATIC_CHECK

```
// Detects narrowing conversions
template<class To, class From> To safe_cast(From from)
{
  STATIC_CHECK(sizeof(From) <= sizeof(To),
               NarrowingConversion);
  return reinterpret_cast<To>(from);
}

int main() {
  void* p = 0;
  int i = safe_cast<int>(p);
  cout << "int cast okay" << endl;
  //! char c = safe_cast<char>(p);
}

/* Error message when last line is uncommented:
Cannot cast from 'Error_NarrowingConversion' to 'StaticCheck<0>' in
function char safe_cast<char,void *>(void *)
*/
```

Had Enough?

- We're skipping:
  - Expression Templates
    - a certain form of metaprogramming
    - heavily used in professional math libraries
  - Explicit Instantiation
    - a way of controlling what get instantiated when
  - Template Compilation models
    - Inclusion
      - everything in header files
    - Separation
      - a small degree of separation
  - It's all in the book