C++ Programming
~ Generic Programming ~

Prepared for Ingenix, Inc.

Copyright 2004, Fresh Sources, Inc.

Generic Programming

- Code with “type parameters”
  - “Parametric Polymorphism”
- Allows type-independent code to be written once
  - Like vector, list, etc.
- C++ templates are type-safe
  - Type-specific code is generated from templates on demand
- Enables extremely high-level programming

Agenda

- Function Templates
- Class Templates
- Template Parameters
- Explicit Template Specialization
- Generic Algorithms
- Function objects and adapters
- Generic Containers
- Iterators
- Volume 1: 16; Volume 2: 5-7

Before we begin

- Let’s create FixedStack together
  - holds ints in an array on the heap
  - stack size passed as a constructor parameter
  - member functions push, pop, top, size
  - just use asserts for pre-conditions for now

Consider making FixedStack hold doubles

- What changes are necessary?

Now make FixedStack a template

- Change int to the template parameter (T) where appropriate
- For practice, move member functions outside the class

Template Parameters

- 3 Kinds:
  - type
    - the most common
  - non-type
    - integer values (bitset, for example)
  - templates
    - "template template parameters"

Type Parameters

- The original motivation for templates
- Container logic is independent of its containee's type
- Containers of "T":
  - vector<int> v; // int is a type
  - template<class T> // T is a type parameter
    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 global objects or functions
    - rarely used
- Often used to place array data members on the stack

Exercise

- Make the stack size for FixedStack a nontype template parameter instead of a constructor parameter
- This allows you to place the underlying array on the stack
  - so do it already!

Default Template Arguments

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

The vector Container Declaration

- template<class T, class Allocator = allocator<T> >
  class vector;
- Note how the second parm uses the first
- allocator<T> is usually ignored, by the way
- Note the space between the '>'s

Exercise-lette

- Add a default argument for FixedStack to make its size default to 10
- Instantiate in main( ) a FixedStack that uses the default value

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
- Example: TempTemp.cpp, TempTemp2.cpp, TempTemp4.cpp

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
- I bet this doesn't make a lick of sense
- See TypenameID.cpp, PrintSeq.cpp
- Can also use **typename** instead of class in a template declaration
  - Example: UsingTypename.cpp

Question

- What kind of thing can you define as a member of a class?

Answer

- variables (i.e., fields; both static or non-static)
- functions (i.e., methods, both static and non-static)
- types (nested classes or typedefs)

Member Types
Nested Classes

```
class bitset {
        class reference {
                …
        };
        reference operator[](int 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>&);
- Example: MemberClass.cpp

Type Deduction of 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: StringConvTest.cpp

Function Template Overloading

- You can define functions with the same name as a function template
- The “best match” will be used
- You can also overload a function template by having a different number of arguments
- Example: MinTest.cpp
  - and the line further down in the book with 3 parms

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 do?
- Some templates are better than others also
  - more “specialized” if matches more combinations of arguments types than another
  - Example: PartialOrder.cpp

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 template to do it
- What if you want special "one-off" behavior for a certain combination 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 MinTest2.cpp
  - And note proper use of const!

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> is (sort of) defined as:
  template<> class vector<bool, allocator<bool> >
  {...};

Partial Specialization of Class Templates

- Can special on a subset of template arguments
  - leaving the rest unspecified
  - can also specialize on pointers
- vector<bool> is actually a partial specialization: template<class Allocator> class vector<bool, Allocator> { }; // allocator open
- The "most specialized" match is preferred
- Example: PartialOrder2.cpp, Sortable.cpp

Template Programming Idioms

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

CRTP

- A class inherits from a template that specializes on it (the first one)
- What???
- class T : public X<T> {…};
- Only valid if the size of X<T> can be determined independently of T.
- Example: CountedClass*.cpp, C10:CuriousSingleton.cpp

Template Metaprogramming

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

The “Hello World” Examples for TMP

- FactorialTM.cpp
- FibonacciTM.cpp
- PowerTM.cpp
- AccumulateTM.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

Other Metaprogramming Uses

- Loop Unrolling
  - a common numerical programming practice
  - eliminates iteration overhead via inlining
  - Example Unroll.cpp
- Compile-time Assertions
  - only for things that be evaluated at compile time
  - Examples: Conditionals.cpp, StaticAssert1-2.cpp

Exercise

- 1, 2, 3 (if we have time)

Generic Algorithms

- Templates
- Can process homogeneous sequences of *any* type
  - arrays, vectors, lists, anything that meets STL requirements
    - must provide iterators
    - pointers are iterators
- Lots of them!

A First Look

- CopyInts.cpp
- CopyStrings.cpp
- How does copy( ) work?

A First Try at copy( )

```
template<typename T>
void copy(T* begin, T* end, T* dest) {
  while(begin != end)
    *dest++ = *begin++;
}
```

A Vector example

- vector iterators are necessarily not real pointers
- But CopyVector.cpp works!

A Second Try at copy( )

- ```
  template<typename Iterator>
  void copy(Iterator begin, Iterator end, Iterator dest) {
    while(begin != end)
      *begin++ = *dest++;
  }
  ```
- The compiler infers the actual type of iterator
- As long as it supports !=, ++, and *, all is well!
  ```

Non-mutating Algorithms

```
for_each            count_if
find                mismatch
find_if             equal
find_first_of       search
adjacent_find       find_end
count               search_n
```

Mutating Algorithms

```
transform
copy
copy_backward
swap
iter_swap
swap_ranges
replace
replace_if
replace_copy
replace_copy_if
fill
fill_n
```

```
generate
generate_n
remove
remove_if
remove_copy
remove_copy_if
unique
reverse
reverse_copy
rotate
rotate_copy
random_shuffle
```

Ordering Algorithms

Sorting

sort
stable_sort
partial_sort
partial_sort_copy
nth_element
merge
inplace_merge
partition
stable_partition

Set Operations

includes
set_union
set_intersection
set_difference
set_symmetric_difference

Heap Operations

push_heap
pop_heap
make_heap
sort_heap

Ordering Algorithms continued...

Searching

binary_search

lower_bound

upper_bound

equal_range

Permutations

next_permutation

prev_permutation

Min/max

min

max

min_element

max_element

lexicographical_compare

Predicates

- Functions that return a **bool**
- Many algorithms have alternate versions that apply predicates to sequence elements
  - for selection, deletion, etc.
- Examples:
  - CopyInts2.cpp
  - CopyStrings2.cpp
  - ReplaceStrings.cpp

Stream Iterators

- Facilitates reading/writing a sequence from/to a stream
  - without you doing a loop explicitly
- ostream_iterator<T>(ostream&, const string& sep)
  - Examples: CopyInts3.cpp, CopyIntsToFile.cpp
- istream_iterator<T>(istream&)
  - Example: CopyIntsFromFile.cpp

Function Objects

- Any class that overloads operator( )
- Can take any number of arguments
- Typically unary or binary
- Example: GreaterThanN(2).cpp

Standard Function Objects

Predicates

`equal_to`
`not_equal_to`
`greater`
`less`
`greater_equal`
`less_equal`
`logical_and`
`logical_or`
`logical_not`

Arithmetic

`plus`
`minus`
`multiplies`
`divides`
`modulus`
`negate`

Negaters

`unary_negate`
`binary_negate`

Member-related

`mem_fun_t`
`mem_fun1_t`
`mem_fun_ref_t`
`mem_fun1_ref_t`

Binders

`binder1st`
`binder2nd`

Pointer-related

`pointer_to_unary_function`
`pointer_to_binary_function`

Using a Standard Function Object

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

int main() {
  greater<int> g;
  cout << g(3, 4) << endl;  // Prints 0 (for false)
  cout << g(5, 4) << endl;  // Prints 1 (for true)
}
```

Function Object Adaptors

- Allow combining function objects in useful ways
- Binders, for example:
  - allow you to treat a binary function as a unary function by fixing (binding a fixed value to) one of the parameters
  - bind2nd( ) creates a function object that stores the function and the fixed 2nd argument
  - It overloads operator( ) so you can provide the missing first argument
  - Examples: CopyInts4.cpp, CountNotEqual.cpp, FBinder.cpp

Exercises

- Write an expression with standard function objects and adapters that searches a sequence of integers for the first element that is not less than 100.
- Exercise 3, 10 (in book)

All Containers...

- Are *homogeneous*
- Provide `insert` and `erase` capability
  - grow as needed
- Support the following methods:

```
size_type size() const;
size_type max_size() const;
bool empty() const;
```

Standard Containers

- Sequences
  - `vector, deque, list`
- Container Adapters
  - `queue, stack, priority_queue`
- Associative Containers
  - `set, multiset, map, multimap`

A First Look

- A set example
  - IntSet.cpp
  - WordSet.cpp
- A vector example:
  - StringVector.cpp

Sequences

- **`vector`**
  - random access
  - optimal insertion/deletion at end (**`push_back`**)
- **`deque`**
  - random access
  - optimal insertion/deletion at both ends (**`push_front`**)
- **`list`**
  - sequential access only
  - doubly-linked; optimal insertion/deletion anywhere

All Sequences support...

```
void resize(size_type, T = T());
T& front() const;
T& back() const;
void push_back(const T&);
void pop_back();
```

Restricted Sequence Functions

```
// deque and list only:
void push_front(const T&);
void pop_front();


//deque and vector only:
T& at(size_type n);
```

Nagging Questions

- How does one navigate a `list`?
  - It doesn’t have any traversal member functions!
- How does on use the standard algorithms on a sequence?
  - Don’t tell us that all the algorithms have to be repeated as member functions!
- The answer is…

Iterators

- Generalization of a pointer
- Overload at least `operator!=`, `operator==`, `operator*`, `operator++`, `operator->`
- Some overload `operator--`, `operator[]`

Traversing a List

```
list<T> lst;
…
// insert some elements, then:
list<T>::iterator p = lst.begin();
while (p != lst.end())
{
    // Process current element (*p), then:
    ++p;
}
```

- All sequences provide members **`begin`** and **`end`**

Implementing `find`

```
template<class Iterator, class T>
Iterator
find(Iterator start, Iterator past, const T& v)
{
    while (start != past)
    {
        if (*start == v)
            break;
        ++start;
    }
    return start;
}
```

- All algorithms are implemented in terms of *iterators*

Iterator Taxonomy

- Input
- Output
- Forward
- Bi-directional
- Random Access

Input Iterators

- Read-only access to elements
- Single-pass, forward traversal
- `find` expects an Input Iterator

The *Real* Implementation of `find`

(Documentation change only)

```
template<class InputIterator, class T>
InputIterator
find(InputIterator start, InputIterator past,
     const T& v)
{
    while (start != past)
    {
        if (*start == v)
            break;
        ++start;
    }
    return start;
}
```

Output Iterators

- Write-only access to elements
- Single-pass, forward traversal
- Example: **`ostream_iterator:`**

```
copy(a, a+n,
     ostream_iterator<int>(cout," "));
```

Forward Iterators

- Both read and write access
  - can therefore substitute for Input or Output Iterators
- Multiple-pass forward traversal
- **`unique`** expects a Forward Iterator

```
list<T>::iterator p = unique(lst.first(),
                             lst.end());
```

Bi-directional Iterators

- Can do everything a Forward Iterator can
- Also support backwards traversal
  - `operator--()`
  - `operator--(int)`
- `reverse` requires a Bi-directional Iterator

Traversing a List Backwards

```
list::iterator p<T> = lst.end();
while (p > lst.begin())
{
    --p;            // "advances" backwards

    // process *p, then:
    if (p == lst.begin())
        break;
}
```

A Better Way
Reverse Iterators

```
list<T>::reverse_iterator p = lst.rbegin();
while (p != lst.rend())
{
    // process *p, then:
    ++p;           // “advances” backwards
}
```

Random Access Iterators

- Support Pointer Arithmetic in constant time
  - **`operator+, +=, -, -=, [], <, <=, >, >=`**
- **`sort`** expects a Random Access Iterator

How do you Sort a List?

- Doesn't provide a Random Access Iterator
- Generic `sort` will fail on a `list`
- Provides its own `sort` member function
- Also `merge`, `remove`, and `unique`

What's Wrong with this Picture?

```
vector<int> v1;
…
// fill v1, then:
vector<int> v2;
copy(v1.begin(), v1.end(), v2.begin());
```

Iterator Modes

- Iterators work in *overwrite* mode by default
- Need an *insert* mode for cases like above
  - that calls appropriate, underlying insert operation

Insert Iterators

- Replace output calls (`operator*`, `operator=`, etc.) with appropriate *insert* function
- `back_insert_iterator`
  - calls `push_back`
- `front_insert_iterator`
  - calls `push_front`
- `insert_iterator`
  - calls `insert`

Helper Functions

- **`back_inserter`**
  - creates a `back_insert_iterator`
- **`front_inserter`**
  - creates a `front_insert_iterator`
- **`inserter`**
  - creates an `insert_iterator`

Insert Iterator Example

```
vector<int> v1;
…
// fill v1, then:
vector<int> v2;
copy(v1.begin(), v1.end(), back_inserter(v2));
```

Stream Iterators

- `ostream_iterator`
  - an Output Iterator

```
copy(v1.begin(), v1.end(),
        ostream_iterator<int>(cout, " 
"));
```

- `istream_iterator`
  - an Input Iterator

```
copy(istream_iterator<int>(cin),
       istream_iterator<int>(),
       back_inserter(v1));
```

Container Adapters

- High-level abstract data types
  - `queue, stack, priority_queue`
- Use a sequence for implementation

```
stack< string, vector<string> > myStack;
```

- `stack` & `queue` use `deque` by default
- `priority_queue` uses a `vector`
- No iterators are provided
  - more restricted interface

Associative Containers

- **`set`**
  - stores unique elements
  - test for membership
  - **`multiset`** allows duplicates
- **`map`**
  - stores <key, value> pairs
  - keys must be unique
  - **`multimap`** allows duplicate keys
- Support fast (logarithmic), key-based retrieval
- Stored according to an ordering function
  - **`less<T>()`** by default
  - can use as a sequence

Set Example

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

void main()
{
    // Populate a set:
    set<string> s;
    s.insert("Alabama");
    s.insert("Georgia");
    s.insert("Tennessee");
    s.insert("Tennessee");
```

```
    // Print it out:
    set<string>::iterator p = s.begin();
    while (p != s.end())
        cout << *p++ << endl;
    cout << endl;

    // Do some searches:
    string key = "Alabama";
    p = s.find(key);
    cout << (p != s.end() ? "found " : "didn't find ")
         << key << endl;

    key = "Michigan";
    p = s.find(key);
    cout << (p != s.end() ? "found " : "didn't find ")
         << key << endl;
}
```

```
// Output:
Alabama
Georgia
Tennessee

found Alabama
didn't find Michigan
```

Map Example

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

void main()
{
    // Convenient typedefs:
    typedef map<string, string, greater<string>()>
            map_type;
    typedef map_type::value_type element_type;
```

```
// Insert some elements (two ways):
map_type m;
m.insert(element_type(string("Alabama"),
         string("Montgomery")));
m["Georgia"] = "Atlanta";
m["Tennessee"] = "Nashville";
m["Tennessee"] = "Knoxville";

// Print the map:
map_type::iterator p = m.begin();
while (p != m.end())
{
    element_type elem = *p++;
    cout << '{' << elem.first << ','
         << elem.second << "}\n";
}
cout << endl;
```

```
    // Retrieve via a key:
    cout << '"' << m["Georgia"] << '"' << endl;
    cout << '"' << m["Texas"] << '"' << endl;
}

// Output:
{Tennessee,Knoxville}
{Georgia,Atlanta}
{Alabama,Montgomery}

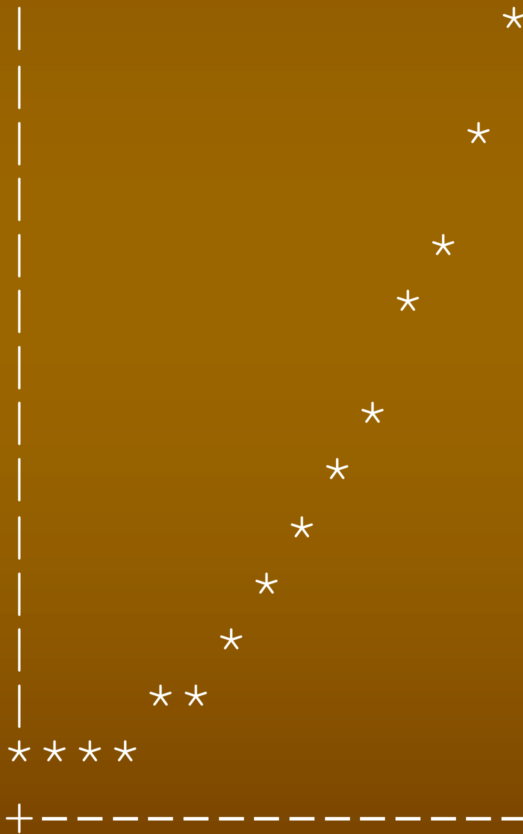
"Atlanta”
""
```

Applications

Grid
A Sample Application

- Graphs `y = f(x)`
- Models the `x-y` plane as rows of characters
- In other words, a vector of vectors of char
- Illustrates reverse iterators, random acccess iterators

Grid Output



Grid

Source Code

```
// grid.cpp: Stores an x-y graph in a grid of chars
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;

namespace
{
    typedef vector<char> row_type;
    typedef vector<row_type> grid_type;
    const int xmax = 15;     // # of columns
    const int ymax = xmax;   // # of rows
}
```

```
int main()
{
    void print_grid(const grid_type&);
    double f(double);          // Function to graph
    grid_type grid;            // A vector of rows

    // Draw y-axis and clear 1st quadrant:
    grid.reserve(ymax);
    row_type blank_row(xmax);
    blank_row[0] = '|';
    for (int y = 0; y < ymax; ++y)
        grid.push_back(blank_row);

    // Draw x-axis:
    grid[0][0] = '+';
    for (int x = 1; x < xmax; ++x)
        grid[0][x] = '-';
```

```
    // Populate with points of f():
    for (int x = 0; x < xmax; ++x)
        grid[f(x)][x] = '*'; // row-oriented

    print_grid(grid);
}

double f(double x)
{
    // NOTE: Contrived to fix within grid!
    return x * x / ymax + 1.0;
}
```

```
void print_grid(const grid_type& grid)
{
    grid_type::const_reverse_iterator yp;
    for (yp = grid.rbegin(); yp != grid.rend(); ++yp)
    {
        // Print a row:
        copy(yp->begin(), yp->end(),
             ostream_iterator<char>(cout, ""));
        cout << endl;
    }
}
```

Duplicate Line Filter

- Like UNIX's `uniq`
- Except it processes *unsorted* files
- Need a way to detect duplicate lines without disturbing original line order

Sample Data

Input

```
each
peach
pear
plum
i
spy
tom
thumb
```
`tom`
`thumb`
```
in
the
cupboard
```
`i`
`spy`
```
mother
hubbard
```

Output

```
each
peach
pear
plum
i
spy
tom
thumb
in
the
cupboard
mother
hubbard
```

uniq2

- What data structure(s)?
  - Why won’t a `set` suffice?
- Accesses lines through an *index*
- Sorts the index based on line content, then removes adjacent duplicates with the unique algorithm
- Re-sorts the surviving indexes numerically to adhere to original order

uniq2.cpp

```
(#includes omitted to save space)

namespace
{
    vector<string> lines;

    // Sort Predicates:
    bool less_by_idx(int a, int b)
    {
        return lines[a] < lines[b];
    }

    bool equal_by_idx(int a, int b)
    {
        return lines[a] == lines[b];
    }
}
```

```
int main()
{
    vector<int> idx;

    // Read lines into memory:
    string line;
    int nlines = 0;
    for ( ; getline(cin, line, '\n'); ++nlines)
    {
        lines.push_back(line);
        idx.push_back(nlines);  // Identity index
    }
```

```
    stable_sort(idx.begin(), idx.end(), less_by_idx);

    // Remove indexes to duplicate lines:
    vector<int>::iterator uniq_end =
        unique(idx.begin(), idx.end(), equal_by_idx);

    // Restore correct order of remaining lines:
    sort(idx.begin(), uniq_end);

    // Output unique lines:
    int nuniq = uniq_end - idx.begin();
    for (int i = 0; i < nuniq; ++i)
        cout << lines[idx[i]] << endl;
}
```

Exercise

- Write the cross-reference program from scratch without looking back at the code you saw on the first day!
- Extra credit: have your map ignore case in processing keys!