

Object – Oriented Programming

Week 5, Spring 2009

Reference & Templates

Weng Kai

<http://fm.zju.edu.cn>

Wednesday, Mar. 18, 2009

Contents

- Reference
- Template

Declaring references

- References are a new data type in C++

- `char c; // a character`

- `char* p = &c; // a pointer to a character`

- `char& r = c; // a reference to a character`

- Local or global variables

- `type& refname = name;`

- For ordinary variables, the initial value is required

- In parameter lists and member variables

- `type& refname`

- Binding defined by caller or constructor

References

- Declares a new *name* for an *existing* object

```
int    X = 47;  
int& Y = X;    // Y is a reference to X  
  
// X and Y now refer to the same variable  
cout << "Y = " << Y;    // prints Y = 47  
Y = 18;  
cout << "X = " << X;    // prints X = 18
```

Rules of references

- References must be initialized when defined
- Initialization establishes a binding

- In declaration

```
int x = 3;  
int& y = x;  
const int& z = x;
```

- As a function argument

```
void f ( int& x );  
f(y);      // initialized when function is called
```

Rules of references

- Bindings don't change at run time, unlike pointers
- Assignment changes the object referred-to

```
int& y = x;  
y = 12; // Changes value of x
```

- The target of a reference must have a location!

```
void func(int &);  
func (i * 3);      // Warning or error!
```

- Example: Reference.cpp

Pointers vs. References

- References
 - can't be null
 - are dependent on an existing variable, they are an alias for an variable
 - can't change to a new "address" location
- Pointers
 - can be set to null
 - pointer is independent of existing objects
 - can change to point to a different address

Restrictions

- No references to references
- No pointers to references

```
int&* p;           // illegal
```

–Reference to pointer is ok

```
void f(int*& p);
```

- No arrays of references

Templates

Why templates?

- Suppose you need a list of X and a list of Y
 - The lists would use similar code
 - They differ by the type stored in the list
- Choices
 - Require common base class
 - May not be desirable
 - Clone code
 - preserves type-safety
 - hard to manage
 - Untyped lists
 - type unsafe

Templates

- Reuse source code
 - generic programming
 - use types as parameters in class or function definitions
- Template functions
 - Example: sort function
- Template classes
 - Example: containers such as stack, list, queue...
 - Stack operations are independent of the type of items in the stack
 - template member functions

Function Templates

- Perform similar operations on different types of data.
- Swap function for two int arguments:

```
void swap( int& x, int& y ) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

- What if we want to swap floats, strings, Currency, Person?

Example: swap function template

```
template < class T >
void swap( T& x, T& y ) {
    T temp = x;
    x = y;
    y = temp;
}
```

- The `template` keyword introduces the template
- The `class T` specifies a parameterized type name
 - `class` means any built-in type or user-defined type
- Inside the template, use `T` as a type name

Function Template Syntax

- Parameter types represent:
 - types of arguments to the function
 - return type of the function
 - declare variables within the function

Template Instantiation

- Generating a declaration from a template class/function and template arguments:
 - Types are substituted into template
 - New body of function or class definition is created
 - syntax errors, type checking
 - Specialization -- a version of a template for a particular argument(s)

Example: Using swap

```
int i = 3;   int j = 4;  
swap(i, j);  // use explicit int swap
```

```
float k = 4.5; float m = 3.7;  
swap(k, m);  // instantiate float swap  
std::string s("Hello");  
std::string t("World");  
swap(s, t);  // std::string swap
```

- A template function is an instantiation of a function template

Interactions

- Only *exact* match on types is used
- No conversion operations are applied
 - swap(int, int); // ok
 - swap(double, double); // ok
 - swap(int, double); // error!
- Even *implicit* conversions are ignored
- Template functions and regular functions coexist

Overloading rules

- Check first for unique function match
- Then check for unique function template match
- Then do overloading on functions

```
void f(float i, float k) {};
```

```
template <class T>
```

```
void f(T t, T u) {};
```

```
f(1.0, 2.0);
```

```
f(1, 2);
```

```
f(1, 2.0);
```

Function Instantiation

- The compiler deduces the template type from the actual arguments passed into the function.
- Can be explicit:
 - for example, if the parameter is not in the function signature (older compilers won't allow this...)

```
template < class T >
void foo( void ) { /* ... */ }
foo<int>();        // type T is int
foo<float>();      // type T is float
```

Class templates

- Classes parameterized by types
 - Abstract operations from the types being operated upon
 - Define potentially infinite set of classes
 - Another step towards reuse!
- Typical use: container classes
 - `stack <int>`
 - is a stack that is parameterized over `int`
 - `list <Person&>`
 - `queue <Job>`

Example: Vector

```
template <class T>
class Vector {
public:
    Vector(int);
    ~Vector();
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    T& operator[](int);
private:
    T* m_elements;
    int m_size;
}
```

Usage

```
Vector<int> v1(100);
```

```
Vector<Complex> v2(256);
```

```
v1[20] = 10;
```

```
v2[20] = v1[20]; // ok if int->Complex  
                // defined
```

Vector members

```
template <class T>
Vector<T>::Vector(int size) : m_size(size) {
    m_elements = new T[m_size];
}

template <class T>
T& Vector<T>::operator[](int indx) {
    if (indx < m_size && indx > 0) {
        return m_elements[indx];
    } else {
        ...
    }
}
```

A simple sort function

```
// bubble sort -- don't use it!
template < class T >
void sort( vector<T>& arr ) {
    const size_t last = arr.size()-1;
    for (int i = 0; i < last; i++) {
        for (int j = last; i < j; j--) {
            if (arr[j] < arr[j - 1]) {
                // which swap?
                swap(arr[j], arr[j - 1]);
            }
        }
    }
}
```


Sorting the vector

```
vector<int> vi(4);  
vi[0] = 4; vi[1] = 3; vi[2] = 7; vi[3] = 1;  
sort( vi );          // sort( vector<int>& )
```

```
vector<string> vs;  
vs.push_back("Fred");  
vs.push_back("Wilma");  
vs.push_back("Barney");  
vs.push_back("Dino");  
vs.push_back("Prince");  
sort( vs );          // sort( vector<string>& )  
//NOTE: sort uses operator< for comparison
```

Templates

- **Templates can use multiple types**

```
template< class Key, class Value>
class HashTable {
    const Value& lookup(const Key&) const;
    void install(const Key&, const Value&);
    ...
};
```

- **Templates nest — they're just new types!**

```
Vector< Vector< double *> > // note space > >
```

- **Type arguments can be complicated**

```
Vector< int (*) (Vector<double>&, int)>
```

Expression parameters

- Template arguments can be *constant* expressions
- Non-Type parameters
 - can have a default argument

```
template <class T, int bounds = 100>
class FixedVector {
public:
    FixedVector();
    // ...
    T& operator[] (int);
private:
    T elements[bounds]; // fixed size array!
};
```

Non-Type parameters

```
template <class T, int bounds>  
T& FixedVector<T,bounds>::operator[]( int i ) {  
    return elements[i]; // no error checking  
}
```

Usage: Non-type parameters

- Usage

- `FixedVector<int, 50> v1;`
- `FixedVector<int, 10*5> v2;`
- `FixedVector<int> v3; // uses default`

- Summary

- Embedding sizes not necessarily a good idea
- Can make code faster
- Makes use more complicated
 - size argument appears everywhere!
- Can lead to (even more) code bloat

Templates and inheritance

- **Templates can inherit from non-template classes**

```
template <class A>  
class Derived : public Base { ...
```

- **Templates can inherit from template classes**

```
template <class A>  
class Derived : public List<A> { ...
```

- **Non-template classes can inherit from templates**

```
class SupervisorGroup : public  
    List<Employee*> { ...
```

Notes

- friends
- static members
- In general put the definition and the declaration for the template in the header file
 - won't allocate storage for the class at that point
 - compiler/linker has mechanism for removing multiple definitions

Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune
- Review implementation
 - Which types should be parameterized?
- Convert non-parameterized version into template
- Test against established test cases

What is STL

- STL = Standard Template Library
- Part of the ISO Standard C++ Library
- Data Structures and algorithms for C++.

Why should I use STL?

- Reduce development time.
 - Data-structures already written and debugged.
- Code readability
 - Fit more meaningful stuff on one page.
- Robustness
 - STL data structures grow automatically.
- Portable code.
- Maintainable code
- Easy

C++ Standard Library

- Library includes:
 - A **Pair** class (pairs of anything, int/int, int/char, etc)
 - Containers
 - **Vector** (expandable array)
 - **Deque** (expandable array, expands at both ends)
 - **List** (double-linked)
 - **Sets and Maps**
 - Basic Algorithms (sort, search, etc)
- All identifiers in library are in **std** namespace
using namespace std;

The three parts of STL

- Containers
- Algorithms
- Iterators

The 'Top 3' data structures

- **map**
 - Any key type, any value type.
 - Sorted.
- **vector**
 - Like c array, but auto-extending.
- **list**
 - doubly-linked list

Example using the vector class

- Use “namespace std” so that you can refer to vectors in C++ library
- Just declare a vector of ints (no need to worry about size)
- Add elements
- Have a pre-defined iterator for vector class, can use it to print out the items in vector

```
#include <iostream>
```

```
using namespace std;
```

```
#include <vector>
```

```
int main( ) {
```

```
    vector<int> x;
```

```
    for (int a=0; a<1000; a++)
```

```
        x.push_back(a);
```

```
    vector<int>::iterator p;
```

```
    for (p=x.begin();
```

```
        p<x.end(); p++)
```

```
        cout << *p << " ";
```

```
    return 0;
```

```
}
```

Class Exercises

- The code for the vector example exists at `vector.cpp`. Modify this code so it puts 5000 items in the vector, and then prints out every fifth element
 - Element 0, element 5, element 10, etc.

Basic Vector Operations

- Constructors

```
vector<Elem> c;  
vector<Elem> c1(c2);
```

- Simple Methods

```
V.size( )           // num items  
V.empty( )         // empty?  
==, !=, <, >, <=, >=  
V.swap(v2) // swap
```

- Iterators

```
I.begin( )         // first position  
I.end( )           // last position
```

- Element access

```
V.at(index)  
V[index]  
V.front( ) // first item  
V.back( )  // last item
```

- Add/Remove/Find

```
V.push_back(e)  
V.pop_back( )  
v.insert(pos, e)  
V.erase(pos)  
V.clear( )  
V.find(first, last, item)
```


Class Exercises

- Take a look at the code in `vector2.cpp` .
Predict the output of this program.
- Run the program to check your output.

List Class

- Same basic concepts as vector
 - Constructors
 - Ability to compare lists (`==`, `!=`, `<`, `<=`, `>`, `>=`)
 - Ability to access front and back of list
 - `x.front()`, `x.back()`**
 - Ability to assign items to a list, remove items
 - `x.push_back(item)`, `x.push_front(item)`**
 - `x.pop_back()`, `x.pop_front()`**
 - `x.remove(item)`**

Sample List Application

- Declare a list of strings
- Add elements
 - Some to the back
 - Some to the front
- Iterate through the list
 - Note the termination condition for our iterator
p != s.end()
 - Cannot use **p < s.end()** as with vectors, as the list elements may not be stored in order

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

int main( ) {
    list<string> s;
    s.push_back("hello");
    s.push_back("world");
    s.push_front("tide");
    s.push_front("crimson");
    s.push_front("alabama");
    list<string>::iterator p;
    for (p=s.begin(); p!=s.end(); p++)
        cout << *p << " ";
    cout << endl;
}
```

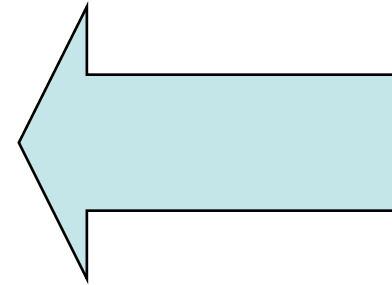
Maintaining an ordered list

- Declare a list
- Read in five strings, add them in order
- Print out the ordered list

```
#include <iostream>
using namespace std;
#include <list>
#include <string>
int main( ) {
    list<string> s; string t;
    list<string>::iterator p;
    for (int a=0; a<5; a++) {
        cout << "enter a string : ";
        cin >> t;
        p = s.begin();
        while (p != s.end() && *p < t) p++;
        s.insert(p, t);
    }
    for (p=s.begin(); p!=s.end(); p++)
        cout << *p << " ";
    cout << endl; }
```

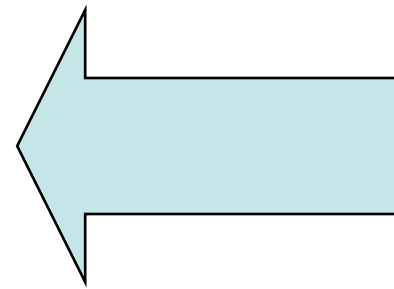
Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
    total += price[item];
```



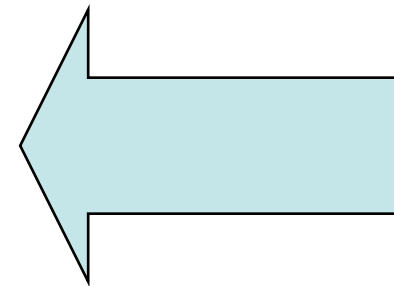
Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
    total += price[item];
```



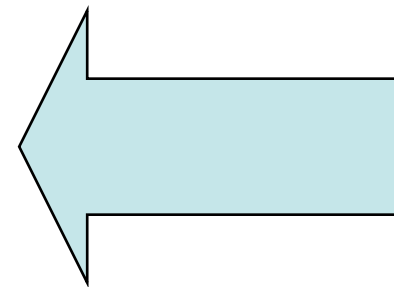
Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
    total += price[item];
```



Example Program

```
#include <map>
#include <string>
map<string,float> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;
string item;
double total=0;
while ( cin >> item )
    total += price[item];
```



Simple Example of Map

```
map<long,int> root;  
root[4] = 2;  
root[1000000] = 1000;  
long l;  
cin >> l;  
if (root.count(l)) cout<<root[l]  
else cout<<"Not perfect square";
```

Two ways to use Vector

- Preallocate

```
vector<int> v(100);  
v[80]=1; // okay  
v[200]=1; // bad
```

- Grow tail

```
vector<int> v2;  
int i;  
while (cin >> i)  
    v.push_back(i);
```

Example of List

```
list<int> L;  
for(int i=1; i<=5; ++i)  
    L.push_back(i);  
//delete second item.  
L.erase( ++L.begin() );  
copy( L.begin(). L.end(),  
    ostream_iterator<int>(cout, ","));  
// Prints: 1,2,3,5
```

Iterators

- Declaring
 `list<int>::iterator li;`
- Front of container
 `list<int> L;`
 `li = L.begin();`
- Past the end
 `li = L.end();`

Iterators

- Can increment

```
list<int>::iterator li;
```

```
list<int> L;
```

```
li=L.begin();
```

```
++li; // Second thing;
```

- Can be dereferenced

```
*li = 10;
```

Algorithms

- Take iterators as arguments

```
list<int> L;
```

```
vector<int> V;
```

```
// put list in vector
```

```
copy(    L.begin(),  
        L.end(),  
        V.begin() );
```

List Example Again

```
list<int> L;  
for(int i=1; i<=5; ++i)  
    L.push_back(i);  
//delete second item.  
L.erase( ++L.begin() );  
copy( L.begin(). L.end(),  
    ostream_iterator<int>(cout, ","));  
// Prints: 1,2,3,5
```

Typdefs

- Annoying to type long names
 - `map<Name, list<PhoneNum> > phonebook;`
 - `map<Name, list<PhoneNum> >::iterator finger;`
- Simplify with typedef
 - `typedef PB map<Name, list<PhoneNum> >;`
 - `PB phonebook;`
 - `PB::iterator finger;`
- Easy to change implementation.

Using your own classes in STL Containers

- Might need:
 - Assignment Operator, `operator=()`
 - Default Constructor
- For sorted types, like `map<>`
 - Need less-than operator: `operator<()`
 - Some types have this by default:
 - `int`, `char`, `string`
 - Some do not:
 - `char *`

Example of User-Defined Type

```
struct point  
{  
    float x;  
    float y;  
}
```

```
vector<point> points;  
point p; p.x=1; p.y=1;  
points.push_back(1);
```

Example of User-Defined Type

- Sorted container needs sort function.

```
struct full_name {  
    char * first;  
    char * last;  
    bool operator<(full_name & a)  
        {return strcmp(first, a.first) < 0;}  
}  
map<full_name,int> phonebook;
```

What do I need?

- g++ 2.96
 - Fine for all examples in this talk
 - 3.0.x is even better
 - using namespace std;
- Mostly works with MSVC++
 - So i am told.

Performance

- Personal experience 1:
 - STL implementation was 40% slower than hand-optimized version.
 - STL: used deque
 - Hand Coded: Used “circular buffer” array;
 - Spent several days debugging the hand-coded version.
 - In my case, not worth it.
 - Still have prototype: way to debug fast version.

Performance

- Personal experience 2
- Application with STL list ~5% slower than custom list.
- Custom list “intrusive”
 - struct foo {
 - int a;
 - foo * next;
 - };
- Can only put foo in one list at a time ☹️

Pitfalls

- Accessing an invalid vector<> element.

```
vector<int> v;
```

```
v[100]=1; // Whoops!
```

Solutions:

- use push_back()
- Preallocate with constructor.
- Reallocate with reserve()
- Check capacity()

Pitfalls

- Inadvertently inserting into map<>.

```
if (foo["bob"]==1)
```

```
//silently created entry "bob"
```

Use count() to check for a key without creating a new entry.

```
if ( foo.count("bob") )
```


Pitfalls

- Not using `empty()` on `list<>` .

- Slow

- `if (my_list.count() == 0) { ... }`

- Fast

- `if (my_list.empty()) {...}`

Pitfalls

- Using invalid iterator

```
list<int> L;
```

```
list<int>::iterator li;
```

```
li = L.begin();
```

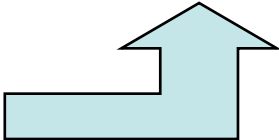
```
L.erase(li);
```

```
++li;           // WRONG
```

- Use return value of erase to advance

```
li = L.erase(li); // RIGHT
```

Common Compiler Errors

- `vector<vector<int>> vv;`
missing space 
lexer thinks it is a right-shift.
- any error message with `pair<...>`
`map<a,b>` implemented with `pair<a,b>`

STL versus Java Containers

STL

- Holds any type
- No virtual function calls
- Static type-checking

Java Containers

- Holds things derived from Object
- Virtual Function Call overhead
- No Static type-checking

Other data structures

- set, multiset, multimap
- queue, priority_queue
- stack , deque
- slist, bitset, valarray

Generic Programming Resources

- STL Reference Pages

www.sgi.com/tech/stl/

More Generic Programming

- GTL : Graph Template Library
- BGL : Boost Graph Library
- MTL : Matrix Template Library
- ITL : Iterative Template Library