Today's topic: Standard Template Library (STL)

- container classes
- iterators

Goal: introduction to concepts, enough background to use containers


Header Files

- As with all libraries, you need to include the header files that define classes and functions of the library
- STL headers
  - are part of the installed system, i.e. use `<...>`
  - do not end in `.h`
  - are part of the std namespace
- Example:
  ```cpp
  #include <string>
  using std::string;
  ```

I/O Classes

- The STL includes an implementation of the iostream library
- If you use stream I/O and you use STL classes, include the STL version of iostream
  ```cpp
  #include <list>
  using std::list;
  #include <iostream> // replaces iostream.h
  using std::cout;
  using std::endl;
  ```

Container Classes

- The STL has many of the same "containers" found in the Java API
  - One-dimensional lists (list, vector, etc)
  - Sets
  - Maps
- To use a class, you need to know which header file to include, e.g.
  ```cpp
  #include <list>
  using std::list;
  ```

Compile-Time Type Checking

- As we've seen before, C++ likes to do as much as possible at compile-time
- Elements of containers must all be of the same type
- The type is declared along with the name of the container instance, using template syntax
  ```cpp
  list<string> nameList;
  list<int> scores;
  ```

Polymorphic Containers

- If you want a container to hold more than one type, define a new base class
- Example: list of memory objects for performance simulator
  ```cpp
  class Memory {...};
  class RAM : Memory {...};
  class Disk : Memory {...};
  list<Memory> mem;
  ```
- Question: why a list of pointers to Memory objects instead of a list of Memory objects?
STL Containers Hold Copies of Objects

- Operations that insert items into STL containers make copies of the objects
  ```cpp
  list<string> words;
  string s = "hello"; // add s to end of list
  s += ", world";
  words.push_back(s) << endl;
  should print "hello"
  ```
- Operations that return items from a container also make copies

Holding References in Containers

- You can define a container to hold pointers to objects
  - efficiency
  - putting an object in more than one container
- But this can be a major source of errors
  ```cpp
  list<Gene*> chromosome;
  Gene* gp = new Gene(...);
  chromosome.push_back(gp);
  delete gp;
  ```
- The list thinks the Gene still exists, ...
  - Be extra careful when you use pointers

Pairs

- A very simple container is called a pair
  ```cpp
  #include <utility> // not req'd in Solaris
  using std::pair;
  using std::make_pair;
  ```
- One use is for the return type of a function that produces two output values, e.g. a Gene class method:
  ```cpp
  pair<int, int> location() {
    return pair<int, int>(_start, _end);
  }
  ```
- or use a built-in function named make_pair
  ```cpp
  return make_pair(_start, _end);
  ```

Pairs (cont’d)

- Pairs have no methods for getting or setting contents
- Just access the values directly, by name
  ```cpp
  for (i = 0; i < N; i++) {
    pair<int, int> loc = gene[i].location();
    cout << "Gene " << i << " at " << loc.first << "...
    << loc.second << endl;
  }
  ```

Lists

- Lists are ordered collections
- Properties:
  - fast insertion/deletion of elements
  - insertion/deletion equally efficient at any location
  - slow access to random elements
- List Constructors
  ```cpp
  #include <list>
  using std::list;
  ```
  ```cpp
  // create an initially empty list
  list<string> names;
  // create a list with N strings
  list<string> placeholders(N);
  // make a list with N empty strings
  list<string> blanks(N,"");
Nonmodifying List Operations

- To get the number of elements in a list:
  `names.size()`
- To see if a list is empty:
  `names.empty()`
- To compare two lists:
  `L1 == L2`  
  `L1 < L2`  
  `// etc`

Adding Elements to a List

- You can add elements to the front or back of a list:
  `names.push_front("Agatha Christie");`
  `names.push_back("Minette Walters");`
- You can insert elements into the middle if you have a "pointer" to a location in the list:
  `names.insert(p, "PD James");`
  Insert string before the item pointed to by p
- Special pointers called *iterators* are used to access list elements; iterators are described later in this lecture

Accessing List Elements

- The only methods that return a list element access the first or last element in the list:
  `string s = names.front();`
  `string t = names.back();`
- These methods do not check to see if the element exists; you might get an undefined value
- Check to see if the element exists before you use it:
  `if (!names.empty())`
  `cout << names.front() << endl;`

Removing Elements from a List

- The following operations remove elements, but do not return the removed item(s):
  `names.pop_front();`
  `names.pop_back();`
- To remove all strings that match a:
  `names.remove(s);`
- To remove all items:
  `names.clear();`

Special List Operations

- Sort a list:
  `names.sort();`
- Merge two sorted lists:
  `names.merge(newAuthors);`
- Reverse a list:
  `names.reverse();`

List Iterators

- To go along with the main list class is an auxiliary iterator class:
  `ListIterators`
- An iterator is an object that refers to an element inside a container:
  - Think of an iterator as a pointer
    - dereferenced with the * operator
    - incremented with ++
    - decremented with --
The list iterator class definition is imported into your program when you include the list header.

Declare an iterator by using the class name and the type iterator, e.g.,

```cpp
list<string>::iterator p;
```

defines p to be an object which points to elements of lists of strings.

Consider a for loop that iterates over elements of an array

```cpp
for (int i = 0; i < 5; i++)
    cout << a[i] << endl;
```

The array has 5 values, a[0] thru a[4]. The iterator i takes on the five values in 0..4, plus one final value of 5, which is one past the end of the array

```cpp
0 1 2 3 4 5
```

5 is a legal iterator value, as long as we don't use it in a dereference operation, i.e., we don't try to access a[5].

The same idea applies to list iterators.

We want the iterator to take on values that point to all elements in the list, plus one extra value that indicates the end of the list.

```
first iterator value
```

```
last iterator value ("null")
```

```
for (p = L.begin(); p != L.end(); p++)
```

```cpp
if (foo(*p))
    L.push_front(bar(*p)); // bad idea!
```

Don't change the contents of a container when you are traversing it with an iterator.

The next few slides will give brief overviews of other container classes in the STL.

Points to keep in mind:

- all containers have few basic operations in common, e.g., size() and empty()
- all have iterators, and methods that return iterator values, e.g., begin() and end()
Vectors

- A vector is a sequential list
  - fast access to random elements
  - expensive to insert/delete items in the middle

- Like the Java Vector class, C++ vectors will expand to handle the addition of new elements

Vector Constructors

```cpp
#include <vector>
using std::vector;
vector<int> a; // empty vector
vector<int> b(N); // N elements
vector<int> c(N,0); // N 0's
```

typedef pair<int, int> Point;
typedef vector<Point> Polygon;
Polygon hexagon(6);

Vector Operations

- Adding elements to a vector:
  ```cpp```
  a.push_back(n);
  ```cpp```
  n = a.back();
  ```cpp```
  a.pop_back();
- Changing the size of a vector (adds elements if necessary)
  ```cpp```
  a.resize(20);
- Accessing an element in the middle of a vector:
  ```cpp```
  n = a.at(i);
  n = a[i]; // note overloaded [] operator
- Note: at() does bounds checking, [] does not

Deques

- A deque (double-ended queue, pronounced "deck") is a vector that can grow at either end

```
#include <deque>
```

- Header:
  ```cpp```
  #include <deque>
  ```cpp```
- In addition to attributes of vector class, has
  ```cpp```
  push_front()
  ```cpp```
  pop_front()
  ```cpp```

Sets and Multisets

- Sets are collections of elements
  - items are not accessed by position
  - finding an item (e.g. "is X in the set?") and inserting an item are efficient, but not constant time

Sets and Multisets (cont’d)

- A multiset (aka "bag") can have duplicate items
  ```cpp```
  if (s.size() > n) {
    // this code should not be executed...
  }
  ```cpp```
  But if s is a multiset it would be possible to add a second copy of the string
Set and Multiset Declarations

- Sets and multisets are both defined in the `set` header file:
  ```
  #include <set>
  using std::set;
  using std::multiset;
  ```
- The base type of the set must be a type that supports comparisons:
  ```
  set<string> s;
  ```
  is OK because objects of the string class can be compared with `==` and `<`.
- You can override the default comparison functions, or define your own comparison class.

Set and Multiset Operations

- Add elements to a set or multiset using the `insert` method:
  ```
  s.insert("hello");
  ```
- Special operations for looking up items:
  ```
  int n = s.count("hello");
  ```
  ```
  set<string>::iterator p = s.find("hello");
  ```
  ```
  if (p == s.end()) {
    cout << "hello" is not in the set;
  }
  ```
- To traverse a set:
  ```
  for (p = s.begin(); p != s.end(); p++) {
    cout << *p << endl;
  }
  ```

Do Not Modify Elements in Sets

- The set and multiset classes will not be able to keep items sorted if you change them after they have been inserted.
- Not a problem if you allow the container to copy elements.
- Could be a problem if the container holds pointers to objects:
  ```
  set<Gene*> chromosome;
  Gene* gp = new Gene(...);
  chromosome.insert(gp);  // a bad idea...
  ```

Maps and Multimaps

- A map is set of associations between key/value pairs.
- The declaration defines the type used for the keys and the types of the values:
  ```
  #include <map>
  using std::map;
  ```
- Multimap containers allow multiple values for any key.

Map Elements are Pairs

- When inserting an item into a map, insert the key and value as a pair:
  ```
  multimap<const string,string> books;
  typedef pair<const string,string> item;
  books.insert(item("Christie","Murder on the Orient Express"));
  books.insert(item("Christie","Curtain"));
  books.insert(item("Christie","The Mysterious Affair at Styles"));
  ```

Map Elements (cont’d)

- Retrieve elements as pairs, also:
  ```
  multimap<const string,string>::iterator p;
  for (p = books.begin(); p != books.end(); p++) {
    cout << "key = " << p->first << " val = " << p->second << endl;
  }
  ```
Maps as Associative Arrays

- The map class overloads the [] operator
- Use key values to index into the map

```cpp
map<const string, string> addrBook;
addrBook["Zena"] = "Ariola, Zena x6-4448 305 Des";
addrBook["Al"] = "Malony, Allen x6-4407 307 Des";
addrBook["Steve"] = "Fickas, Steve x6-3963 313 Des";
```
- Look up an entry using the key as index:
  ```cpp
cout << addrBook["Steve"] << endl;
```

Associative Arrays (cont’d)

- Be careful: if the item does not exist, it will be created
- The statement
  ```cpp
cout << addrBook["John"] << endl;
```
  creates a new entry, initialized to "" (via a call to the default string constructor)

Maps and Solaris

- I was not able to compile the associative map example with Solaris CC
- It does compile with GNU C++ (g++)
- See stl.C for demos of lists, vectors, etc
- See amap.C for the associative map demo

Both files (and associated classes and the Makefile) are in stl.tar