COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Associative Containers

- Associate values with keys
  - eg. hashtable
- Provide efficient insertion/retrieval
- Maintain an internal ordering
  - May differ with library implementations
  - ie. you can't trust the internal order

Maps

- Maps are the most common associative container
- Correspond to hashtables/hashmaps
- Operator overloading makes them easy to use

Simple Map Example

```
#include <map>
#include <string>
using namespace std;
int main() {
        map<string, int> count;
        string s;
        while (cin >> s) {
                count[s] = count[s] + 1;
                // OR: count[s]++;
        }
}
```

Accessing map items

- map[key]
  - Each key has only one value
  - Returns reference to value for key
  - Can modify the value
  - Creates key and value if needed

Map Iterator Example

```
#include <map>
#include <string>
using namespace std;
int main() {
    map<string, int> count;
    // fill count as previously
    for (map<string,int>::const_iterator
      i=count.begin(); i != count.end(); ++i)
          cout << i->first << '\t'
                  << i->second << endl;
}
```

Pair

- Dereferencing a map<key, value> iterator gives a pair<key, value> object
- A pair is a simple data structure
- The first member gives the key
- The second member gives the value
- You can use pairs directly in your code
  - ie. first doesn't have to be a key, second doesn't have to be a value
  - You can use anywhere you need to store two things together

Manually inserting into a map

- Given
  map<string, int> m;
- then can insert a pair with
  pair<string, int> p("test", 42);
  m.insert(p);
- or more simply
  m.insert(pair<string, int>("test", 42));

Set

- Similar to a map
- Has only a key, with no value
- Useful when you only care about existence

set example

```
#include <set>
using namespace std;
int main() {
    set<int> nums;
    int val;
    while (cin >> val)
        nums.insert(val);
    for (set<int>::const_iterator i =
        nums.begin(); i != nums.end(); ++i)
            cout << *i << endl;
}
```

Testing Existance

- How do you check if a particular key exists?
- Iterating over the set/map too be slow
- Both set and map provide a find() method
- It returns an iterator
  - To the entry with that key, if found
  - end() if key not found

set find() example

```
set<string> s;
set<string>::const_iterator si = s.find("it");
if (si != s.end())
    cout << "Found it" << endl;
else
    cout << "Didn't find it" << endl;
```

Another set find()

```
int main() {
    set<int> nums;
    int val;
    while (cin >> val)
        nums.insert(val);

    if (nums.find(42) != nums.end())
        cout << "42 entered sometime";
    else
        cout << "42 not entered";
}
```

map find() example

```
map<string, double> m;
map<string, double>::const_iterator mi
                                = m.find("it");
if (mi != m.end())
    cout << "It is : " << mi->second
                                << endl;
else
    cout << "Didn't find it" << endl;
```

Another map find()

```
int main() {
    map<string, int> count;
    string s;
    while (cin >> s) count[s]++;
    if (count.find("hello") != count.end())
        cout << "You said hello";
    else
        cout << "You never said hello";
}
```

Removing from a map/set

- Use the erase() method
- Can take a key:
  m.erase(key);
- or iterator:
  m.erase(m.find(key));
- Some STL implementations have a buggy key version
  - If so, just use the iterator version

multimap / multiset

- There are also 'multi' versions of map and set
- They allow multiple identical keys
- multimaps are not used too often
  - map<key, vector<value> > used instead

multiset Example

```
#include <set>
int main() {
   multiset<int> nums;
   int val;
   while (cin >> val)
        nums.insert(val);
   for (multiset<int>::const_iterator i =
      nums.begin(); i != nums.end(); ++i)
        cout << *i << endl;
}
```

Exceptions

- Allows seperation of error handling
- Detect errors locally
- Handle errors where appropriate
- Has a performance impact with most compilers

C++ Exceptions

- Exceptions can be of any type
- Could throw a vector<string>, int, string, Person, or whatever
- Usually it is best to use specific exception classes

Throwing An Exception

```
struct Domain_Error {
      int number;
      Domain_Error(int d) : number(d) { }
};

double square_root(int n) {
      if (n < 0) {
            throw Domain_Error(n);
      }
      return sqrt(n);
}
```

Catching An Exception

```
int main() {
   int num;
   while (cin >> num) {
      try {
         cout << square_root(num) << '\n';
      } catch(Domain_Error e) {
         cerr << e.number
                  << " is not valid\n";
      }
   }
}
```

Uncaught exceptions

- If an exception is thrown in a function but not caught, it goes to the calling function
- This continues all the way to main()
- If it's still not caught by main(), the program exits
- Usually with the message Aborted (core dumped)
- Can then use gdb as normal

Multiple Catchers

```
try {
      // do something...
} catch (SomeType st) {
      // handle SomeType exception
} catch (SomeOtherType sot) {
      // handle SomeOtherType exception
} catch (...) {
      // handle anything else
}
```

Multiple Catchers II

- First catch type that matches is used
- So with inheritance
  - Derived classes must come first
  - Otherwise the base will match
- Inherited exceptions can be useful

```
class IOError { };
class NoFileError : public IOError { };
try { } catch (IOError e) { }
```

Rethrowing Exceptions

- A catch block can throw an exception
- All remaining catch blocks on the same level are ignored
- The original exception can be rethrown
  - throw;
  - Even in a catch(...) block

```
try {
    try {
        throw 10;
    } catch (int i) {
        cerr << "Caught : " << i << endl;
        throw 20.0;
    } catch (...) {
        cerr << "Won't reach" << endl;;
    }
} catch (int i) {
    cerr << "int : " << i << endl;
} catch (double d) {
    cerr << "double : " << d << endl;
    throw;
}
```

Exception Specifications

- Functions can declare what exceptions they throw
- void func() throw (e1, e2);
  - func can throw types e1 and e2
  - Or classes inheriting from them
- void func() throw ();
  - func can not throw any exceptions
- void func();
  - func can throw any exceptions

Function Pointers

- The exception specifications of a function pointer must match the function

```
void foo() throw (X);
void foo2();

void (*pf1)() throw (X,Y) = &foo; // OK
void (*pf2)() throw () = &foo;    // error
void (*pf3)() throw (X) = &foo2;  // error
```