COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

vector

- Simplest container, often most efficient
- Has a size and a capacity
- Insertion can cause reallocation
- Reallocation gets new storage memory
  - Eg. when larger capacity needed
- Reallocation invalidates iterators
- Random Access Container
- Back Insertion Sequence

Bad Vector Code

```
void duplicate(vector<int> &v) {
    vector<int>::const_iterator
                        b = v.begin(),
                        e = v.end();
    for(; b != e; ++b)
        v.push_back(*b);
}
```

- Not safe
- Reallocation may invalidate `b` and `e`
- Easily fixed in this case

Better Vector Code

- Use `reserve()` to allocate space

```
void duplicate(vector<int> &v) {
    v.reserve(v.size()*2);
    vector<int>::const_iterator
                        b = v.begin(),
                        e = v.end();
    for(; b != e; ++b)
        v.push_back(*b);
}
```

list

- A doubly linked list
- Reversible Container
- Front Insertion Sequence
- Back Insertion Sequence
- Insertion and splicing do not invalidate iterators
- Deletion only invalidates iterators at deleted element

Using List

- A list is preferred over a vector when
  - Insertion from front/middle common
  - Deletion at front/middle is common
  - Random access is not required

List Example

```
#include <iostream>
#include <string>
#include <list>
int main() {
    list<string> l;
    string s;
    while (cin >> s) l.push_front(s);
    l.sort();
    copy(l.begin(), l.end(),
      ostream_iterator<string>(cout, "\n"));
}
```

ostream_iterator

- Recall copy(Il begin, Il end, Ol out)
- ostream_iterator is an output iterator for ostream objects
- Constructor:

```
template <typename T>
ostream_iterator<T>(ostream &os,
                    const char* sep = "");
```

- Creates an ostream_iterator object for outputting objects of type T to the ostream os separated by sep

Container output iterators

- What about output iterators which store items in a container?

```
list<string> l;
vector<string> v;
copy(l.begin(), l.end(), back_inserter(v));
```

- back_inserter(c) returns an iterator which does c.push_back()
  - Requires Back Insertion Sequence
- Similarly front_inserter(c)
  - Requires Front Insertion Sequence

istream_iterator

- What about istream input iterators?
- istream_iterator is an input iterator for istream objects
- Constructor:

```
template <typename T>
istream_iterator<T>(istream &is);
```

- Creates an istream_iterator object for inputting objects of type T from the istream is

istream_iterator example

```
int main() {
    list<int> l;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(l));
    copy(l.begin(), l.end(),
        ostream_iterator<int>(cout, ", "));
    cout << endl;
}
```

- Input: 2 63 42 1 8
- Output: 2, 63, 42, 1, 8

front_inserter example

```
int main() {
    list<int> l;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         front_inserter(l));
    copy(l.begin(), l.end(),
        ostream_iterator<int>(cout, ", "));
    cout << endl;
}
```

- Input: 2 63 42 1 8
- Output: 8, 1, 42, 63, 2

Another example

```
int  main()  {
 copy(istream_iterator<string>(cin),
        istream_iterator<string>(),
        ostream_iterator<string>(cout,  "\n"));
}
```

- Outputs each word input on its own line

deque

- Double-ended queue
- Random Access Container
- Front Insertion Sequence
- Back Insertion Sequence
- Slower than vector
  - Due to front insertion

set

- Sorted Associative Container
- Simple Associative Container
- Unique Associative Container
- Inserting does not invalidate iterators
- Deleting only invalidates iterators at deleted element

multiset

- Sorted Associative Container
- Simple Associative Container
- Multiple Associative Container
- Inserting does not invalidate iterators
- Deleting only invalidates iterators as deleted element

map

- Sorted Associative Container
- Pair Associative Container
- Unique Associative Container
- Inserting does not invalidate iterators
- Deleting only invalidates iterators at deleted element

multimap

- Sorted Associative Container
- Pair Associative Container
- Multiple Associative Container
- Inserting does not invalidate iterators
- Deleting only invalidates iterators at deleted element

Adapters

- Adapters are wrappers around a container
- Work with any container which meets requirements
- Provide more specific interfaces
- Always use them if you want only that interface
  - Eg: allows them to be replaced with a faster implementation

stack

- Last In First Out (LIFO)
- Only insert, retrieve, delete top element
- Can use any back insertion sequence
- Defaults to using a deque
- top() returns top element
- push() insert element at top
- pop() removes top element

queue

- First In First Out (FIFO)
- Only add at back, retrieve from front
- Can use any front and back insertion sequence
- By default a deque is used
- front() returns front element
- push() adds element to back
- pop() removes front element

priority_queue

- Can only retrieve/delete top element
- Largest element always at top
- Use LessThan Comparable elements
- Use any Random Access Container
- Uses a vector by default
- top() returns top element
- push() inserts an element
- pop() removes top element

Overriding default containers

- Definition of queue class:

```
template <class T, class Seq = deque<T> >
class queue { ... };
```

- Thus these are equivalent:
  - queue<int> q;
  - queue<int, deque<int> > q;
- And overriding deque is easy:
  - queue<int, list<int> > q;
- Similarly for stack, priority_queue

Function Objects

- Often used to supply a function to an algorithm
- Makes the algorithm more generic
- Sorting is the obvious example
  - Specify the comparison operation
- The STL uses function objects a lot

What is a Function Object?

- Something which can be called
  - ie. if `foo(...)` is valid then `foo` is a function object
- A function pointer is the simplest example
- Objects of classes can also be used
- `operator()` needs to be overloaded
- Using an object is more flexible
  - It can maintain state
  - ie. remember things, etc

Types of Function Objects

- Generator
  - Called with no arguments
- Unary Function
  - Called with one argument
- Unary Predicate
  - A Unary function that returns a boolean

Types of Function Objects II

- Binary Function
  - Called with two arguments
- Binary Predicate
  - A Binary Function that returns a boolean

Strict Weak Ordering

- Binary Predicate
  - ie. compares 2 objects, returns T/F
- `f(x,x)` is false
- `f(x,y)` implies `!f(y,x)`
- `!f(x,y)` and `!f(y,x)` implies x and y are equivalent (`x == y`)
- x and y equivalent and y and z equivalent, implies x and z equivalent
  - `x == y` and `y == z` implies `x == z`

Strict Weak Ordering Ex

- Less-than binary predicate (`<`) defines a Strict Weak Ordering, since
- `x < x` is false
- `x < y` implies `!(y < x)`
- `!(x < y)` and `!(y < x)` implies `x == y`
- `x == y` and `y == z` implies `x == z`

Less-than implementation

```
template <typename T>
struct less {
   bool operator()(const T &a,
                   const T &b) {
         return (a < b);
   }
};
less<int> comp;
int a, b;
if (comp(a, b))
      cout << a << "<" << b << endl;
```

Random Number Generator

- Unary Function
- f(N) returns an integer in range [0,N)
- Every integer in [0,N) will appear an equal number of times
- The nrand() function is an example

STL Function Objects

- STL provides a lot of function objects
- #include <functional> to access them

- Binary functions: plus, minus, ...
- Binary predicates: logical_and, logical_or, less, greater, equal_to, ...
- Unary predicates: logical_not, ...
- Unary functions: negate, ...