COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

String streams

- Streams allow:
  - input via >>
  - output via <<
- Input/output can be to/from anything
  - Usually files/terminal
  - But can be other things
- String streams allow input/output with >> and << to/from strings

String stream versions

- "New" and "old" versions
- Old version (strstream)
  - Uses C-style strings of fixed length
  - Not part of ANSI C++ standard
  - In most C++ libraries
- New version (sstream)
  - Uses C++ style string objects
  - Part of the ANSI C++ standard
  - Not yet in all C++ libraries
    - eg: the undergrad machines

Old string streams: strstream

- #include <strstream>
- Classes: istrstream, ostrstream
- Specify string as char * to constructor
- ostrstream also requires maximum string length

istrstream example

```
#include <iostream>
#include <strstream>
#include <string>
int main() {
        string s = "here is a short string";
        istrstream is(s.c_str());
        string w;
        while (is >> w) {
                cout << w << endl;
        }
}
```

ostrstream example

```
#include <iostream>
#include <iomanip>
#include <strstream>
#include <string>
int main() {
        char a[14] = "";
        ostrstream os(a, 14);
        os << setw(12) << setfill('*')
                                << 42.12 << endl;
        string s(a);    cout << s;
}
```

New string streams: sstream

- #include <sstream>
- Classes: istringstream, ostringstream
- Specify string as string object to constructor
- Strings grow automatically when required
- Not present on undergrad systems, so: http://www.cs.usyd.edu.au/~kev/pp/sstream
  - Use `-I.` when compiling
  - `g++ -Wall -g -I. -o prog main.cc`

istringstream example

```
#include <iostream>
#include <sstream>

int main() {
    string s = "here is a short string";
    istringstream is(s);
    string w;
    while (is >> w) {
        cout << w << endl;
    }
}
```

ostringstream example

```
#include <iostream>
#include <iomanip>
#include <sstream>

int main() {
    ostringstream os;
    os << setw(12) << setfill('*')
                   << 42.12 << endl;
    string s = os.str();
    cout << s;
}
```

Iterators

- A convention
  - Not part of the C++ language
- Designed to act like restricted pointers
- Allow access to contents of container
  - Without revealing internal structure
  - Independant of specific container
- The interface between container and algorithms using it
- Allow idiomatic code

Terminology

- A requirement is some operation doing something
  - ++ to move a step is a requirement
- A concept is a set of requirements
  - An iterator is a concept
- A model is something that fulfils a concept
  - A pointer is a model of an iterator

Basic Concepts

- Assignable
  - Possible to copy values
  - Possible to assign new values
  - int is a model
  - const int is not
- Default Constructable
  - Can construct object with no args
  - T() and T t; valid constructs
  - int is a model
  - int& is not

Basic Concepts

- Equality Comparable
  - Can compare two values for equality
  - `x == y` and `x != y` must do so
- LessThan Comparable
  - Can test if an object is less than another
  - `x < y` and `x > y` must do so

Input Iterator

- Simplest iterator
- Requirements
  - Equality Comparable
  - Assignable
  - Can dereference for reading
  - Can increment
  - Dereference and increment must alternate

Input Iterator II

- Requirements are minimums
- A type which provides more is still a model
- Code requiring input iterators can only assume the minimum requirements are provided

Output Iterators

- The other simple iterators
- Requirements
  - Equality Comparable
  - Assignable
  - Can dereference for writing
  - Can increment
  - Dereference and increment must alternate

Iterator Example

```
template <typename II, typename OI>
OI copy(II begin, II end, OI out) {
    for (; begin != end; ++out, ++begin)
        *out = *begin;
    return out;
}
```

Forward Iterator

- Both an input and output iterator
- Dereference and increment don't have to alternate
  - Allows multipass algorithms
- `p = q; ++q; *p = x;` does what it should
  - `*q` is not modified

Bidirectional Iterator

- A Forward Iterator
- Also provides decrement

```
template <typename BI, typename OI>
OI reverse_copy(BI begin, BI end,
                                OI out) {
  while (begin != end)
    *out++ = *--end;
  return out;
}
```

Bidirectional Example

```
template <typename BI>
void reverse(BI begin, BI end) {
    while ( (begin != end) &&
            (begin != --end) ) {
        swap(*begin++, *end);
    }
}
```

Random Access Iterator

- A Bidirectional Iterator
- Supports random access
  - i + a and i - a moves a elements
  - i[a] is equivalent to *(i + a)
  - i1 - i2 gives distance between iterators
  - LessThan Comparable
- Random Access must be constant time

Random Access Example

```
template <typename RI>
void random_shuffle(RI begin, RI end) {
    if (begin == end) return;
    for (RI i = begin + 1; i != end; ++i) {
        RI other = begin +
                   nrand(i - begin + 1);
        swap(*i, *other);
    }
}
```

Ranges

- Iterators are usually used as ranges
- begin and end iterators
- The range is [begin,end)
  - begin is in the range
  - end is past the end of the range
  - ie. begin is included in the range, end is not

Containers

- Container
  - Provides Input Iterators
  - Only one iterator active at a time
  - begin() and end() return iterators
- Forward Container
  - A Container which also provides Forward Iterators
  - Multiple iterators can be active
  - No reordering between mutative operations

Containers

- Reversible Container
  - A Forward Container which also provides Bidirectional Iterators
  - rbegin() and rend() return reversed iterators
- Random Access Container
  - A Reversible Container which also provides Random Access Iterators

Container Typedefs

- value_type
  - type of element
- reference
  - reference to element
- const_reference
  - const reference to element
- pointer
  - pointer to element
- const_pointer
  - const pointer to element

Container Typedefs

- iterator
  - type of the iterator
- const_iterator
  - type of non-modifying iterator
- difference_type
  - represents distance between elements
- size_type
  - represents container size

Container Members

- a.size()
  - return number of elements
- a.max_size()
  - upper bound on size
- a.empty()
  - equivalent to a.size() == 0
- a.swap(b)
  - swaps container contents
- a.begin(), a.end()
  - return iterators

Container Abstractions

- Sequence
  - A Forward Container
  - Elements are not reordered
  - Add or delete elements at any point
    - insert() and erase() members
    - Use iterators to specify position
- Associative Containers
  - Elements are looked up via keys
  - Elements added/deleted via keys
    - find(), insert(), erase() members

Sequence Abstractions

- Front Insertion Sequence
  - push_front(), pop_front()
  - Insertion at front in constant time
  - Access first item in constant time
- Back Insertion Sequence
  - push_back(), pop_back()
  - Append to end in constant time
  - Access last item in constant time

Associative Abstractions

- Unique Associative Container
  - No two elements have same key
  - Conditional insertion
- Multiple Associative Container
  - Elements can have same key
- Simple Associative Container
  - Elements are their own keys

Associative Abstractions

- Pair Associative Containers
  - value_type is pair<key,value>
- Sorted Associative Container
  - Elements sorted by key
- Hashed Associative Container
  - Uses a hash table implementation