

I/O and STL Algorithms

See also: Chapter 14, Chapter 22

I/O and STL Algorithms

CS193D, 2/29/06

Raw Input and Output

```
ostream::put(char ch);  
ostream::write(const char* data, int dataSize);
```

```
char ch = 'a';
```

```
cout.put(ch);  
cout.write("test", 4);
```

```
int istream::get();  
istream::unget();  
istream::putback(char ch);  
int istream::peek();  
istream::getline(char* buffer, int size);
```

```
// reading without tokens  
while (inStream.get(next)) {  
    name += next;  
}
```

```
// looking ahead
int next = cin.peek();
if (isdigit(next)) {
    processNumber();
} else {
    processText();
}
```

```
// getting a line of text
char buffer[kBufferSize + 1];
cin.getline(buffer, kBufferSize);
```

```
// or...
```

```
string myLine;
std::getline(cin, myLine);
```

Flushing Output Streams

A *buffered* output stream's built-up contents are written when:

- An affiliated input stream is used.
- A sentinel (`endl`) is reached.
- The stream is destructed.
- The buffer is full.
- You call `flush`:

```
cout << "Notice the lack of an end of line." ;  
cout.flush() ;
```

Error Handling

Streams can be in a *good* state or a *bad* state. A not-good stream may have encountered a file error, the end of the file, bad input, etc.

```
good()    // is the stream usable?
bad()     // has a fatal error occurred?
fail()    // did the most recent operation fail?
clear()   // clear the error state (after you fixed it)
eof()     // has the end been reached? (input stream)
```

```
ifstream myFile("test.in");
if (!myFile.good()) {
    cerr << "Unable to open test.in!" << endl;
    exit(1);
}
```

I/O Manipulators

Manipulators are objects that can change I/O behavior mid-stream. For example, *endl* is really a manipulator.

```
#include <ios>
#include <iomanip>

bool b = true;
cout << noboolalpha << b << boolalpha << b << endl;
```

```
1true
```

Other Manipulators:

```
setprecision // parameterized manipulator for fractional #s
setw // parameterized for numerical field width
setfill // set the character to pad numbers (parameterized)
ws // skip current whitespace (input)
```

Non-Console Streams

Streams of strings:

```
ostream outStream;  
outStream << "test " << myInteger;  
cout << "stream is " << outStream.str() << endl;  
  
istringstream inStream("<foo><bar></bar></foo>");
```

File streams:

```
ofstream outFile("file.txt");  
outFile.seekp(4, ios_base::beg);  
  
ifstream inFile("file.txt");  
inFile.seekg(-2, ios_base::end);  
ios_base::pos_type curPos = inFile.tellg();
```

STL Algorithms

STL's library of algorithms is separate from its container classes. Since algorithms work on iterators, you can mix and match algorithms and containers.

Example: Determine if there's a 3 in myIntVector

```
#include <algorithm>
#include <numeric>

vector<int>::iterator location =
    find(myIntVector.begin(), myIntVector.end(), 3);

if (location != myIntVector.end()) {
    cout << "3 was found!" << endl;
} else {
    cout << "3 wasn't found." << endl;
}
```

find_if using Function Pointers

```
bool perfectScore(int num) { return num == 100; }

void anyPerfect(vector<int> scores) {
    vector<int>::iterator result =
        find_if(scores.begin(), scores.end(), perfectScore);

    if (result != scores.end()) {
        cout << "there was at least one perfect." << endl;
    }
}
```

find_if using Functors

```
class PerfectFinder {
    public:
        bool operator() (int num) { return num == 100; }
};

void anyPerfect(vector<int> scores) {
    PerfectFinder dontFunctorWithMyHeart;
    vector<int>::iterator result =
        find_if(scores.begin(), scores.end(),
                dontFunctorWithMyHeart);

    if (result != scores.end()) {
        cout << "there was at least one perfect." << endl;
    }
}
```

Why Use Functors?

Answer: They let you build little machines with state.

```
class PerfectFinder {
public:
    PerfectFinder() {
        mAllowExtraCredit = false;
    }

    bool operator()(int num) {
        if (mAllowExtraCredit && num >= 100) {
            return true;
        }
        return (num == 100);
    }

    bool mAllowExtraCredit;
}
```

accumulate

If you want to do something over a collection of elements, use *accumulate*.

```
// simply add all elements - no operation specified
double arithmeticMean(const vector<int>& nums) {
    double sum = accumulate(nums.begin(), nums.end(), 0);
    return (sum / nums.size());
}

// geometric mean
int product(int num1, int num2) { return (num1 * num2); }

double geometricMean(const vector<int>& v) {
    double mult = accumulate(v.begin(), v.end(), 1, prod);
    return (pow(mult, 1.0 / nums.size()));
}
```

Built-in Functors

```
#include <function>

// includes plus, minus, multiples, divides, etc.

int main() {
    plus<int> myPlus;
    cout << myPlus(4, 5) << endl;
}

// rewrite geometric mean using built-in functor
double geometricMean(const vector<int>& v) {
    double mult = accumulate(v.begin(), v.end(), 1,
                             multiplies<int>());
    return (pow(mult, 1.0 / v.size()));
}
```

Note: Objects in the container must support the arithmetic operator being used!

Comparison Functors

The STL also provides `equal_to`, `not_equal_to`, `less`, `greater`, `less_equal`, and `greater_equal`.

By default, `priority_queue` uses `less`, but it can be changed:

```
priority_queue<int, vector<int>, greater<int> > myReverseQ;

myReverseQ.push(3);
myReverseQ.push(4);
myReverseQ.push(2);
myReverseQ.push(1);

while (!myReverseQ.empty()) {
    cout << myReverseQ.top() << endl;
    myReverseQ.pop();
}
```

Logical Functors

The problem: `find()` and `find_if()` both take single-argument callbacks, so you can't use comparison functors.

Solution: Binders

```
void anyPerfect(vector<int> scores) {
    vector<int>::iterator result =
        find_if(scores.begin(), scores.end(),
                bind2nd(greater_equal<int>(), 100));

    if (result != scores.end()) {
        cout << "there was at least one perfect." << endl;
    }
}
```

Negators

Negators simply negate the result of a predicate. Not to be confused with The New Gators, a garage band from Florida.

```
// find the first score < 100
vector<int>::iterator result = find_if(v.begin(), v.end(),
    not1(bind2nd(greater_equal<int>(), 100)));
```

Use `not1` for unary functions, and `not2` for binary functions.

Making STL-aware Functors

You can make your custom functors compatible with the STL func by subclassing `unary_function` or `binary_function`.

```
class myIsDigit : public unary_function<char, bool>
{
    public:
        bool operator()(char c) const {return ::isdigit(c);}
};
```

```
bool isNumber(const string& s) {
    vector<int>::iterator it = find_if(s.begin(), s.end(),
        not1(myIsDigit()));
    return (it == s.end());
}
```

Utility Algorithms

```
int x = 4;  
int y = 5;
```

```
cout << max(x, y);  
cout << min(x, y);
```

```
swap(x, y);
```

```
cout << max(x, y);  
cout << min(x, y);
```

Note: These are just utilities and don't work on sequences, so they don't take an iterator.

Other Non-Modifying Algorithms

```
find()
find_if()
adjacent_find() // find two consecutive equal elements
find_first_of() // search for several values simultaneously
search()        // find a particular sequence
search_n()      // find first seq of n consec. matching els
min_element()   // finds the minimum element
max_element()   // finds the maximum element

accumulate()

count()         // count number of elements with given val
count_if()      // count() with a predicate

equal()         // is the entire range the same?
mismatch()      // obtain iter to first unequal point
lexicographical_compare() // all in one less than the other
```

```

int main(int argc, char** argv) {
    // The list of elements to be searched
    int elems[] = {5, 6, 9, 8, 8, 3};

    // Construct a vector from the list, exploiting the
    // fact that pointers are iterators too.
    vector<int> myVector(elems, elems + 6);
    vector<int>::const_iterator it, it2;

    // Find the min and max elements in the vector.
    it = min_element(myVector.begin(), myVector.end());
    it2 = max_element(myVector.begin(), myVector.end());
    cout << "The min is " << *it << " and the max is " <<
        *it2 << endl;

    // Find the first pair of matching consec elements.
    it = adjacent_find(myVector.begin(), myVector.end());
    if (it != myVector.end()) {
        cout << "Found two consec equal elements of value "
            << *it << endl;
    }
}

```

```

// Find the first of two values.
int targets[] = {8, 9};
it = find_first_of(myVector.begin(), myVector.end(),
                  targets, targets + 2);

if (it != myVector.end()) {
    cout << "Found one of 8 or 9: " << *it << endl;
}

// Find the first subsequence.
int sub[] = {8, 3};
it = search(myVector.begin(), myVector.end(), sub,
            sub + 2);
if (it != myVector.end()) {
    cout << "Found subsequence 8, 3 at position " <<
        it - myVector.begin()
        << endl;
}

// Find the last subsequence (which should be the same

```

```

// as the first).
it2 = find_end(myVector.begin(), myVector.end(), sub,
              sub + 2);
if (it != it2) {
    cout << "Err: search & find_end found diff subseqs"
         << " even though there is only one match.\n";
}

// Find the first subsequence of two consecutive 8s.
it = search_n(myVector.begin(), myVector.end(), 2, 8);
if (it != myVector.end()) {
    cout << "Found two consecutive 8s starting at pos "
         << it - myVector.begin() << endl;
}
}

```

```

The min is 3 and the max is 9
Found two consecutive equal elements of value 8
Found one of 8 or 9: 9
Found subsequence 8, 3 at position 4
Found two consecutive 8s starting at position 3

```

```

// Numerical Processing Examples
template<typename Container>
void populateContainer(Container& cont)
{
    int num;

    while (true) {
        cout << "Enter a number (0 to quit): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        cont.push_back(num);
    }
}

int main(int argc, char** argv)
{
    vector<int> myVector;
    list<int> myList;
}

```

```

cout << "Populate the vector:\n";
populateContainer(myVector);
cout << "Populate the list:\n";
populateContainer(myList);
if (myList.size() < myVector.size()) {
    cout << "Sorry, the list is not long enough.\n";
    return (0);
}

// Compare the two containers.
if (equal(myVector.begin(), myVector.end(),
          myList.begin())) {
    cout << "The two containers have equal elements\n";
} else {
    // If the containers were not equal, find out why
    pair<vector<int>::iterator, list<int>::iterator>
    miss =
        mismatch(myVector.begin(), myVector.end(),
                 myList.begin());

    cout << "The first mismatch is at position "

```

```

        << miss.first - myVector.begin() <<
            ". The vector has value "
        << *(miss.first) << " and the list has value "
            << *(miss.second)
        << endl;
    }

    // Now order them.
    if (lexicographical_compare(myVector.begin(),
                               myVector.end(), myList.begin(),
                               myList.end())) {
        cout << "The vector is lexicographically first.\n";
    } else {
        cout << "The list is lexicographically first.\n";
    }
}

```

Populate the vector:

Enter a number (0 to quit): 5

Enter a number (0 to quit): 6

Enter a number (0 to quit): 7

Enter a number (0 to quit): 8

Enter a number (0 to quit): 0

Populate the list:

Enter a number (0 to quit): 5

Enter a number (0 to quit): 6

Enter a number (0 to quit): 7

Enter a number (0 to quit): 9

Enter a number (0 to quit): 0

The first mismatch is at position 3. The vector has value 8
and the list has value 9

The vector is lexicographically first.

The Operational Algorithm (for_each)

```
void printPair(const pair<int, int>& elem) {
    cout << elem.first << "->" << elem.second << endl;
}

int main()
{
    map<int, int> map;
    map.insert(make_pair(4, 40));
    map.insert(make_pair(5, 50));
    map.insert(make_pair(6, 60));
    map.insert(make_pair(7, 70));
    map.insert(make_pair(8, 80));

    for_each(map.begin(), map.end(), printPair);
}
```

You can do some neat single-pass processing using a functor (e.g. MinAndMax)

Modifying Algorithms

```
transform()      // Like for_each() but you can modify els

copy()          // Copy elements from one range to another

replace()       // Replace elements in a range
replace_if()    // Replace based on a predicate
replace_copy()  // Put results in a destination range
replace_copy_if() // Ditto

remove()        // Remove elements from the range
remove_if()     // Remove with a predicate

unique()        // Removes duplicate contiguous elements

reverse()       // Changes the order

// and there's more!
```

Sorting Algorithms

In general, the most important ones are:

```
sort()           // Uses something close to quicksort
```

```
merge()         // Merge two sorted ranges together
```

```
sort(v.begin(), v.end());  
sort(v2.begin(), v2.end());
```

```
vector<int> vMerged;
```

```
merge(v.begin(), v.end(), v2.begin(), v2.end(),  
      vMerged.begin());
```

And there's a lot of other stuff, too

The STL algorithms also include randomly shuffling elements, set operations (union, difference, etc.), different types of sorts, and other potentially useful operations.

If a container has its own version of an algorithm (e.g. `map.find()`), use the container's version instead. It's probably faster.