Generic programs

Why bother?

C++ Standard Library

- Provides generic
  - containers
  - iterators
  - algorithms
- Algorithms operate on (most) any container
- Iterators provide the glue between algorithms and containers

Using the library

- Obviously, avoids (re)writing the same algorithms over and over
- Less obviously, lets us write surprisingly succinct programs to solve common programming problems

Review

- 5 kinds of iterator categories
- Iterators provide data structure independence
- Algorithms use iterators to manipulate the contents of unknown kinds of containers
- We can write N algorithms for use with M container types in O(M+N) effort rather than O(N*M)

A simple example

- Copy a `vector` into a built-in array:

```
vector<string> v(100);
// fill up the vector

string array[100];
copy(v.begin(), v.end(), array);
```

Algorithms & containers

- Algorithms operate on elements not on containers.
- A common mistake:

```
list<string> l;
// fill up the list

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

- This code fails because we never allocated any space for `v`.

Alternatives

```
// Alternative 1
vector<string> v(l.size());
copy(l.begin(), l.end(), v.begin());

// Alternative 2
vector<string> v;
v.resize(l.size());
copy(l.begin(), l.end(), v.begin());

// Alternative 3
vector<string> v;
copy(l.begin(), l.end(), back_inserter(v));
```

Containers with unknown size

- The library supplies iterators that read from and write to streams.
- We can use these as we would any other iterator:

```
copy(istream_iterator<string>(cin),
     istream_iterator<string>(),
     back_inserter(v));
```

Solving the homework

- Use an iterator to find the sum of floating point numbers read in from the standard input:

```
cout <<
    accumulate(istream<double>(cin),
               istream<double>(),
               0.0);
```

- Include `<numeric>` to get `accumulate`

The hard part

```
template<class T> class Istream_iterator {
  istream* str;
  T value;
  bool end_marker;
  friend bool operator!=
       (const Istream_iterator<T>&,
        const Istream_iterator<T>&);
  void read() {
     end_marker = (*str) ? true : false;
     if (end_marker) *str >> value;
     end_marker = (*str) ? true : false;
  }
```

More of the hard stuff

```
public:
 Istream_iterator():
         str(&cin), end_marker(false) { }
 Istream_iterator(istream& s):
         str(&s), end_marker(false) { read(); }
 const T& operator*() const { return value; }
 Istream_iterator<T> operator++(int)
     { Istream_iterator ret = *this;
       read();
       return ret;
     }
};
```

The rest of the hard stuff

```
template <class T>
bool operator!=
       (const Istream_iterator<T>& lhs,
        const Istream_iterator<T>& rhs)
{
    return !(  lhs.str == rhs.str &&
               lhs.end_marker == rhs.end_marker
            || l.end_marker == false &&
               rhs.end_marker == false  );
}
```

The easy part

```
template <class It, class T>
accum(It b, It e, T sum) {
  while (b != e)
    sum += *b++;
  return sum;
}
int main() {
  cout << accum(Istream_iterator<double>(cin),
                Istream_iterator<double>(),
                0.0);
  return 0;
}
```

A word-processing example

- Assume we are writing a WYSIWYG editor.
- We want to allow the user to change the paragraph style, switching from block indented paragraphs to space indented paragraphs.

Block indented

This is a block-indented paragraph. Note that there is no indentation on the first line of the paragraph.

Each paragraph is separated from the next by a blank line.

Space indented

   This is not a block-indented paragraph. Note that the first line of each paragraph begins with spaces.
   Paragraphs are not separated from each other by blank lines.

Strategy

- Assume the document to reformat is stored in a `vector`.
- Write a function that will:
  - find consecutive empty lines
  - delete the empty lines
  - insert indentation in the next line, checking first that the next line is not itself empty.

The code

```
void indent(vector<string>& doc) {
  int i = 0;
  while (i < doc.size()) {
    // find empty lines
    while (i < doc.size() && doc[i].empty()) {
      // erase empty lines
      doc.erase(doc.begin() + i);
      // insert indentation, if appropriate
      if (i < doc.size() && !doc[i].empty())
        doc[i].insert(0,"   ");
    }
    ++i;
  }
}
```

Destructive operations

- `erase` *removes* the indicated element
  - there are fewer elements in `doc` after the `erase` which explains all those tests on `doc.size()`
  - all the elements after the one `erased` must be moved
- Our program works but performance degrades with large inputs
- Why?

Another approach

- Apparently, we need a data structure from which we can *efficiently* remove, for example, `list`
- But, first, we need to eliminate the dependence on indices
  - indices *are* the problem
  - `list` does not support index operations

Use iterators instead

```
void indent(vector<string>& doc) {
 vector<string>::iterator iter = doc.begin();
 while (iter != doc.end()) {
   // find empty lines
   while (iter != doc.end() && iter->empty()) {
      // delete the empty line
      iter = doc.erase(iter);
      // insert indentation, if appropriate
      if (iter != doc.end() && !iter-> empty())
        iter->insert(0, "    ");
   }
   if (iter != doc.end()) ++iter;
 }
}
```

One subtlety

- Note that we check before incrementing `iter`. Why?
  - Incrementing past the `end()` value is *undefined* and the call to `erase` might have advanced `iter` to the `end()`.
  - The `while` loop tests
    `iter != doc.end()`
    which is more general: Most iterators only provide (in)equality.

Using List Instead

```
void indent(list<string>& doc) {
 list<string>::iterator iter = doc.begin();
 while (iter != doc.end()) {
   // find empty lines
   while (iter != doc.end() && iter->empty()) {
      // delete the empty line
      iter = doc.erase(iter);
      // insert indentation, if appropriate
      if (iter != doc.end() && !iter-> empty())
        iter->insert(0, "    ");
   }
   if (iter != doc.end ) ++iter;
 }
}
```

Why bother?

File Size	`list`	`vector`
938	0.0	0.0
1870	0.1	0.2
10120	0.7	4.4
20240	1.5	22.6

Another example

- Produce a cross-reference
  - for each word in the input
  - list the lines on which the word occurred
- We'll need to store the words and an associated container that will hold the line numbers

The `map` class

- Associative arrays are containers that behave like arrays but their indices can be any well-ordered type
- AWK, Perl and some other languages have associative arrays built-in
- In C++, they are part of the library

First, a simpler problem

- We'll start by just counting the number of times each word occurs in the input

```
map<string> m;
string s;
while (cin >> s)
    m[s]++;
```

Printing the contents

- Dereferencing a `map` yields a `pair`
- `pair` is a simple library class that contains two values, called `first` and `second`.
- These data members are `public`.

Printing the `map`

```
map<string,int>::const_iterator
               iter = m.begin();
while (iter != m.end()) {
       cout << iter->first
            << " "
            << iter->second
            << endl;
       ++iter;
}
```

Strategy for X-ref

- Read a line of input, remembering the current line number;
- Break the line into words;
- Strip punctuation;
- Store the word in a `map`;
- Update the value indexed by the word to indicate that it occurred on the current line number.

Variables

```
// map from words to line numbers
map<string,vector<int> > m;

// temporary to hold words as we read them
string s;

// line counter
int line_cnt = 0;
```

Read the input

```
while (getline(cin, s)) {
  line_cnt++;
  string::iterator b, e = s.begin();
  while((b = find_if(e, s.end(),
        not1(ptr_fun(isspace)))) != s.end()) {
     e = find_if(b, s.end(), isspace);
     string w(b, e);
     w.erase(remove_if(w.begin(), w.end(),
                  ispunct), w.end());
     vector<int>& v = m[w];
     if (v.empty() || line_cnt != v.back())
        v.push_back(line_cnt);
    }
}
```

Library functions

- `find_if` is like `find` but it tests a predicate rather than looking for a specific value

```
find_if(e, s.end(), not1(ptr_fun(isspace)))
```

is equivalent to

```
bool notspace(char c) {
   return !isspace(c);
}

// ...
find_if(e, s.end(), notspace);
```

Print the `vector`

```
map<string, vector<int> >::const_iterator
                        map_it=m.begin();
while (map_it != m.end()) {
  cout << map_it->first << ": ";
  const vector<int>& v = map_it->second;
  vector<int>::const_iterator
              vec_it = v.begin();
  while (vec_it != v.end()) {
    cout << *vec_it;
    if (++vec_it != v.end())
      cout << ",";
    else
      cout << endl;
  } ++map_it;
}
```

Homework

- Reimplement the cross-reference program without using the standard library algorithms or iterators.