Library Algorithms

Rationale for Library algorithms

- many container operations apply to more than one type of container (e.g. `insert`, `erase`. etc)
- Every container has *iterators*
- STL exploits these common interfaces to provide collection of standard algorithms
- like containers, algorithms use a consistent interface
- most algorithms defined in `<algorithm>` header

string box concatenation revisited

We said that for

```
for (vector<string>::const_iterator it = bottom.begin();
     it != bottom.end(); ++it)
    ret.push_back(*it);
```

vector provided a direct operation:

```
ret.insert(ret.end(), bottom.begin(), bottom.end());
```

But there is a *general* solution:

```
copy(bottom.begin(), bottom.end(), back_inserter(ret));
```

- copy is a *generic algorithm*
- back_inserter is an *iterator generator*

Generic algorithm: `copy`

- Not part of any kind of container
- STL generic algorithms usually take iterators as arguments
  - access to elements through `*`, `++`, etc operations
- `copy(begin, end, out)` copies elements
  in `[beging, end)` to sequence starting at `out`

Iterator adaptors

- Functions that yield iterators
- Defined in `<iterator>`
- `bact_inserter` takes container as argument and gives iterator, when used as destination, that appends values to container.

Note the *WRONG* calls to copy:

```
// won't compile
copy(bottom.begin(), bottom.end(), ret);

// compiles but undefined behaviour
copy(bottom.begin(), bottom.end(), ret.end());
```

String splitting revisited

```
// 'true' if the argument is whitespace, 'false' otherwise
bool space(char c)
{
        return isspace(c);
}

// 'false' if the argument is whitespace, 'true' otherwise
bool not_space(char c)
{
        return !isspace(c);
}

vector<string> split(const string& str)
{
        typedef string::const_iterator iter;
        vector<string> ret;

        iter i = str.begin();
        while (i != str.end()) {

                // ignore leading blanks
                i = find_if(i, str.end(), not_space);

                // find end of next word
                iter j = find_if(i, str.end(), space);

                // copy the characters in '[i,' 'j)'
                if (i != str.end())
                        ret.push_back(string(i, j));
                i = j;
        }
        return ret;
}
```

String splitting revisited (2)

- `find_if` first two arguments are iterators that delimit sequence `[begin, end)`, third argument is predicate
  - calls predicate on each elements in the sequence, stopping as soon as predicate is `true`
  - returns corresponding iterator, or second argument if no element found that match
- note that `isspace` is *overloaded* in STL
  - never easy to pass overloaded function directly as argument as compiler has no idea which one to use
  - write a wrapper that does an explicit call to overloaded function
- note that STL algorithms are writen to handle empty ranges *gracefully*
  - returns the `end` iterator if the range is empty

Palindromes

```
bool is_palindrome(const string& s)
{
        return equal(s.begin(), s.end(), s.rbegin());
}
```

- `rbegin()` returns iterator that start at last element of container, and marches backward
- `equal` compares two sequences for equality
  - first two arguments are iterators that delimit first sequence `[begin, end)`
  - third argument is iterator indicating starting point of second sequence; assumes enough elements in this sequence

Finding URLs

Simplified solution: looking for sequences of characters of the form *protocol-name*`://`*resource-name*

protocol-name contains only letters, *resource-name* may consist of letters, digits and permitted punctuation.

Valid URL: at least one valid character before and after the `://` delimiter.

Finding URLs (2): find_urls

```
vector<string> find_urls(const string& s)
{
        vector<string> ret;
        typedef string::const_iterator iter;
        iter b = s.begin(), e = s.end();

        // look through the entire input
        while (b != e) {

                // look for one or more letters followed by '://'
                b = url_beg(b, e);

                // if we found it
                if (b != e) {
                        // get the rest of the URL
                        iter after = url_end(b, e);

                        // remember the URL
                        ret.push_back(string(b, after));

                        // advance 'b' and check for more URLs on this line
                        b = after;
                }
        }
        return ret;
}
```

Finding URLs (3): `url_end`

```cpp
string::const_iterator
url_end(string::const_iterator b, string::const_iterator e)
{
        return find_if(b, e, not_url_char);
}

bool not_url_char(char c)
{
        // characters, in addition to alphanumerics, that can appear in a URL
        static const string url_ch = "~;/?:@=&$-_.+!*'(),";

        // see whether 'c' can appear in a URL and return the negative
        return !(isalnum(c) ||
                 find(url_ch.begin(), url_ch.end(), c) != url_ch.end());
}
```

- `static` local variables are created on first call and preserved across calls
- `find` works like `find_if` but uses a specific value instead of a predicate

Finding URLs (3): `url_beg`

```cpp
string::const_iterator
url_beg(string::const_iterator b, string::const_iterator e)
{
        static const string sep = "://";

        typedef string::const_iterator iter;

        // 'i' marks where the separator was found
        iter i = b;

        while ((i = search(i, e, sep.begin(), sep.end())) != e) {

                // make sure the separator isn't at the beginning or end of the line
                if (i != b && i + sep.size() != e) {

                        // 'beg' marks the beginning of the protocol-name
                        iter beg = i;
                        while (beg != b && isalpha(beg[-1]))
                                --beg;

                        // is there at least one appropriate character
                        //before and after the separator?
                        if (beg != i && !not_url_char(i[sep.size()]))
                                return beg;
                }

                // the separator we found wasn't part of a URL;
                // advance 'i' past this separator
                i += sep.size();
        }
        return e;
}
```

Finding URLs (4): `url_beg` con't

- `search` takes two pairs of iterators
  - first pair denotes a sequence we are looking into
  - second pair denotes sequence we are looking for
  - returns iterator to start of search sequence in searched sequence
  - returns second argument on failure
- if container supports indexing, so do its iterators
  - `beg[i]` is `*(beg + i)`
  - `beg[-1]` is `*(beg - 1)`
- decrement operation on iterator

Comparing grading schemes

Remember the student grading using medians...
Students could exploit this scheme to only do half of their homework without impact on their final mark!
Question: do students who do all the homework have better marks than those who don't?
What if

- we use average instead of median, giving 0 to homework not done
- we use median of homework actually done

We need program that

- reads student records and separates students into those who did all the homework from those who didn't
- apply each of the 3 grading schemes (median, average, median of work done), and report median grade of each group

Comparing grading schemes (2): classifying students

```
bool did_all_hw(const Student_info& s)
{
    return ((find(s.homework.begin(), s.homework.end(), 0))
            == s.homework.end());
}


// students who did and didn't do all their homework
vector<Student_info> did, didnt;

// read the student records and partition them
Student_info student;
while (read(cin, student)) {
  if (did_all_hw(student))
    did.push_back(student);
  else
    didnt.push_back(student);
 }

// verify that the analyses will show us something
if (did.empty()) {
  cout << "No student did all the homework!" << endl;
  return 1;
 }
if (didnt.empty()) {
  cout << "Every student did all the homework!" << endl;
  return 1;
 }
```

Comparing grading schemes (3): comparing student groups

```
void write_analysis(ostream& out, const string& name,
                    double analysis(const vector<Student_info>&),
                    const vector<Student_info>& did,
                    const vector<Student_info>& didnt)
{
        out << name << ": median(did) = " << analysis(did) <<
                       ", median(didnt) = " << analysis(didnt) << endl;
}
```

Third parameter represents a function

Comparing grading schemes (4): analysis function – median

```
// this version does not work
double median_analysis(const vector<Student_info>& students)
{
        vector<double> grades;

        transform(students.begin(), students.end(),
                  back_inserter(grades), grade);
        return median(grades);
}
```

`transform` takes 3 iterators and a function

- first 2 operators delimit a range
- third operator is destination where to put elements after applying the function to them
- it is programmer's responsibility to ensure destination has enough capacity

Comparing grading schemes (5): analysis function – median issues

- Major issue with previous version of `median_analysis` is that `grade` is overloaded
  - So compiler does not know which version we mean
- Second issue, the `grade` function we want can `throw` an exception if a student did no homework. So better handle this exception to stop it spreading and killing the program.

Write auxiliary function that solves both issues

Comparing grading schemes (6): analysis function – median fixed

```
double grade_aux(const Student_info& s)
{
        try {
                return grade(s);
        } catch (domain_error) {
                return grade(s.midterm, s.final, 0);
        }
}

// this version works fine
double median_analysis(const vector<Student_info>& students)
{
        vector<double> grades;

        transform(students.begin(), students.end(),
                  back_inserter(grades), grade_aux);
        return median(grades);
}
```

Comparing grading schemes (7): analysis function – average

Computing averages:

```
double average(const vector<double>& v)
{
    if (v.size() == 0) return 0.0;
    return accumulate(v.begin(), v.end(), 0.0) / v.size();
}
```

- `accumulate` defined in `<numeric>`
  - first two parameters define a range
  - adds all values in the range to the third parameter
  - type of the sum is the type of the third argument ⇒ *must* use 0.0

Comparing grading schemes (8): analysis function – average

```
double average_grade(const Student_info& s)
{
    return grade(s.midterm, s.final, average(s.homework));
}

double average_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(),
              back_inserter(grades), average_grade);
    return median(grades);
}
```

Comparing grading schemes (9): optimistic median

```
// median of the nonzero elements of 's.homework'
// '0' if no such elements exist
double optimistic_median(const Student_info& s)
{
        vector<double> nonzero;
        remove_copy(s.homework.begin(), s.homework.end(),
                    back_inserter(nonzero), 0);

        if (nonzero.empty())
                return grade(s.midterm, s.final, 0);
        else
                return grade(s.midterm, s.final, median(nonzero));
}

double optimistic_median_analysis(const vector<Student_info>& students)
{
        vector<double> grades;

        transform(students.begin(), students.end(),
                  back_inserter(grades), optimistic_median);
        return median(grades);
}
```

- there are "copy" versions of many algorithms
- `remove_copy` takes range, destination and value: destination gets copies of all elements in the range that differ from the value.

Comparing grading schemes (10): putting it all together

```
int main()
{
        // students who did and didn't do all their homework
        vector<Student_info> did, didnt;

        // read the student records and partition them
        Student_info student;
        while (read(cin, student)) {
                if (did_all_hw(student))
                        did.push_back(student);
                else
                        didnt.push_back(student);
        }

        // verify that the analyses will show us something
        if (did.empty()) {
                cout << "No student did all the homework!" << endl;
                return 1;
        }
        if (didnt.empty()) {
                cout << "Every student did all the homework!" << endl;
                return 1;
        }


        // do the analyses
        write_analysis(cout, "median", median_analysis, did, didnt);
        write_analysis(cout, "average", average_analysis, did, didnt);
        write_analysis(cout, "median of homework turned in",
                       optimistic_median_analysis, did, didnt);

        return 0;
}
```

Classifying students, revisited

There are efficient algorithmic solutions to the classification problem:

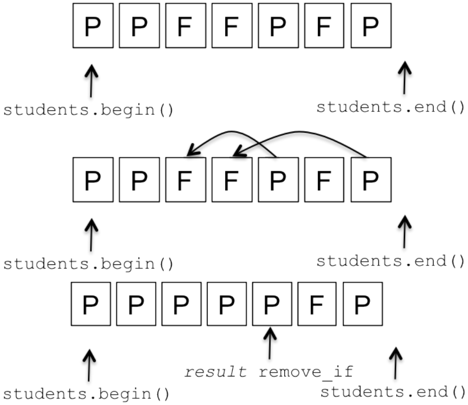
```
bool pgrade(const Student_info& s)
{
        return !fgrade(s);
}

vector<Student_info>
extract_fails(vector<Student_info>& students) {
        vector<Student_info> fail;
        remove_copy_if(students.begin(), students.end(),
                       back_inserter(fail), pgrade);
        students.erase(remove_if(students.begin(), students.end(),
                                 fgrade), students.end());
        return fail;
}
```

remove

`remove` and its associated functions (e.g. `remove_if`) does *not remove anything*.

Instead, it moves elements to be kept towards the beginning of the container, overwriting those that should be removed. The result of the function is an iterator to one past the last kept element.

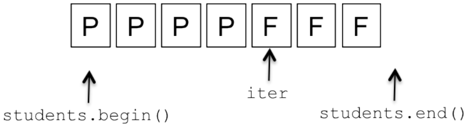


Classifying students: one pass solution

```
vector<Student_info>
extract_fails(vector<Student_info>& students)
{
        vector<Student_info >::iterator iter =
                stable_partition(students.begin(), students.end(), pgrade);
        vector<Student_info> fail(iter, students.end());
        students.erase(iter, students.end());

        return fail;
}
```

`stable_partition` (and `partition`): elements that satisfy the predicate are places before those that don't



Associative containers

`map`

`map` provides an associative array and stores *key-value* pairs.

Each `map` element is a `pair` (`first` and `second` data members).

For `map`, the keys are always `const`

Counting words

```
int main()
{
        string s;
        map<string, int> counters; // store each word and an associated counter

        // read the input, keeping track of each word and how often we see it
        while (cin >> s)
                ++counters[s];

        // write the words and associated counts
        for (map<string, int>::const_iterator it = counters.begin();
             it != counters.end(); ++it) {
                cout << it->first << "\t" << it->second << endl;
        }
        return 0;
}
```

- `counters[s]` is the integer associated with the `string` `s`
- when indexing a `map` with a new key, the `map` automatically creates a new element with that key, and the value is *value-initialized* (for `int` initialised to 0).

Cross-referencing table

```
// find all the lines that refer to each word in the input
map<string, vector<int> >
        xref(istream& in,
             vector<string> find_words(const string&) = split)
{
        string line;
        int line_number = 0;
        map<string, vector<int> > ret;

        // read the next line
        while (getline(in, line)) {
                ++line_number;

                // break the input line into words
                vector<string> words = find_words(line);

                // remember that each word occurs on the current line
                for (vector<string>::const_iterator it = words.begin();
                     it != words.end(); ++it)
                        ret[*it].push_back(line_number);
        }
        return ret;
}
```

Cross-referencing table (2)

- `map<string, vector<int> >`: note the > > as the compiler would get confused with >> which it would interpret as an input operator
- `find_words` defines a function operator with a *default* value
  ```
  xref(cin); // split to find words
  xref(cin, find_urls); // find_urls to find words
  ```

Print the cross-reference table

```cpp
int main()
{
        // call 'xref' using 'split' by default
        map<string, vector<int> > ret = xref(cin);

        // write the results
        for (map<string, vector<int> >::const_iterator it = ret.begin();
             it != ret.end(); ++it) {
                // write the word
                cout << it->first << " occurs on line(s): ";

                // followed by one or more line numbers
                vector<int>::const_iterator line_it = it->second.begin();
                cout << *line_it;       // write the first line number

                ++line_it;
                // write the rest of the line numbers, if any
                while (line_it != it->second.end()) {
                        cout << ", " << *line_it;
                        ++line_it;
                }
                // write a new line to separate each word from the next
                cout << endl;
        }

        return 0;
}
```