Library Algorithms and Associative Containers

Laurent Mathy

Object-Oriented Programming Projects

February 25, 2019

Outline

1. Library Algorithms
2. Associative Containers

Rationale for Library algorithms

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

string box concatenation revisited

We said that for:

```
1 for (const auto& x : v2)
2     v1.push_back(x);
```

vector provided a direct method:

```
1 v1.insert(v1.end(), v2.begin(), v2.end());
```

But there is an even more *generic* solution:

```
1 copy(v2.begin(), v2.end(), back_inserter(v1));
```

- 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, and access elements through `*`, `++`, *etc.*
- `copy(begin, end, out)` copies elements in `[begin, end)` to sequence starting at `out`.

Iterator adaptors

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

Note the **wrong** calls to `copy`:

```
1 // Won't compile, v1 is not an iterator
2 copy(v2.begin(), v2.end(), v1);
```

```
1 // Compiles, but undefined behaviour
2 copy(v2.begin(), v2.end(), v1.end());
```

Remember that any operation that modifies the container **invalidates** its iterators, hence the need for iterator adaptors.

String splitting revisited

```
 9  static bool is_space(char c) { return isspace(c); }
10
11  vector<string> split(const string& s) {
12      typedef string::const_iterator iter;
13      vector<string> ret;
14
15      iter i = s.begin();
16      while (i != s.end()) {
17          // Find start of next word
18          i = find_if_not(i, s.end(), is_space);
19          // Find end of next word
20          iter j = find_if(i, s.end(), is_space);
21          // Copy the characters in [i, j)
22          if (i != s.end())
23              ret.push_back(string(i, j));
24          i = j;
25      }
26      return ret;
27  }
```

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 matching element is found.
  - `find_if_not` returns as soon as predicate is `false` instead.
- 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

```
8  bool is_palindrome(const string& s) {
9      return equal(s.begin(), s.end(), s.rbegin());
10 }
```

- `rbegin()` returns an 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`

```
42  vector<string> find_urls(const string& s) {
43      vector<string> ret;
44      typedef string::const_iterator iter;
45      // Look through the entire input
46      iter b = s.begin(), e = s.end();
47      while (b != e) {
48          // Look for one or more letters followed by `://`
49          b = url_begin(b, e);
50          // If we found it
51          if (b != e) {
52              // Get the rest of the URL
53              iter after = url_end(b, e);
54              // Remember the URL
55              ret.push_back(string(b, after));
56              // Advance `b` and check for more URLs
57              b = after;
58          }
59      }
60      return ret;
61  }
```

Finding URLs (3): `url_end`

```
8   static bool not_url_char(char c) {
9       // Special characters that can appear in URLs
10      static const string url_ch = "~;/?:@=&$-_.+!*'(),";
11      // Return false if `c` can appear in URLs
12      return !(isalnum(c)
13          || find(url_ch.begin(), url_ch.end(), c) != url_ch.end());
14  }
15
16  static string::const_iterator
17  url_end(string::const_iterator b, string::const_iterator e) {
18      return find_if(b, e, not_url_char);
19  }
```

- `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_begin

```
21  static string::const_iterator
22  url_begin(string::const_iterator b, string::const_iterator e) {
23      static const string sep = "://";
24      typedef string::const_iterator iter;
25      iter i = b; // `i` marks where separator was found
26      while ((i = search(i, e, sep.begin(), sep.end())) != e) {
27          // Make sure the separator isn't at the end of string
28          if (i + sep.size() != e) {
29              iter beg = i; // `beg` marks start of protocol-name
30              while (beg != b && isalpha(beg[-1]))
31                  --beg;
32              // At least one good char before and after ://?
33              if (beg != i && !not_url_char(i[sep.size()]))
34                  return beg;
35          }
36          // Found separator wasn't part of a URL, move past it
37          i += sep.size();
38      }
39      return e;
40  }
```

Finding URLs (4): `url_begin` 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 homeworks 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 a program that:

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

Comparing grading schemes (2): classifying students

```
9   static bool did_all_hws(const Student_info& s) {
10      return find(s.homeworks.begin(), s.homeworks.end(), 0)
11          == s.homeworks.end();
12  }
```

```
71  // Read the student records and partition them
72  vector<Student_info> did, didnt;
73  Student_info student;
74  while (read(cin, student)) {
75      if (did_all_hws(student)) did.push_back(student);
76      else didnt.push_back(student);
77  }
78  // Verify that the analyses will show us something
79  if (did.empty()) {
80      cerr << "No student did all the homeworks!" << endl;
81      return 1;
82  }
83  if (didnt.empty()) {
84      cerr << "Every student did all the homeworks!" << endl;
85      return 1;
86  }
```

Comparing grading schemes (3): comparing student groups

```
14  static void write_analysis(
15          ostream& out, const string& name,
16          double analysis(const vector<Student_info>&),
17          const vector<Student_info>& did,
18          const vector<Student_info>& didnt)
19  {
20      out << name << ": median(did) = " << analysis(did)
21          << ", median(didnt) = " << analysis(didnt) << endl;
22  }
```

Third parameter represents a function.

Comparing grading schemes (4): analysis function – median

```
1  // This version does not work
2  double median_analysis(const vector<Student_info>& students)
3  {
4    vector<double> grades;
5    transform(students.begin(), students.end(),
6              back_inserter(grades), grade);
7    return median(grades);
8  }
```

`transform` takes 3 iterators and a function.

- First 2 iterators delimit a range.
- Third iterator 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 from spreading and killing the program.

Write auxiliary function that solves both issues.

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

```
24  static double grade_aux(const Student_info& s) {
25      try { return grade(s); }
26      catch (domain_error) {
27          return grade(s.midterm, s.final, 0);
28      }
29  }
30
31  static double median_analysis(
32          const vector<Student_info>& students)
33  {
34      vector<double> grades;
35      transform(students.begin(), students.end(),
36                back_inserter(grades), grade_aux);
37      return median(grades);
38  }
```

Comparing grading schemes (7): analysis function – average

```
7  double average(const vector<double>& xs) {
8      if (xs.empty())
9          return 0.0;
10     double sum = accumulate(xs.begin(), xs.end(), 0.0);
11     return sum / xs.size();
12 }
```

- `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
    $\Longrightarrow$ **must** use 0.0.

Comparing grading schemes (8): analysis function – average

```
40  static double average_grade(const Student_info& s) {
41      return grade(s.midterm, s.final, average(s.homeworks));
42  }
43
44  static double average_analysis(
45          const vector<Student_info>& students)
46  {
47      vector<double> grades;
48      transform(students.begin(), students.end(),
49                back_inserter(grades), average_grade);
50      return median(grades);
51  }
```

Comparing grading schemes (9): analysis function – optimistic median

```
53  // Median of the nonzero elements of `s.homeworks`, or 0 if none
54  static double optimistic_median(const Student_info& s) {
55      vector<double> nonzero;
56      remove_copy(s.homeworks.begin(), s.homeworks.end(),
57                  back_inserter(nonzero), 0);
58      double homework_grade = nonzero.empty() ? 0 : median(nonzero);
59      return grade(s.midterm, s.final, homework_grade);
60  }
61
62  static double optimistic_median_analysis(
63          const vector<Student_info>& students) {
64      vector<double> grades;
65      transform(students.begin(), students.end(),
66                back_inserter(grades), optimistic_median);
67      return median(grades);
68  }
```

- 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 value

Comparing grading schemes (10): putting it all together

```
70  int main() {
71      // Read the student records and partition them
72      vector<Student_info> did, didnt;
73      Student_info student;
74      while (read(cin, student)) {
75          if (did_all_hws(student)) did.push_back(student);
76          else didnt.push_back(student);
77      }
78      // Verify that the analyses will show us something
79      if (did.empty()) {
80          cerr << "No student did all the homeworks!" << endl;
81          return 1;
82      }
83      if (didnt.empty()) {
84          cerr << "Every student did all the homeworks!" << endl;
85          return 1;
86      }
87      // Do the analyses
88      write_analysis(cout, "median", median_analysis, did, didnt);
89      write_analysis(cout, "average", average_analysis, did, didnt);
90      write_analysis(cout, "median of homework turned in",
91                     optimistic_median_analysis, did, didnt);
92
93      return 0;
94  }
```

Don't Repeat Yourself (DRY)

`median_analysis()`, `average_analysis()` and `optimistic_median_analysis()` are **awfully similar**, differing only in the grading function passed to `transform()`.

Factorize their functionality into a single `analysis()` function, that takes the grading function as a parameter, and modify `write_analysis()` so that it calls `analysis()` directly.

Don't Repeat Yourself (DRY): refactored version

```
14  static double analysis(const vector<Student_info>& students,
15                         double grade(const Student_info&)) {
16      vector<double> grades;
17      transform(students.begin(), students.end(),
18                back_inserter(grades), grade);
19      return median(grades);
20  }
21
22  static void write_analysis(ostream& out, const string& name,
23                             double grade(const Student_info&),
24                             const vector<Student_info>& did,
25                             const vector<Student_info>& didnt) {
26      out << name << ": median(did) = " << analysis(did, grade)
27          << ", median(didnt) = " << analysis(didnt, grade)
28          << endl;
29  }
```

```
68  // Do the analyses
69  write_analysis(cout, "median", grade_aux, did, didnt);
70  write_analysis(cout, "average", average_grade, did, didnt);
71  write_analysis(cout, "median of homework turned in",
72                 optimistic_median, did, didnt);
```

Classifying students, revisited

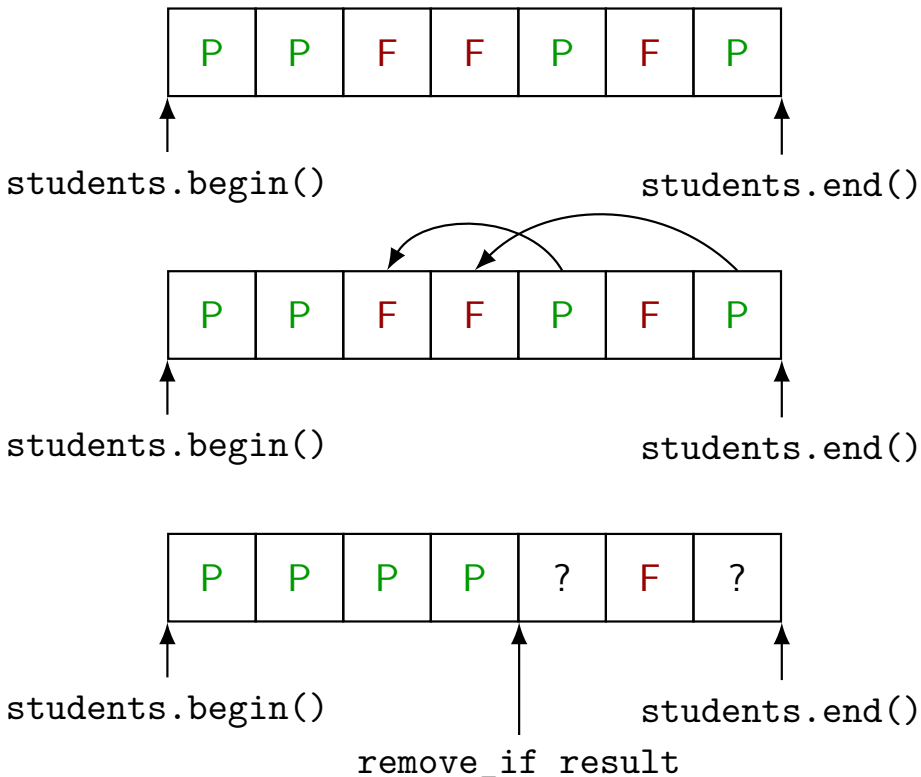
There are efficient algorithmic solutions to the classification problem:

```
19  vector<Student_info> extract_fails_1(vector<Student_info>& students)
20  {
21      vector<Student_info> fails;
22      copy_if(students.begin(), students.end(),
23              back_inserter(fails), fgrade);
24      students.erase(remove_if(students.begin(), students.end(),
25                               fgrade),
26                     students.end());
27      return fails;
28  }
```

`remove`

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

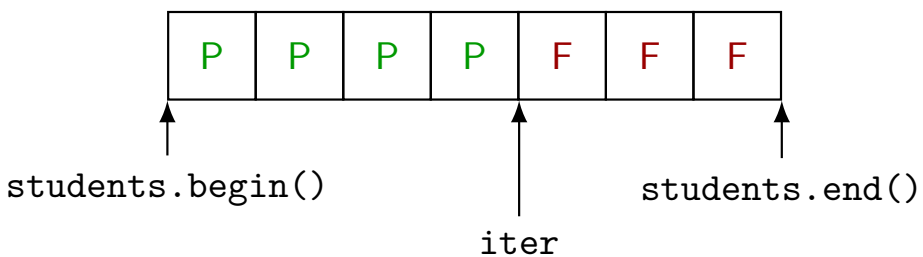
Instead, it moves elements to be kept towards the beginning of the container, overwriting those that should be removed. The result of the function is an iterator to one past the last kept element.



Classifying students: one pass solution

```
30  vector<Student_info> extract_fails_2(
31          vector<Student_info>& students)
32  {
33      vector<Student_info>::iterator iter =
34          stable_partition(students.begin(), students.end(), pgrade);
35      vector<Student_info> fails(iter, students.end());
36      students.erase(iter, students.end());
37      return fails;
38  }
```

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



Order is not preserved!

Outline

1. Library Algorithms
2. 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

```
7   string s;
8   map<string, int> counters; // (word, counter) pairs
9
10  // Read the input, keeping track of word counts
11  while (cin >> s)
12      ++counters[s];
13
14  // Write the words and associated counts
15  for (const auto& c : counters)
16      cout << c.first << "\t" << c.second << endl;
```

- `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

```
10  static map<string, vector<int> >
11  xref(istream& in, vector<string> find_words(const string&) = split)
12  {
13      string line;
14      int line_number = 0;
15      map<string, vector<int> > ret;
16
17      // Read the next line
18      while (getline(in, line)) {
19          ++line_number;
20          // Break the input line into words
21          vector<string> words = find_words(line);
22          // Remember that each word occurs on the current line
23          for (const auto& w : words)
24              ret[w].push_back(line_number);
25      }
26
27      return ret;
28  }
```

Cross-referencing table (2)

- `map<string, vector<int> >`: note the space in `> >`. A C++98 compiler would get confused with `>>`, which it would interpret as an input operator. No space needed since C++11.
- `find_words` defines a function parameter with a **default value**:

```
1 xref(cin);                  // split to find words
2 xref(cin, find_urls); // find_urls to find words
```

The default value must be visible by the caller
$\Longrightarrow$ should go in the header file for public functions.

Print the cross-reference table

```
31  // Call `xref` using `split` (default)
32  map<string, vector<int>> ret = xref(cin);
33
34  // Write the results
35  for (const auto& p : ret) {
36      // Write the word
37      cout << p.first << " occurs on line(s): ";
38      // Followed by one or more line numbers
39      auto line_it = p.second.cbegin();
40      cout << *line_it; // Write the first line number
41      // Write the rest of the line numbers, if any
42      ++line_it;
43      while (line_it != p.second.end()) {
44          cout << ", " << *line_it;
45          ++line_it;
46      }
47      // Write a new line to separate each word from the next
48      cout << endl;
49  }
```