Using Library Algorithms

Lectures 15 and 16

Hartmut Kaiser
hkaiser@cct.lsu.edu
http://www.cct.lsu.edu/~hkaiser/fall_2012/csc1254.html
Programming Principle of the Day

• Avoid Creating a YAGNI (You aren’t going to need it) - You should try not to add functionality until you need it.

http://en.wikipedia.org/wiki/YAGNI
Abstract

• Many containers share similar operations, like insert() or erase(). Those have the same interface for all of them (even for strings).
• All containers expose a companion iterator type allowing to navigate through the elements stored in the container. Again, all of them expose a similar interface
• We will see how the library exploits these similarities by exposing generic algorithms: by exposing uniform interfaces independent of the container they are applied to.
Analyzing Strings

• Looking back to picture concatenation:

    // copy entire bottom picture, use iterators
    for (vector<string>::const_iterator it = bottom.begin();
        it != bottom.end(); ++it)
    {
        ret.push_back(*it);
    }

    // copy entire bottom picture, use vector facilities
    ret.insert(ret.end(), bottom.begin(), bottom.end());

• There is an even more generic solution:

    // copy entire bottom picture, use standard algorithm
    std::copy(bottom.begin(), bottom.end(), back_inserter(ret));
Generic Algorithms

- `std::copy` is a *generic* algorithm
  - Not part of any container
  - Its operation is determined by its arguments
    - Most of the time the standard algorithms expect iterators
- `std::copy` takes 3 iterators (begin, end, out) and copies the range `[begin, end)` to a sequence starting at out (extending as necessary)
Standard Algorithm: copy

• Writing

    copy(begin, end, out);

• Is equivalent to (except for iterators not being copied):

    while (begin != end)
        *out++ = *begin++;

• What does ‘*out++ = *begin++;’ mean?

    {  *out = *begin; ++out; ++begin;  }
Iterator Adaptor

- `std::back_inserter()` is an *iterator adaptor*
  - Function returning an iterator created based on its arguments
  - Here, it takes a container and returns an iterator, which when used as a destination, appends elements to that container
- This will append all of `bottom` to `ret`:

  ```c++
  copy(bottom.begin(), bottom.end(), back_inserter(ret));
  ```
Caveats: copy

• This will not work (why?):
  
  ```cpp
  copy(bottom.begin(), bottom.end(), ret);
  // ret is not an iterator, but a container
  ```

• This will compile, but not work (why?):
  
  ```cpp
  copy(bottom.begin(), bottom.end(), ret.end());
  // while ret.end() is an iterator, it does not refer to
  // any element (remember ‘points’ past last element)
  ```

• Many problems, why designed that way?
  
  ▫ Separation of copying and appending (expanding a container) allows for more flexibility
  ▫ `back_inserter` useful in other contexts as well
Splitting Strings: Take 1

```cpp
vector<string> split(const string& s) {
    vector<string> words;
typedef string::size_type string_size;
    string_size i = 0;

    // invariant: we have processed characters [original value of i, i)
    while (i != s.size()) {
        // ignore leading blanks, find begin of word
        while (i != s.size() && isspace(s[i])) // short-circuiting
            ++i;

        // find end of next word
        string_size j = i;
        while (j != s.size() && !isspace(s[j])) // short-circuiting
            ++j;

        // if we found some non-whitespace characters, store the word
        if (i != j) {
            // copy from s starting at i and taking j - i chars
            words.push_back(s.substr(i, j - i));
            i = j;
        }
    }
    return ret;
}
```
Splitting Strings: Take 2

```cpp
vector<string> split(string const& str) {
    typedef string::const_iterator iterator_type;
    vector<string> ret;
    iterator_type i = str.begin();
    while (i != str.end()) {
        // ignore leading blanks
        i = find_if(i, str.end(), not_space);

        // find end of next word
        iterator_type 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;
}
```
Here are the predicates:

```cpp
// 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);
}
```
Standard Algorithm: find_if

- Find an entry in a sequence

  std::find_if(begin, end, pred);
  ▫ Goes over the sequence [begin, end) and calls the predicate ‘pred’ for each element
  ▫ Returns current position (iterator) as soon as the predicate returns true for the first time
  ▫ Essentially this finds the first element in the sequence matching the predicate
Palindromes

- Palindromes are words that are spelled the same way front to back as back to front: civic, eye, level, madam, etc.
- Simplest solution using a library algorithm:
  
  ```cpp
  bool is_palindrome(string const& s) {
    return equal(s.begin(), s.end(), s.rbegin());
  }
  ```
  
- New constructs: equal(), rbegin()
Reverse Iterators

• Like begin(), rbegin() returns an iterator
  ▫ It is an iterator that starts with the last element in the container
  ▫ When incremented, it marches backward through the container
  ▫ The iterator returned is called reverse iterator

• Correspondingly, like end(), rend() returns an iterator that marks the element before the first one
Standard Algorithm: `equal`

- The standard algorithm `equal()` compares two sequences
  - Returns whether these sequences hold the same elements
    - `std::equal(begin1, end1, begin2)`
  - Compares `[begin1, end1)` with elements in sequence starting at `begin2`
  - Assumes second sequence is long enough
Palindromes, Take Two

• Solution using other library algorithms:
  ▫ Find the iterator pointing to the middle element and use that as the end of the first sequence:

```cpp
bool is_palindrome(string const& s)
{
    string::const_iterator it = s.begin();
    advance(it, s.size() % 2 ? s.size()/2 + 1 : s.size()/2);
    return equal(s.begin(), it, s.rbegin());
}
```
Standard Algorithm: advance

- Advance a given iterator N times:
  
  ```
  void std::advance(it, n);
  ```

- Uses most efficient implementation depending on iterator type
  - Random access containers: uses operator+=()
  - Sequential containers: uses operator++() - N times
Finding URLs

- **Goal:** find all URLs embedded in a text document
  - **URL:** sequence of characters like `protocol-name://resource-name` (http://google.com/)
    - Protocol name contains letters only
    - Resource name contains letters, digits, punctuation characters
  - Look for `://` and then for protocol name preceding it and resource name that follow it
Finding URLs

- **Find all URLs embedded in a text document**

```cpp
vector<string> find_urls(string const& s) {
    vector<string> ret;
    typedef string::const_iterator iterator_type;
    iterator_type 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) {
            iterator_type after = url_end(b, e); // get the rest of the URL
            ret.push_back(string(b, after)); // remember the URL
            b = after; // advance b and check for more URLs on this line
        }
    }
    return ret;
}
```
Finding URLs

• This looks like:

```
...text http :// www.acceleratedcpp.com more text...
```

• The functions url_begin() and url_end() locate the url inside the overall string (document)
Finding URLs: url_end

- Look for one or more letters allowed in an url:

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

- Where not_url_char is our predicate:

```cpp
bool not_url_char(char c)
{
    // characters, in addition to alpha-numerics, that can
    // appear in a URL
    static string const 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());
}
```
Standard Algorithm: find

- Find an entry in a sequence
  \[
  \text{std::find(begin, end, value)};
  \]
  - Goes over the sequence \([\text{begin}, \text{end})\) and compares each element with ‘value’
  - Returns current position (iterator) as soon as the comparison evaluates to true for the first time
  - Essentially this finds the first element in the sequence matching the value
Finding URLs: `url_begin`

- We place several iterators inside our string:

```plaintext
...text http :// www.acceleratedcpp.com more text...
```

b <- beg
i

```
```

e
Finding URLs: url_begin

- Look for one or more letters followed by://

```cpp
string::const_iterator
url_beg(string::const_iterator b, string::const_iterator e)
{
    static string const sep = "://";
    typedef string::const_iterator iterator_type;
    iterator_type i = b; // i marks where the separator was found
    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
            iterator_type 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;
}
```
Standard Algorithm: search

- Find a sequence inside another sequence:
  - The search algorithm takes to sequences [begin1, end1) and [begin2, end2)
  - It tries to find [begin2, end2) inside [begin1, end1)
  - Returns iterator pointing to first element inside [begin1, end1) if found, and returns ‘end1’ otherwise
Comparing Grading Schemes

- Devious students could exploit our grading scheme (median):
  - Bottom half of results does not influence outcome
  - Stop turning in homework after having good median
- What’s the difference between final grades of students
  - Who submitted all and not all of homework
  - What would have been the final grade if we
    - Used average, while treating missing homework as zero
    - Used median only of submitted homework
Comparing Grading Schemes

• Separate students into two groups
  ▫ Those having all homework
  ▫ Those having missed homework

• Apply all three grading schemes to each student
  ▫ Average while treating missed homework as zero
  ▫ Median while treating missed homework as zero
  ▫ Median of submitted homework only

• Report the median of each of the groups
Comparing Grading Schemes

• Separate students into groups:

```cpp
// read all the records, separating them based on
// whether all homework was done
vector<student_info> did, didnt;
student_info student;
while (read(cin, student)) {
    if (did_all_homework(student))
        did.push_back(student);
    else
        didnt.push_back(student);
}

// check that both groups contain data
if (did.empty())
    cout << "No student did all the homework!" << endl;
if (didnt.empty())
    cout << "Every student did all the homework!" << endl;
```
Comparing Grading Schemes

- Test if student did all homework
  ```cpp
  bool did_all_homework(student_info const& s)
  {
    return find(s.homework.begin(), s.homework.end(), 0) == s.homework.end();
  }
  ```

- Student did all homework if no homework grade is zero
Standard Algorithm: find

- Find an entry in a sequence
  
  ```cpp
  std::find(begin, end, value);
  ```
  
  - Goes over the sequence `[begin, end)` and compares each element with `value`
  - Returns current position (iterator) as soon as the comparison evaluates to true for the first time
  - Essentially this finds the first element in the sequence matching the value
Comparing Grading Schemes

- Analyzing the grades
  - Three analyses to perform, all on two different data sets (groups of students)
  - Reporting requires both results at the same time
    - For each of the analysis types

- Encapsulate analysis types into a function each
  - Pass those functions to the reporting, together with the two data sets

- Reporting function interface:
  
  ```c++
  write_analysis(cout, "median", median_analysis, did, didnt);
  ```
Median Analysis of Grades

• Needed for write_analysis

// this function doesn't quite work
double median_analysis(vector<student_info> const& students)
{
    vector<double> grades;
    transform(students.begin(), students.end(),
              back_inserter(grades), grade);
    return median(grades);
}

• Doesn’t quite work! (Why?)
  ▫ Function grade() is overloaded
  ▫ Function grade() may throw exception
Standard Algorithm: transform

- The transform algorithm is like copy on steroids
  - It not only copies the input sequence, but calculates a new value on the fly
  - It invokes the function for each element and inserts the returned result instead of the original element:
    \[
    \text{transform(begin1, end1, begin2, func)}
    \]
  - The function ‘func’ is expected to have one parameter (the sequence element) and to return the value to insert
Median Analysis of Grades

• Create a indirection layer (as usual):
  ```cpp
double median_grade(student_info const & s)
  {
    try {
      return grade(s);  // throws if no homework
    }
    catch (domain_error) {
      return grade(s.midterm, s.final, 0);
    }
  }
```

• Now we can use it as:
  ```cpp
double median_analysis(vector<student_info> const & students)
  {
    vector<double> grades;
    transform(students.begin(), students.end(),
              back_inserter(grades), median_grade);
    return median(grades);
  }
```
Invoking Analysis Functions

• We define write_analysis as

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

• Parameter type to pass functions
• Return type ‘void’
Comparing Grading Schemes

• Pulling everything together:

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

    // read the student records and partition them
    // ... (see previous slides)

    // 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;
}
```
Analyzing Averages

• We need to calculate averages now (instead of medians):

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

• Calculating average grade:

```cpp
double average_grade(student_info const& s)
{
    return grade(s.midterm, s.final, average(s.homework));
}
```
Standard Algorithm: accumulate

- Unlike the other algorithms it’s declared in `<numeric>`
- Adds the values in the range denoted by its first two arguments, starting the summation with the value given by its third argument
- The type of the sum is the type of the third argument, crucial to write ‘0.0’
Analyzing Averages

- **Average analysis is straight forward now:**
  ```cpp
double average_analysis(vector<student_info> const& students)
{
    vector<double> grades;
    transform(students.begin(), students.end(),
        back_inserter(grades), average_grade);
    return median(grades);
}
```

- **Only difference is use of average_grade() instead of median_grade()**
Median of Completed Homework

- Optimistic assumption that the students' grades on the homework that they didn't turn in would have been the same as the homework that they did turn in

```cpp
// median of the nonzero elements of s.homework,
// or 0 if no such elements exist
double optimistic_median(student_info const& 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);
    return grade(s.midterm, s.final, median(nonzero));
}
```
Standard Algorithm: remove_copy

Library provides copying versions like ‘_copy’
- Equivalent to in-place versions except that they create a copy
- Therefore remove_copy() is copying equivalent of remove()

The remove() algorithm finds all values that match a given value and ‘removes’ those values from the container.
- All non-matching values are retained/copied
- The first two iterators denote the input sequence.
- The third denotes the beginning of the destination for the copy.
- Assumes that there is enough space in the destination (same as std::copy)

Classifying Students, Revisited

Here is what we had:

```cpp
// second try: correct but potentially slow
vector<student_info> extract_fails(vector<student_info>& students)
{
    vector<student_info> fail;
    vector<student_info>::size_type i = 0;

    // invariant: elements [0, i) of students represent passing grades
    while (i != students.size()) {
        if (fail_grade(students[i])) {
            fail.push_back(students[i]);
            students.erase(students.begin() + i);
        } else
            ++i;
    }

    return fail;
}
```
Classifying Students, Revisited

- We promised to show an algorithmic solution (instead of using list<>)
- We’ll show two solutions
  - Slower one: uses two passes over the data
  - And second one uses one pass over the data
Classifying Students, Revisited

- Two pass solution:

```cpp
template<typename T>
void partition(vector<T>& elements, int left, int right)
{
    int j = left;
    T pivot = elements[right];
    for (int i = left; i < right; i++)
    {
        if (elements[i] < pivot)
        {
            std::swap(elements[i], elements[j]);
            j++;
        }
    }
    std::swap(elements[j], elements[right]);
}
```

```cpp
vector<student_info> extract_fails(vector<student_info>& students)
{
    vector<student_info> fail;
    // create copy of student records while ignoring pass grades
    remove_copy_if(students.begin(), students.end(),
                   back_inserter(fail), pass_grade);

    // remove fail grades from original sequence
    students.erase(
                   remove_if(students.begin(), students.end(), fail_grade),
                   students.end());

    return fail;
}
```
Standard Algorithm: \texttt{remove\_copy\_if}

- Same as \texttt{remove\_copy()}, except it expects a predicate instead of the value
  - Will ‘remove’ all elements for which predicate returns true
  - New sequence will contain only elements for which predicate returned false
- The algorithm \texttt{remove\_if()} is similar to \texttt{remove()}, except that it takes a predicate
  - Does not really ‘remove’ elements, just copies non-matching elements to the front.
Standard Algorithm: remove_if

---

**Pass:**
- students.begin()

**Fail:**
- students.end()

---

**Remove_if:**
- pass
- pass
- fail
- fail
- pass
- fail
- pass

**Result of remove_if:**
- students.begin()
- result of remove_if
- students.end()

---

**Erase:**
- pass
- pass
- pass
- pass

**Students.end():**
- students.begin()
Member Function: erase

- Erases all the elements in the range delimited by the iterators passed in: [begin, end)
  - Changes the size of container
  - Frees up space, might invalidate iterators (depending on container type)
Classifying Students, Revisited

- Current solution calculated homework grades twice: remove_copy_if, remove_if
- What we really need is a way to partition our student records based on the pass/fail criteria
  - One pass solution
  - stable_partition
Classifying Students, Revisited

• Single pass solution:

```cpp
vector<student_info> extract_fails(vector<student_info>& students) {
    // partition input sequence based on pass_grade
    vector<student_info>::iterator iter =
        stable_partition(
            students.begin(), students.end(), pass_grade);
    // copy failed student records
    vector<student_info> fail(iter, students.end());
    // remove failed student records from original vector
    students.erase(iter, students.end());
    return fail;
}
```
Standard Algorithm: stable_partition

- Takes a sequence and rearranges its elements so that the ones that satisfy a predicate precede the ones that do not satisfy it.
- Two versions of this algorithm: partition and stable_partition
  - Algorithm partition might rearrange the elements within each category
  - Algorithm stable_partition keeps them in the same order aside from the partitioning.
- Both return an iterator to the first element of the second data partition
Standard Algorithm: stable_partition

partition:

erase:

students.begin()

students.end()
Classifying Students, Revisited

• Results:
  ▫ Algorithm based solutions are roughly as fast as list based solution presented earlier
  ▫ Algorithmic solutions are substantially better than vector base solution
  ▫ One pass solution is roughly twice as fast as two pass solution
Algorithms, Containers, and Iterators

- Important piece of information alert!
  - *Algorithms act on container elements—they do not act on containers*

- This call acts on elements only:
  ```cpp
  // library algorithm, as no container is changed
  remove_if(students.begin(), students.end(), fail_grade)
  ```

- But this changes the container:
  ```cpp
  // member function of container as erase() changes the vector
  students.erase(
    remove_if(students.begin(), students.end(), fail_grade),
    students.end()));
  ```
Algorithms, Containers, and Iterators

- Changing the container (erase, insert) invalidates iterators
  - Not only iterators pointing to erased elements
  - But also those pointing to elements after the erased ones
- Moving elements around (remove_if, partition) will change the element an iterator is referring to
  - *Be careful when holding on to iterators!*