Using Library Algorithms

Lectures 14

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

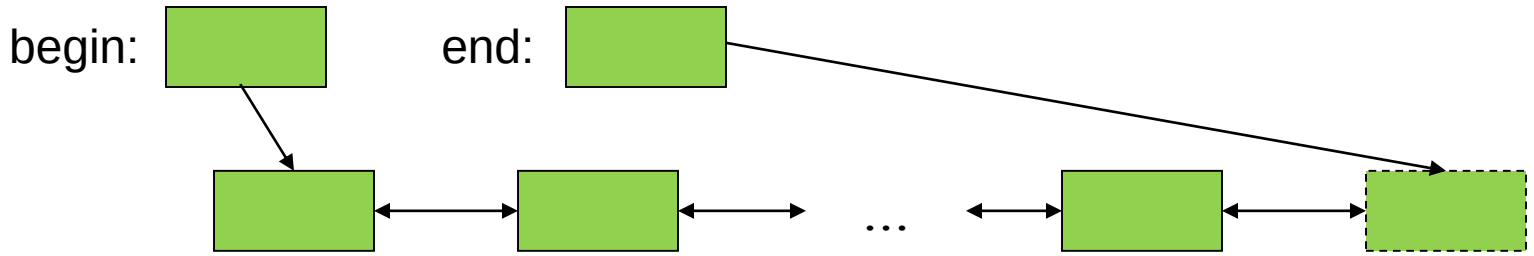

do{
beer++;
}while(sober);

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.

Basic Model: Pair of Iterators (Range)

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)



- An iterator is a type that supports the "iterator operations" of
  - ++ Point to the next element
  - * Get the element
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations *(e.g.*, --, +, and [ ])

Generic Algorithms

Analyzing Strings

- Looking back to picture concatenation:

```
// copy entire bottom picture, use iterators
for (auto 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(), std::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

```
std::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 the container `ret`:

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

Caveats: copy

- This will not work (why?):

```
std::copy(bottom.begin(), bottom.end(), ret);
// ret is not an iterator, but a container
```

- This will compile, but not work (why?):

```
std::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
  - `std::back_inserter` useful in other contexts as well

Another Copy Example

```
void f(std::vector<double> const& vd, std::list<int>& li)
{
    if (vd.size() < li.size())
        throw std::runtime_error("target container too small");

    std::copy(li.begin(), li.end(), vd.begin());   // note: different container types
                                                   // and different element types
                                                   // (vd better have enough elements
                                                   // to hold copies of li's elements)
    // ...
}
```

Input and Output Iterators

```
// we can provide iterators for output streams:

std::ostream_iterator<std::string> oo(std::cout);    // assigning to *oo is to
                                                     // write to cout

*oo = "Hello, ";     // meaning: std::cout << "Hello, "
++oo;                // "get ready for next output operation"
*oo = "world!\n";    // meaning: std::cout << "world!\n"

// we can provide iterators for input streams:

std::istream_iterator<std::string> ii(std::cin);    // reading *ii is to read a
                                                    // string from cin

std::string s1 = *ii;    // meaning: std::cin >> s1
++ii;                    // "get ready for the next input operation"
std::string s2 = *ii;    // meaning: std::cin >> s2
```

Make a Quick Dictionary (using a std::vector)

```
std::string from, to;
std::cin >> from >> to;     // get source and target file names

std::ifstream is(from);     // open input stream
std::ofstream os(to);       // open output stream

std::istream_iterator<std::string> ii(is); // make input iterator for stream
std::istream_iterator<std::string> eos;    // input sentinel (defaults to EOF)

std::ostream_iterator<std::string> oo(     // make output iterator for
    os, "\n");                             // stream
                                           // append "\n" each time

std::vector<std::string> b(ii, eos);       // b is a vector initialized
                                           // from input
std::sort(b.begin(), b.end());             // sort the buffer
std::unique_copy(b.begin(), b.end(), oo);  // copy buffer to output,
                                           // discard replicated values
```

An Input File

This lecture and the next presents the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with “policies”.

Part of the Output

(data)
(processing)
(the
C++
First,
Function
I
It
STL
The
This
a
algorithms
algorithms.
an
and
are
concepts,
containers
data
dealing
examples
extensible
finally
Framework
fundamental
general
ideal,
in
is
iterator
key
lecture
library).
next
notions
objects
of
parameterize
part
present
presented.
presents
program.
sequence
…

More Generic Algorithms

Splitting Strings: Take 1

```
std::vector<string> split(std::string const& s)
{
    std::vector<std::string> words;
    typedef std::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

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

        // find end of next word
        auto j = std::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;
}
```

Splitting Strings: Take 2

- Here are the predicates:

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

// false if the argument is whitespace, true otherwise
bool not_space(char c)
{
    return !std::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:

```
bool is_palindrome(std::string const& s)
{
    return std::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 `std::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
  - There is an additional version allowing to specify the end of the second sequence as well:

    ```
    std::equal(begin1, end1, begin2, end2)
    ```

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:

```
bool is_palindrome2(std::string const& s)
{
    auto it = s.begin();
    std::advance(it, s.size() % 2 ? s.size()/2 + 1 : s.size()/2);
    return std::equal(s.begin(), it, s.rbegin());
}

bool is_palindrome3(std::string const& s)
{
    return std::equal(s.begin(),
        std::next(s.begin(), s.size() % 2 ? s.size()/2 + 1 : s.size()/2),
        s.rbegin());
}
```

Standard Algorithm: advance (next)

- 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
- The algorithm `next` is similar, except that it returns the new iterator

Finding URLs

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

```
std::vector<std::string> find_urls(std::string const& s)
{
    std::vector<std::string> ret;
    // look through the entire input
    auto b = s.begin(), e = s.end();
    while (b != e) {
        // look for one or more letters followed by ://
        b = url_beg(b, e);
        // if we found it
        if (b != e) {
            auto after = url_end(b, e);             // get the rest of the URL
            ret.push_back(std::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:

- 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:

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

- Where `not_url_char` is our predicate:

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

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

Standard Algorithm: find

- Find an entry in a sequence

```
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

Finding URLs: url_begin

- We place several iterators inside our string:

Finding URLs: url_begin

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

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

    auto i = b;   // i marks where the separator was found
    while ((i = std::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
            auto beg = i;
            while (beg != b && std::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 `std::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

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:

```
// read all the records, separating them based on
// whether all homework was done
std::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())
    std::cout << "No student did all the homework!" << std::endl;
if (didnt.empty())
    std::cout << "Every student did all the homework!" << std::endl;
```

Comparing Grading Schemes

- Test if student did all homework

```
bool did_all_homework(student_info const& s)
{
    return std::find(s.homework.begin(), s.homework.end(), 0) ==
           s.homework.end();
}
```

- Student did all homework if no homework grade is zero

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:

```
write_analysis(std::cout, "median", median_analysis, did, didnt);
```

Median Analysis of Grades

- Needed for write_analysis

```
// this function doesn't quite work
double median_analysis(std::vector<student_info> const& students)
{
    std::vector<double> grades;
    std::transform(
        students.begin(), students.end(),
        std::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:

```
std::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 an indirection layer (as usual):

```
double median_grade(student_info const& s)
{
    try {
        return grade(s);      // throws if no homework
    } catch (std::domain_error) {
        return grade(s.midterm, s.final, 0);
    }
}
```

- Now we can use it as:

```
double median_analysis(std::vector<student_info> const& students)
{
    std::vector<double> grades;
    std::transform(students.begin(), students.end(),
        std::back_inserter(grades), median_grade);
    return median(grades);
}
```

Invoking Analysis Functions

- We define write_analysis as

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

- Parameter type to pass functions
- Return type ‘void’

Comparing Grading Schemes

- Pulling everything together:

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

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

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

    return 0;
}
```

Analyzing Averages

- We need to calculate averages now (instead of medians:

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

- Calculating average grade:

```
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:

```
double average_analysis(std::vector<student_info> const& students)
{
    std::vector<double> grades;
    std::transform(students.begin(), students.end(),
        std::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

```
// median of the nonzero elements of s.homework,
// or 0 if no such elements exist
double optimistic_median(student_info const& s)
{
    std::vector<double> nonzero;
    std::remove_copy(s.homework.begin(), s.homework.end(),
        std::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 `std::remove_copy()` is copying equivalent of `std::remove()`
- The `std::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

Classifying Students, Revisited

- Here is what we had:

```
// second try: correct but potentially slow
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    std::vector<student_info> fail;
    std::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 `std::list<T>` or `list<T>`)
- We'll show two solutions
  - Slower one: uses two passes over the data
  - And second one uses one pass over the data
- Both solutions expose complexity of O(N)

Classifying Students, Revisited

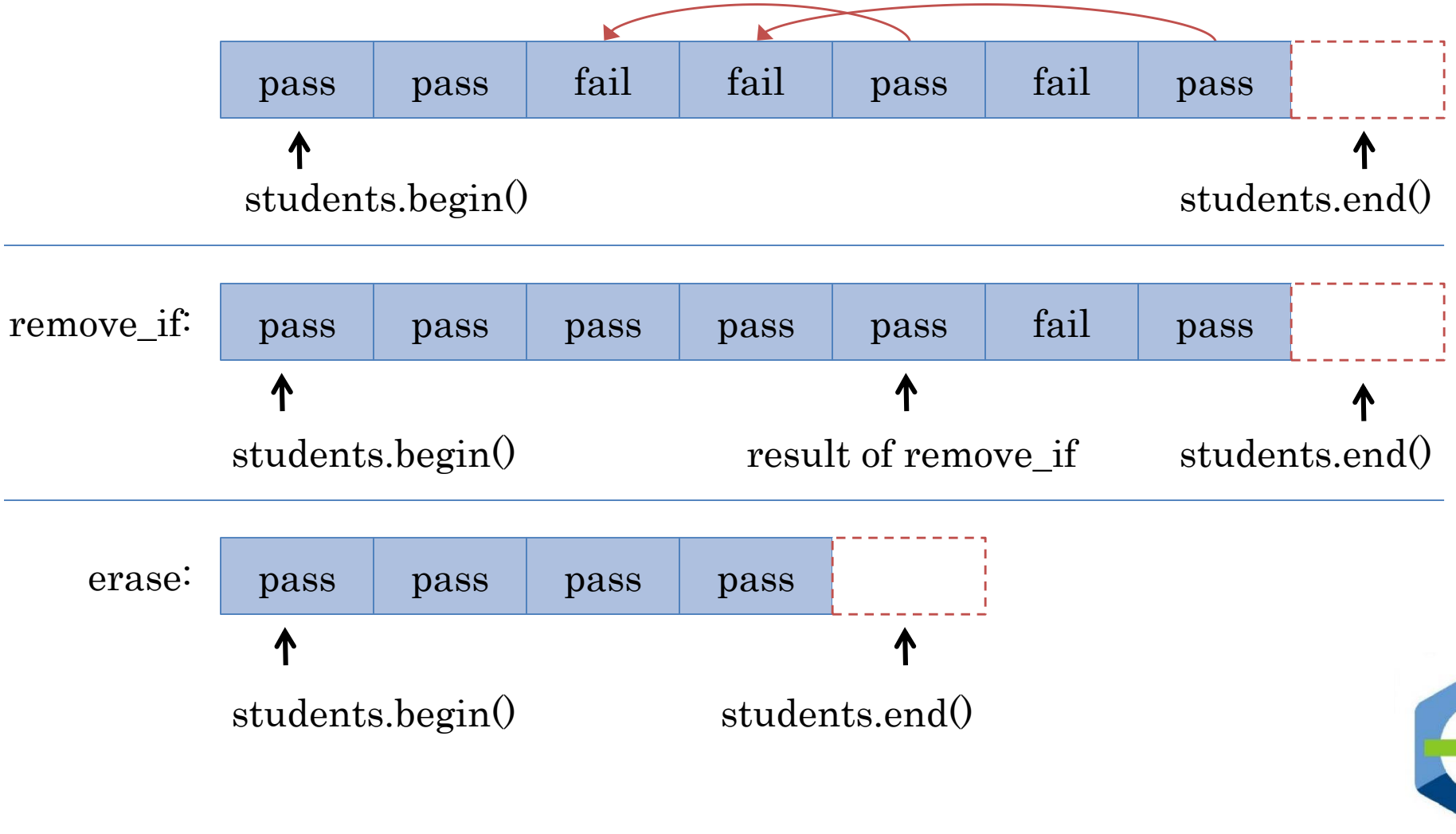
- Two pass solution:

```
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    std::vector<student_info> fail;

    // create copy of student records while ignoring pass grades
    std::remove_copy_if(students.begin(), students.end(),
        std::back_inserter(fail), pass_grade);

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

    return fail;
}
```

Standard Algorithm: remove_copy_if

- Same as `std::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 `std::remove_if()` is similar to `std::remove()`, except that it takes a predicate
  - Does not really 'remove' elements, just copies non-matching elements to the front.

Standard Algorithm: remove_if



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: `std::remove_copy_if`, `std::remove_if`
- What we really need is a way to partition our student records based on the pass/fail criteria
  - One pass solution
  - `std::stable_partition`

Classifying Students, Revisited

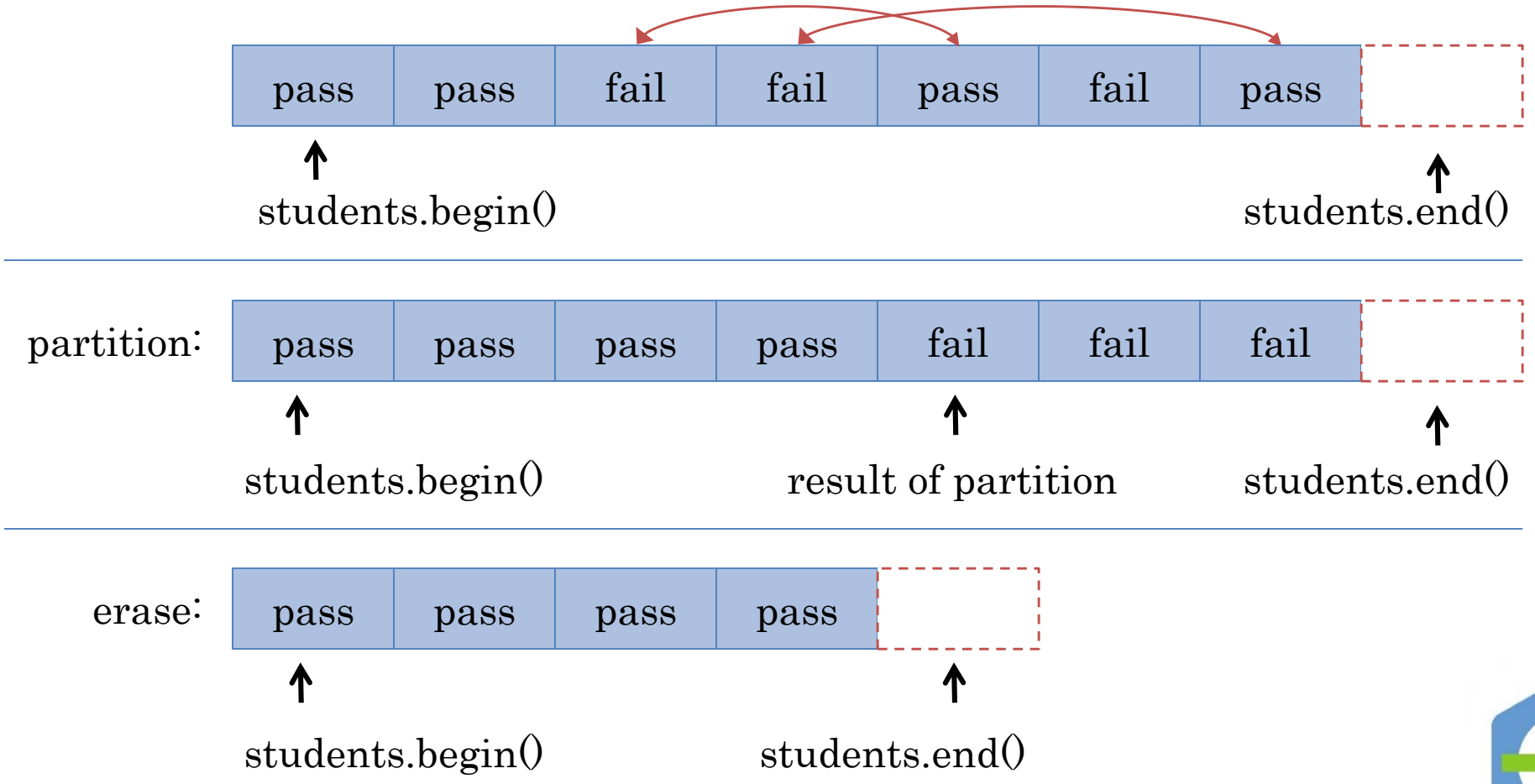
- Single pass solution:

```
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    // partition input sequence based on pass_grade
    auto iter = std::stable_partition(
            students.begin(), students.end(), pass_grade);

    // copy failed student records
    std::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: `std::partition` and `std::stable_partition`
  - Algorithm partition might rearrange the elements within each category
  - Algorithm `std::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



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:

```
// library algorithm, as no container is changed
std::remove_if(students.begin(), students.end(), fail_grade)
```

- But this changes the container:

```
// member function of container as erase() changes the vector
students.erase(
    std::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 (`std::remove_if`, `std::partition`) will change the element an iterator is referring to
  - *Be careful when holding on to iterators!*

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP