Using Sequential Containers

Lecture 11

Hartmut Kaiser

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

Software Development Notes

Systems and Modeling

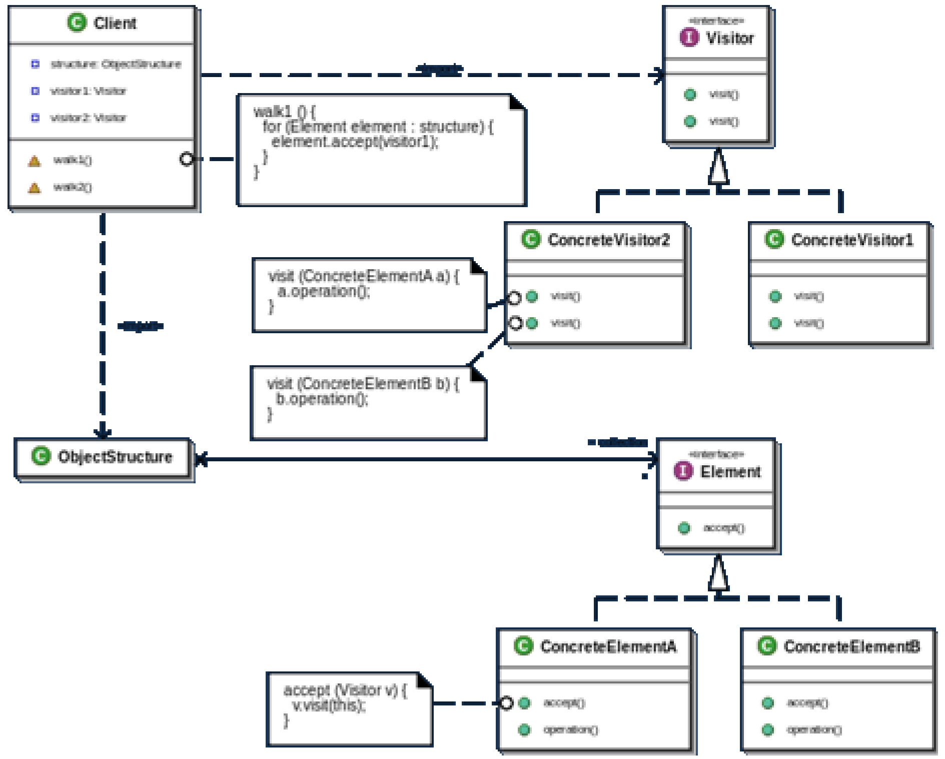
- The use of modeling has a rich history in all engineering disciplines

- The four basic principles of modeling
  - The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped
  - Every model may be expressed at different levels of precision
  - The best models are connected to reality
  - No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models

The Unified Modeling Language User Guide, by G. Booch, J. Rambaugh & I. Jacobson

Unified Modeling Language (UML)

- Unified Modeling Language (UML) is a graphical meta-language for visualizing, specifying, and documenting software systems



```
walk1 () {
  for (Element element : structure) {
    element.accept(visitor1);
  }
}
```

```
visit (ConcreteElementA a) {
  a.operation();
}
```

```
visit (ConcreteElementB b) {
  b.operation();
}
```

```
accept (Visitor v) {
  v.visit(this);
}
```

Unified Modeling Language (UML)

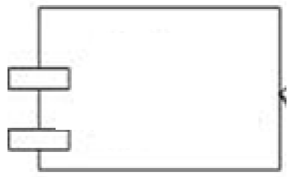
- Developers (the 3 amigos)
  - Grady Booch (Rational Software Corp)
  - James Rumbaugh (General Electric)
  - Ivar Jacobson (Objectory)
- Object Management Group (OMG)
  - An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications
  - UML Standards
    - UML 1.0 (1995)
    - UML 1.x (1995)
    - UML 2.0 (2005), currently most widely used
    - UML 2.x (2006-2015), minor revisions

The Component Model

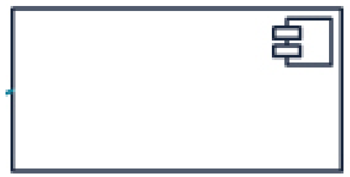
- The component model illustrates the software components that will be used to build the system
- Components are high level aggregations of smaller software pieces, and provide a 'black box' building block approach to software construction
- Typically a component is made up of many internal classes and packages of classes
  - It may be assembled from a collection of smaller components

The Component Model
Component Notation

- The component diagram shows the relationship between software components, their dependencies, communication, location and other conditions
- The graphical representation of a component is a rectangle with tabs:

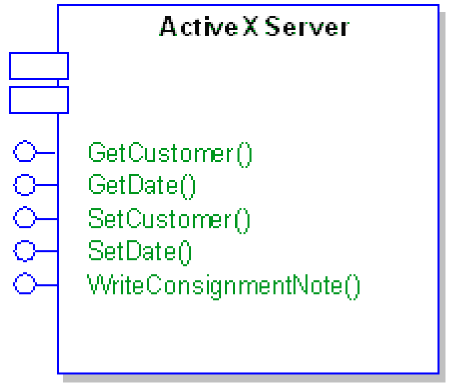


or



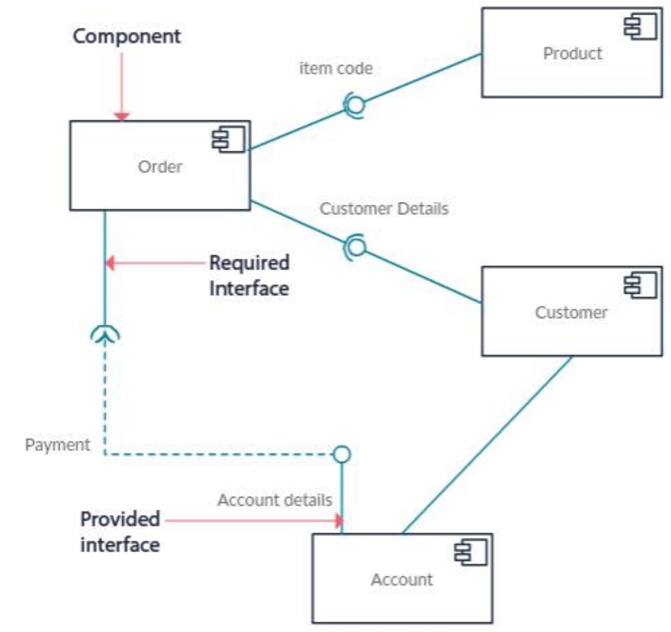
The Component Diagram Interfaces

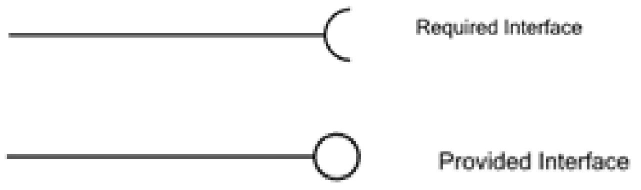
- Components may also expose interfaces
  - These are the visible entry points or services that a component is advertising and making available to other software components and classes
  - Modeled as lollypops



Component Diagram

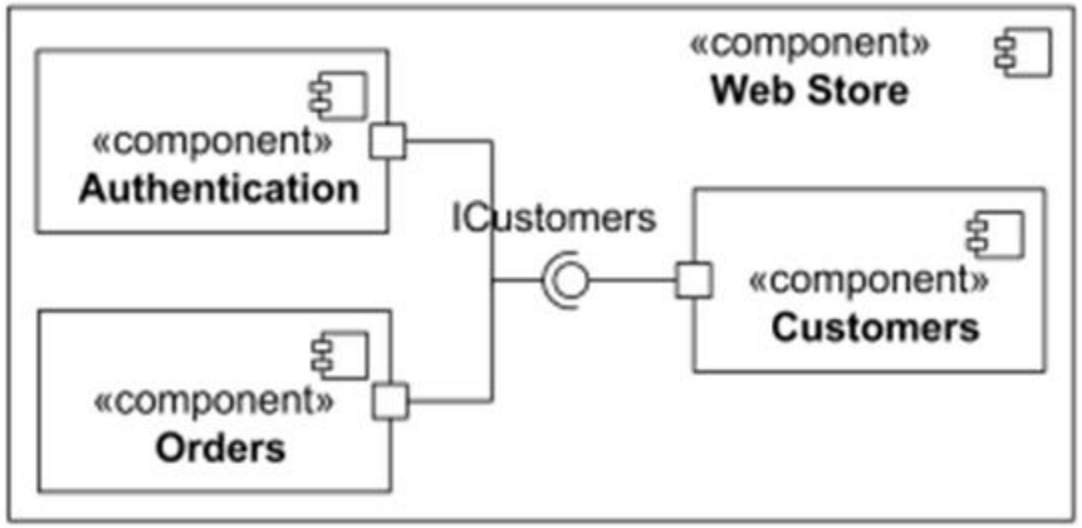
- Components communicate with each other using interfaces (lollypop: circle on a line)
- The interfaces are linked using connectors (wine glass: half circle on a line)





The Unified Modeling Language User Guide, by G. Booch, J. Rambaugh & I. Jacobson
https://creately.com/blog/diagrams/uml-diagram-types-examples/

Sub-Components



https://www.uml-diagrams.org/component-diagrams-reference.html

Analyzing Student Grades

Abstract

- We start looking beyond vector and string. We will focus on sequential containers and demonstrate a couple of problems we can solve when applying them.
- The standard library's architecture will start to get visible. That will help us to start understanding how to use all of the different containers in the standard library.

Organizing Data

- Let’s write program analyzing student grades for a whole course (many students)
- Read grades from a file:

```
Smith 93 91 47 90 92 73 100 87
Carpenter 75 90 87 92 93 60 0 98
```

  - i.e. Name Midterm Final Homework-Grades
- We want to produce output (overall grade)

```
Carpenter      90.4
Smith          86.8
```

  - Alphabetical, formatting vertically lining up

Organizing Data

- Need to store all student data
  - Sorted by name
  - Line up: find longest name
- Let’s assume we can store all data about one student (`student_info`)
  - All students data: `std::vector<student_info>`
  - Should be `Regular`! Maybe `TotallyOrdered`, most likely `StrictWeaklyOrdered`
- Set of auxiliary functions to work with that data
  - Solve the overall problem using those

Organizing Data

- We need to hold all data items related to one student together:

```
// hold all information related to a single student
struct student_info
{
    std::string name;                // students name
    double midterm, final;           // midterm and final exam grades
    std::vector<double> homework;    // all homework grades
};
```

- This is a new type holding four items (members)
  - We can use this type to define new objects of this type
  - We can store the information about all students in a

```
std::vector<student_info> students;
```

Reading Data for one Student

- Very similar to what we already have seen:

```
// read all information related to one student
std::istream& operator>>(std::istream& in, student_info& s)
{
    // read the students name, midterm, and final exam grades
    in >> s.name >> s.midterm >> s.final;
    // read all homework grades for this student
    return read_hw(in, s.homework);
}
```

- Any input error will cause all subsequent input to fail as well
  - Can be called repeatedly

Reading Data for one Student

```
std::istream& read_hw(std::istream& in, std::vector<double>& hw)
{
    if (in) {
        hw.clear();    // get rid of previous content

        // read homework grades
        double x;
        while (in >> x)
            hw.push_back(x);

        // clear the stream so that input will work for
        // the next student
        in.clear();
    }
    return in;
}
```

Reading all Student Records

- Invoke `operator>>` as long as we succeed:

```
std::vector<student_info> students;    // all student records
std::string::size_type maxlen = 0;     // length of longest name

// read and store all the records, find the length of the longest name
student_info record;
while (std::cin >> record) {
    maxlen = std::max(maxlen, record.name.size());
    students.push_back(record);
}
```

- Function `std::max()` is peculiar
  - Both arguments need to have same type

Calculate Final Grade

- Calculate grade based on data read:

Exercise: rewrite `median()` using `std::nth_element`.

```
// compute the median of a std::vector<double>
// note: calling this function copies the whole vector
double median(std::vector<double> vec)
{
    auto size = vec.size();
    if (size == 0) throw std::domain_error("vector is empty, median undefined");

    std::sort(vec.begin(), vec.end());
    auto mid = size / 2;
    return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}

// Calculate the final grade for one student
double grade(student_info const& s)
{
    return 0.2 * s.midterm + 0.4 * s.final + 0.4 * median(s.homework);
}
```

Sort Student Data

- We know that sorting can be done using sort():

```
std::vector<double> vec;
std::sort(vec.begin(), vec.end());
```

- Let's do the same for all students:

```
std::vector<student_info> students;
std::sort(students.begin(), students.end());
```

- Not quite right, why?
  - What criteria to use for sorting?
  - What does it mean to sort the vector of students?
  - How to express the need to sort 'by name'?

Sorting Student Data

- Normally, sort() uses the operator< to determine order
  - Makes no sense for student_info's!
- We can teach sort() how to order by specifying a predicate
  - A function returning a `bool` taking two arguments of the type to be compared
  - Returns `true` if the first argument is smaller than the second (whatever that means)

Sorting Student Data

- We can teach sort() how to order by specifying a predicate:

```
// compare two student_info instances, return whether 'x‘
// is smaller than 'y' based on comparing the stored names
// of the students
bool compare(student_info const& x, student_info const& y)
{
    return x.name < y.name;    // get_name(x) < get_name(y)
}
```

- Now, we can use this function as:

```
std::vector<student_info> students;
std::sort(students.begin(), students.end(), compare);
```

Sorting Student Data

- Alternatively, we could define an appropriate operator

```
// compare two student_info instances, return whether 'x‘
// is smaller than 'y' based on comparing the stored names
// of the students
bool operator<(student_info const& x, student_info const& y)
{
    return x.name < y.name;
}
```

- Now, we would be able to use this function as:

```
std::vector<student_info> students;
std::sort(students.begin(), students.end());
```

- But is this what we really want?

Sorting Student Data

- Alternative: lambda function:

```
// sorting the student data using a lambda function
sort(students.begin(), students.end(),
    [](student_info const& x, student_info const& y)
    {
        return x.name < y.name;
    }
);
```

  - Note: this lambda has no explicit return type
    - Although, it could be specified (`-> bool`)
- Much nicer! Everything is in one place
  - This ordering is called `StrictWeakOrdering`
  - Weaker than `TotallyOrdered` as we might want to sort by grades, etc.

Sorting Student Data

- A `StrictWeakOrdering` is a Binary Predicate that compares two objects, returning true if the first precedes the second
  - Applying `TotalOrdering` to equivalence classes
  - Invoke function on an element and totally order what it returns
- `StrictWeakOrdering`
  - Partial ordering:
    - Irreflexivity: `!f(x, x)`
    - Antisymmetry: `f(x, y) ⇔ !f(y, x)`
    - Transitivity: `f(x, y) && f(y, z) ⇔ f(x, z)`
  - Transitivity of equivalence
    - if x $\cong y$ and y $\cong z$, then x $\cong z$

Generating the Report

- Now we're ready to generate the report:

```
for (std::vector<student_info>::size_type i = 0; i != students.size(); ++i) {
    // write the name, padded on the right side to maxlen + 1 characters
    std::cout << students[i].name
              << std::string(maxlen + 1 - students[i].name.size(), ' ');

    // compute and write the grade
    try {
        double final_grade = grade(students[i]);
        std::streamsize prec = cout.precision();
        std::cout << std::setprecision(3) << final_grade << std::setprecision(prec);
    }
    catch (std::domain_error e) {
        std::cout << e.what();
    }
    std::cout << std::endl;
}
```

Generating the Report

- Now we're ready to generate the report:

```
for (student_info const& si: students)
{
    // write the name, padded on the right side to maxlen + 1 characters
    std::cout << si.name
              << std::string(maxlen + 1 - si.name.size(), ' ');

    // compute and write the grade
    try {
        double final_grade = grade(si);
        std::streamsize prec = cout.precision();
        std::cout << std::setprecision(3) << final_grade << std::setprecision(prec);
    }
    catch (std::domain_error e) {
        std::cout << e.what();
    }
    std::cout << std::endl;
}
```

Separating Students into Categories

- Sort out failed students
  - Who failed?
  - Remove from our data
- Create a new vector of `student_data` containing only students who succeeded:

```
// predicate to determine whether a student failed
bool fail_grade(student_info const& s)
{
    return grade(s) < 60;
}
```

- Push student data onto one of two containers based on this predicate

Separating Students into Categories

- What's wrong here? (Hint: what's the memory consumption?)

```
// separate passing and failing student records: first try
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    std::vector<student_info> pass, fail;
    for (std::vector<student_info>::size_type i = 0;
          i != students.size(); ++i)
    {
        if (fail_grade(students[i]))
            fail.push_back(students[i]);
        else
            pass.push_back(students[i]);
    }
    students = pass;
    return fail;
}
```

Separating Students into Categories

- Requires twice as much memory
  - Each record is held twice
- Better to copy failed students, removing the data from original vector
  - How to remove elements from a vector?
  - Slow, too slow for larger amounts of data.
    - Why?
    - What happens if all students have failed?
  - This can be solved by either using a different data structure or by modifying the algorithm

Erasing Elements in Place

- Slow, but direct solution (Why is it slow?)

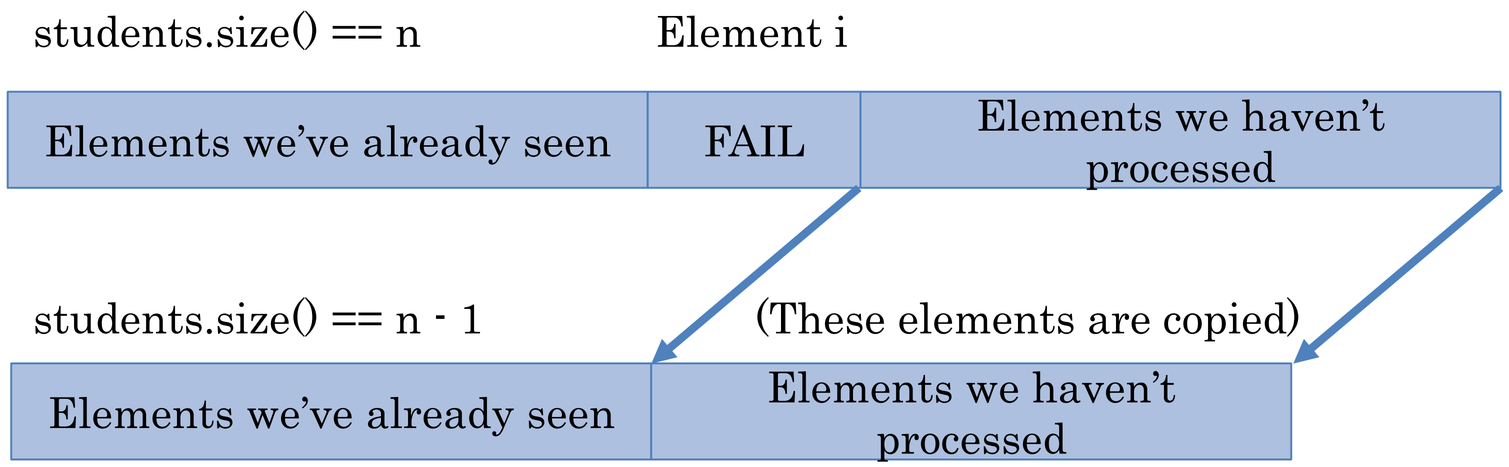
```
// second try: correct but potentially slow
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    std::vector<student_info> fail;
    std::vector<student_info>::size_type i = 0;

    // 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;
}
```

Erasing Elements in Place

- The erase() function takes a special type 'pointing' (referring) to the element to erase, i.e. an iterator:

```
students.erase(students.begin() + i);
```


students.size() == n
Element i
Elements we've already seen
FAIL
Elements we haven't processed
students.size() == n - 1
(These elements are copied)
Elements we've already seen
Elements we haven't processed

Erasing Elements in Place

- Caution: why will this fail?

```
// this code will fail because of misguided optimization
auto size = students.size();
while (i != size) {
    if (fail_grade(students[i])) {
        fail.push_back(students[i]);
        students.erase(students.begin() + i);
    } else
        ++i;
}
```

Sequential Versus Random Access

- Both versions share a non-obvious property
  - The elements are accessed sequentially only
  - We used integer 'i' as an index, which hides that
    - Need to analyze every operation on 'i' to verify
    - We might access student data in arbitrary order
- Every container type has its performance characteristics for certain operations
  - By knowing what access pattern we use we can utilize the 'best' container type

Sequential Versus Random Access

- Let’s restrict our access to being sequential
- The standard library exposes special types we can use to express this intent: *Iterators*
  - By choosing the right type of iterator we ‘tell’ the library what access pattern we need to support
  - Allows for optimal selection of the underlying algorithm implementation

Iterators

Iterators

- Our code uses the index for
  - Access of an element

```
fail_grade(students[i])
```

  - Move to the next element (increment 'i')

```
    while (i != students.size()) {
        // work gets done here; but doesn't change the value of i
        ++i;
    }
```

- We use index for sequential access only!
- But there is no way of telling the library about this

Iterators

- Iterators are special types
  - Identify a container and an element in the container
  - Let us examine the value stored in that element
  - Provide operations for moving between elements in the container
  - Restrict the available operations in ways that correspond to what the container can handle efficiently

Iterators

- Code using iterators is often analogous to index based code:

```
// code based on indicies
for (std::vector<student_info>::size_type i = 0;
     i != students.size(); ++i)
{
    std::cout << students[i].name << std::endl;
}

// code based on iterators
for (std::vector<student_info>::const_iterator iter = students.begin();
     iter != students.end(); ++iter)
{
    std::cout << (*iter).name << std::endl;    // same as iter->name
}
```

Iterator Types

- Every standard container, such as `std::vector`, defines two associated iterator types:

```
        container_type::iterator
        container_type::const_iterator
```

  - Where container_type is the container (`std::vector<student_info>`)
  - Use `iterator` to modify the element, `const_iterator` otherwise (read only access)

- Note, that we don‘t actually see the actual type, we just know what we can do with it.
  - Abstraction is selective ignorance!

Iterators

- Code using iterators is often analogous to index based code:

```cpp
// code based on indicies
for (auto i = 0; i != students.size(); ++i)
{
    cout << students[i].name << endl;
}

// code based on iterators, we don't care about the actual iterator type
for (auto iter = students.begin(); iter != students.end(); ++iter)
{
    cout << (*iter).name << endl;
}

// code based on iterators, we don't care about the actual element type
for (auto const& s : students)
{
    cout << s.name << endl;
}
```

Iterator Types

- Every `container_type::iterator` is convertible to the corresponding `container_type::const_iterator`
  - students.begin() returns an iterator, but we assign it to a const_iterator
- Opposite is not true! Why?

Iterator Operations

- Containers do not only expose their (specific) iterator types, but also actual iterators:

```
students.begin(), students.end()
```

  - begin(): 'points' to the first element
  - end(): 'points' to the element after the last one
- Iterators can be *compared*:

```
iter != students.end()
```

  - Tests, whether both iterators refer to the same element
- Iterators can be *incremented*:

```
++iter
```

  - Make the iterator 'point' (refer) to the next element

Iterator Operations

- Iterators can be *dereferenced*:

```
*iter
```

  - Evaluates to the element the iterator refers to
- In order to access a member of the element the iterator refers to, we write:

```
(*iter).name
```

  - (why not: `*iter.name` ?)
- Syntactic sugar, 100% equivalent:

```
iter->name
```

Iterator Operations

- Some iterators can get a number added

```
students.erase(students.begin() + i);
```

  - Overloaded operator+, makes the iterator refer to the 'i' –s element after begin
  - Equivalent to invoking ++ 'i' times
  - Defined only for iterators from *random access* containers
    - `std::vector`, `std::string` are random access (indexing is possible)
    - Will result in compilation error for sequential (non-random access) containers

Erasing Elements in Place

- Slow, but direct solution

```
// 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;
}
```

Erasing Elements in Place

- Still slow, but without indexing:

```
// version 3: iterators but no indexing
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    std::vector<student_info> fail;

    auto iter = students.begin();
    while (iter != students.end()) {
        if (fail_grade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);    // watch out! Why?
        } else
            ++iter;
    }
    return fail;
}
```

Iterator Invalidation

- What happens to an iterator if the element it refers to is deleted?
  - It is invalidated
  - Certain containers invalidate all iterators after the deleted element as well (vectors)
- For that reason erase() returns the next (valid) iterator:

```
iter = students.erase(iter);
```

Same Problem as before

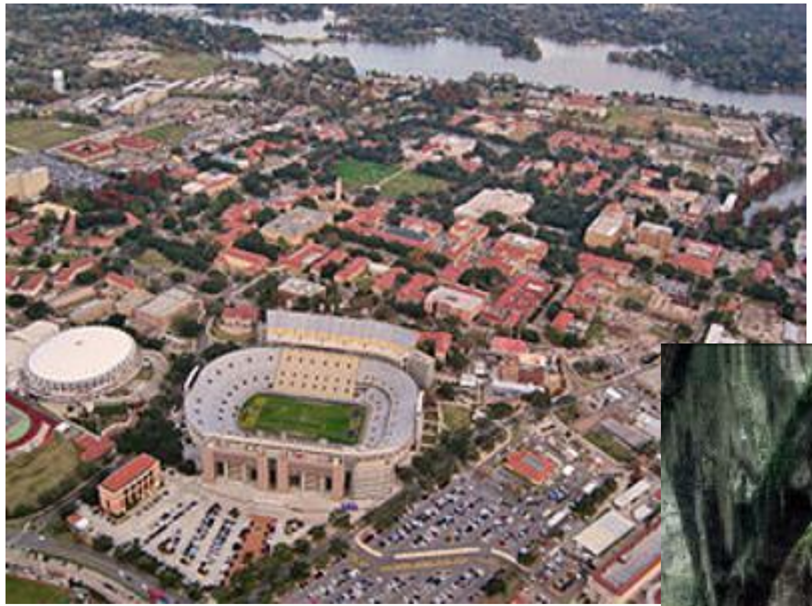
- Why does this code fail:

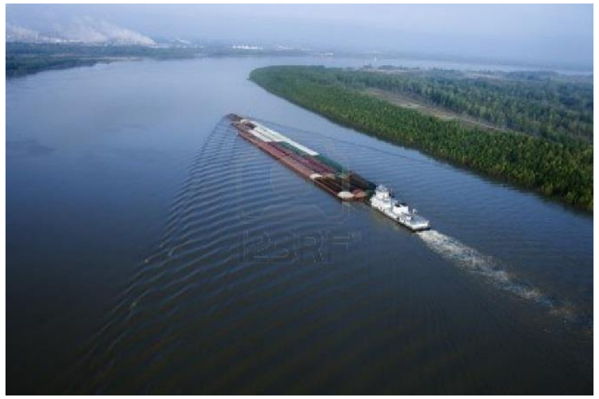
```
// this code will fail because of misguided optimization
auto iter = students.begin();
auto end_iter = students.end();
while (iter != end_iter) {
  // ... erase elements from students without updating end_iter
}
```

- End iterator is invalidated as well when element is erased!

What's the Problem with `std::vector`?

- For small inputs, vector works just fine, larger inputs cause performance degradation
  - Vector is optimized for fast access to arbitrary elements and for fast addition to the end
  - Inserting or removing from the middle is slow.
    - All elements after the inserted/removed element need to be moved in order to preserve fast random access
    - Our algorithm has quadratic performance characteristics
  - Let's utilize a different data structure:
    - Next lecture: *The list type*


LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP