Using Sequential Containers

Lecture 8
Hartmut Kaiser
hkaiser@cct.lsu.edu
http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Programming Principle of the Day

• Principle of least astonishment (POLA/PLA)
  • The principle of least astonishment is usually referenced in regards to the user interface, but the same principle applies to written code.
  • Code should surprise the reader as little as possible. That means following standard conventions, code should do what the comments and name suggest, and potentially surprising side effects should be avoided as much as possible.

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.
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:

```cpp
// 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?)

```cpp
// separate passing and failing student records: first try
vector<student_info> extract_fails(vector<student_info>& students)
{
    vector<student_info> pass, fail;
    for (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?)

```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;
}
```
**Erasing Elements in Place**

- The `erase()` function takes a special type ‘pointing’ (referring) to the element to erase:

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

<table>
<thead>
<tr>
<th>students.size() == n</th>
<th>Element i</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FAIL</strong></td>
<td>Elements we haven’t processed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>students.size() == n - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements we’ve already seen</td>
</tr>
<tr>
<td>Elements we haven’t processed</td>
</tr>
</tbody>
</table>
Erasing Elements in Place

• Caution: why will this fail?

```cpp
// 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

- Our code uses the index for
  - Access of an element
    
    \[ \text{fgrade(students[i])} \]
  - Move to the next element (increment ‘i’)
    
    \[ \text{while (i != students.size())} \{ \]
    
    // 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:

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

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

• Every standard container, such as vector, defines two associated iterator types:
  
  container_type::iterator
  container_type::const_iterator

  • Where container_type is the container (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, C++11

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

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

```
// code based on iterators using C++11 (VS2010, g++4.2)
for (auto iter = students.begin(); iter != students.end(); ++iter)
{
    cout << (*iter).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 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
  • vector, string are random access (indexing is possible)
  • Will result in compilation error for sequential containers
Erasing Elements in Place

• Slow, but direct solution

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

```cpp
// version 3: iterators but no indexing
vector<student_info> extract_fails(vector<student_info>& students)
{
    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 iterator:

```
iter = students.erase(iter);
```
Same problem as Before

• Why does this code fail:

```cpp
// this code will fail because of misguided optimization
auto iter = students.begin();
auto end_iter = students.end();
while (iter != end_iter) {
    // ...
}
```

• End iterator is invalidated as well when element is erased!
What’s the problem with 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*