CSCI-1200 Data Structures — Fall 2025
Lecture 9 — Iterators & STL Lists

Review from Lectures 7 & 8

- Implementing our own version of STL `vector`
- Classes with dynamically-allocated memory – we must write copy constructor, assignment operator, & destructor
- Algorithm Analysis, Formal Definition of Big O Notation

Today

- Another `vector` operation: `pop_back`
- *Erasing items* from vectors is inefficient!
- Iterators and iterator operations
- STL `lists` are a different sequential container class.
- Differences between STL `list` and STL `vector`
- `Vec` iterator implementation
- Motivating example: A program to manage course enrollment with waiting list.

9.1 Review: Constructors, Assignment Operator, and Destructor

From an old test: Match up the line of code with the function that is called. Each letter is used exactly once.

☐	`Foo f1;`	a)	assignment operator
☐	`Foo* f2;`	b)	destructor
☐	`f2 = new Foo(f1);`	c)	copy constructor
☐	`f1 = *f2;`	d)	default constructor
☐	`delete f2;`	e)	none of the above

9.2 Amortized Analysis (*a.k.a. Average*) of STL `vector::push_back`

```
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    m_alloc *= 2;
    if (m_alloc < 1) m_alloc = 1;
    T* new_data = new T[ m_alloc ];
    for (size_type i=0; i<m_size; ++i)
      new_data[i] = m_data[i];
    delete [] m_data;
    m_data = new_data;
  }
  // Add the value at the last location and increment the bound
  m_data[m_size] = val;
  ++ m_size;
}
```

- What is the total cost of 1024 `push_back` operations on an initially empty STL `vector`?
  What is the *average* cost of a single `push_back` operation?

`push_back` #	`m_alloc`	resize "cost"	edit "cost"	total "cost" of each `push_back`
1	0→1	1	1	2
2	1→2	2	1	3
3	2→4	4	1	5
4	4		1	1
5	4→8	8	1	9
6	8		1	1
7	8		1	1
8	8		1	1
9	8→16	16	1	17
10	16		1	1
...			...	...
17	16→32	32	1	33
...			...	...
33	32→64	64	1	65
...			...	...
65	65→128	128	1	129
...			...	...
129	128→256	256	1	257
...			...	...
257	256→512	512	1	513
...			...	...
513	512→1024	1024	1	1025
...			...	...
1024	1024		1	1
TOTAL COST for 1024 `push_back`s:		$\sum_{k=0}^{10} 2^k = 2^{k+1} = 2047$	1024	3071
AVERAGE COST:				3071/1024 = 3 = O(1)

- This analysis seems complicated. I might not have figured this out on my own. Will I be expected to do Big O Notation that is this complicated? How do I learn how to do Big O Notation?
- We will be practicing lots and lots of simple Big O Notation problems throughout the rest of the term. You will get the hang of it with practice, and asking questions during lab and office hours.
- You will learn how to handle more complicated algorithm analysis problems in later courses including CSCI 2200 Foundations of Computer Science *and* CSCI 2300 Introduction to Algorithms!

9.3 Remove the last item from an STL `vector`: `pop_back`

- We have seen how `push_back` adds a value to the end of a vector, increasing the size of the vector by 1. There is a corresponding function called `pop_back`, which removes the last item in a vector, reducing the size by 1.
- There are also vector functions called `front` and `back` which denote (and thereby provide access to) the first and last item in the vector, allowing them to be changed. For example:

```
vector<int> a(5,1); // a has 5 values, all 1
a.pop_back();       // a now has 4 values
a.front() = 3;      // equivalent to the statement, a[0] = 3;
a.back() = -2;      // equivalent to the statement, a[a.size()-1] = -2;
```

- **Exercise:** How efficiently can we implement `pop_back`? What is the Big O Notation of `pop_back`?

9.4 Remove an item from the middle of an STL `vector`

- An important characteristic of the STL `vector` container class is that the **items are stored in the order the user inserted them** with `push_back`. Even if we dynamically re-size the structure, the **order of the elements will be preserved**.
- For example, a course registration system could use one STL `vector` to maintain the names of students who are in the course and – if the course is full (the enrollment cap has been reached) – a second STL `vector` to maintain the `waiting list` of students who want to add the course. It is critical that the order of the second vector be preserved – students who joined the waiting list early in the registration period should be offered spots in the course before students who joined the waiting list at a later time.
- Use `pop_back` to write a function named `erase_at_index` that removes the specified item, but preserves the order or sequence of all other elements in the container:

```
template <class T>
void erase_at_index(std::vector<T> &v, int i) {









}
```

- What is the Big O Notation of `erase_at_index`?
- In some use cases we might not need to maintain the order of data in a vector. Perhaps we are just collecting data that we will sum up, or sort alphabetically, etc. In our registration example, we do not need to preserve the insertion order of students who are in the course. The instructor may choose to sort students alphabetically, or by current grade in the course, etc.
- So let's write an alternate version that does not preserve the order (and is hopefully faster!):

```
template <class T>
void erase_at_index_any_order(std::vector<T> &v, int i) {









}
```

- What is the Big O Notation of `erase_at_index_any_order`?
- What about the opposite of erase? Can we write a function to *insert* an element in the middle of a vector, while preserving the order of the rest of the data? What will the Big O Notation of your `insert_at_index` operation be?
- Wait... does STL `vector` provide `erase` and `insert` functions? Yes! Then why are we writing our own? How do we use the STL versions of these functions? First, we need to learn about iterators!

9.5 What's an Iterator?

- Definition: An iterator
  - identifies a container and a specific element stored in the container,
  - lets us examine (and change, except for const iterators) the value stored at that element of the container,
  - provides operations for moving (the iterators) between elements in the container,
  - restricts the available operations in ways that correspond to what the container can handle efficiently.
- As we will see, iterators for different STL container classes have many operations in common. This can make it easy to modify a program that uses one container to use a different container if we change our program design or purpose.
- Iterators in many ways are generalizations of pointers: many operators / operations defined for pointers are defined for iterators. The C++ syntax for iterators is intentionally similar to the syntax for pointers. However, iterators and pointers are NOT the same thing for all STL containers.

9.6 Iterator Types, Variable Declarations, and Basic Operations

- Iterator types are declared by the container class. For example,

```
std::vector<std::string> enrolled;
enrolled.push_back("Sally");
enrolled.push_back("Bob");
enrolled.push_back("Alyssa");
std::vector<std::string>::iterator p;
std::vector<std::string>::const_iterator q;
```

  defines two (uninitialized) iterator variables, `p` and `q`.

- So how do we initialize an iterator with an interesting value? We can use the `begin()` member function to get an iterator that is attached to (sometimes we say "points to") the first item in the `vector` container. Not quite symmetrically the `end()` member function also gives us an iterator – but it is NOT attached the the last item, but instead it is *just beyond or after* the last item in the container.

```
p = enrolled.begin();
q = enrolled.end();
```

- We can change the container that a specific iterator is attached to *as long as the types match*:

```
std::vector<double> some_data(10, 3.14);
std::vector<double> more_data(20, 6.02);
std::vector<double>::iterator v_itr = some_data.begin();
v_itr = more_data.begin();  // this is ok!
v_itr = enrolled.begin();   // type mismatch, compilation error!
std::string s = "tiger";
std::string::iterator s_itr = s.begin();
s_itr = some_data.begin();  // type mismatch, compilation error!
```

  We cannot switch to a vector templated over a different type, or to a different type of container (e.g., STL `string` is actually a "container" holding `char` type):

- Once we have an iterator, we use the *dereference operator* access the value stored at an element of the container. This syntax is intentionally similar to pointers! We can use a dereferenced iterator *r-value*, e.g., to print the value:

```
std::cout << "before editing: " << *p << std::endl;
```

  Or we can use a dereferenced iterator as an *l-value* (as long as it is not a `const_iterator`). The example below changes the first entry in the `enrolled` vector from "Sally" to "Susan".

```
*p = "Susan";
```

- The dereference operator is combined with dot operator for accessing the member variables and member functions of elements stored in containers. Here's an example using the `Student` class and `students` vector from Lecture 4:

```
std::vector<Student>::iterator itr = students.begin();
(*itr).compute_averages(0.45);
```

  This operation would be illegal if `itr` had been defined as a `const_iterator` because `compute_averages` is a non-const member function. The parentheses on the `*i` are **required** (because of operator precedence).

  There is a "syntactic sugar" for the combination of the dereference operator and the dot operator, which intuitively looks like a little arrow, which is exactly equivalent:

```
itr->compute_averages(0.45);
```

- Just like pointers, iterators can be incremented and decremented using the `++` and `--` operators to move to the next or previous element of any container.

```
++itr;   itr++;   --itr;   itr--;
```

  These operations move the iterator to the next and previous locations in the vector, list, or string. The operations do not change the contents of container! Section 9.10 includes an example with the difference pre- & post- increment & decrement.

- Iterators can be compared using the `==` and `!=` operators. This is helpful when we want to write a loop over all the data in a container.

```
for (std::vector<std::string>::iterator itr = enrolled.begin(); itr != enrolled.end(); itr++) {
  std:cout << *itr << " is enrolled in the course." << std::endl; }
```

  NOTE: Remember that `.end()` *is NOT the last element in the container, but the slot AFTER the last element.*

9.7 STL's `erase` function takes in an iterator argument

- Here's an example showing how to use STL's `erase` member function:

```
std::vector<std::string>::iterator p = enrolled.begin();
++p;
std::vector<std::string>::iterator q = enrolled.erase(p);
```

- After the code above is executed:
  - The string value stored in the second entry of the vector has been removed.
  - The size of the vector has shrunk by one.
  - The return value of erase, which we stored in the iterator `q`, refers to the item that was the third entry and is now the second. (And any other data in the vector after the third location has also been shifted one spot to the left.)

- It is common to reuse the iterator `p` for the return value:

```
std::vector<std::string>::iterator p = enrolled.begin();
++p;
p = enrolled.erase(p);
```

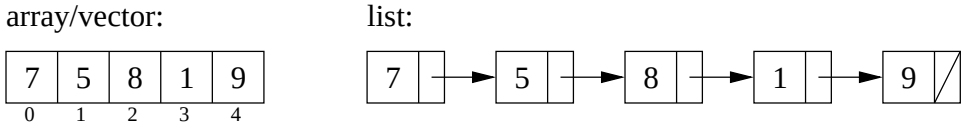
- Unfortunately, the built-in `erase` function for STL `vector` is $O(n)$. It is just as expensive as the version we wrote earlier because it shifts all the data to preserve the sequence.

9.8 The `list` Standard Library Container Class

- When items are continually being inserted and removed from the middle of a *sequence* of values (where the order must be preserved), vectors are not a good choice for the container.

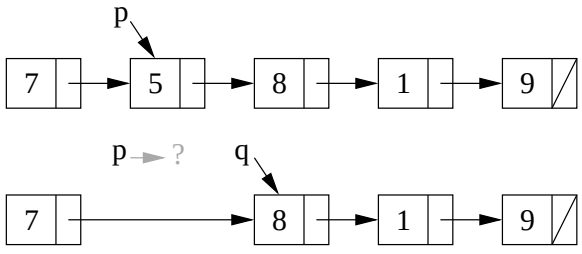
- Instead, let's learn about lists, our second STL container class!

- Both lists & vectors store *sequential data* that can shrink or grow dynamically. However, the use of memory is fundamentally different. Vectors are formed as a single contiguous array-like block of memory. Lists are formed as a sequentially linked structure instead.



- The syntax for erasing from an STL `list` is the same as for `vector`:

```
std::list<std::string>::iterator p = enrolled.begin();
++p;
std::list<int>::iterator q = s.erase(p);
```



- NOTE: The iterator `p` passed into the the STL `list::erase` function is now `INVALID`. You should not attempt to dereference and read or write this iterator – that would be a memory error.

- The necessary edits to the memory to preserve the order of data stored in a list are more localized. We don't need to shift all the data, we can just *bypass* the removed element. Therefore, the `list` version of `erase` has Big O Notation $O(1)$.

9.9 Insert

- STL provides an `insert` function for both STL `list` and `vector` that takes an iterator and a value and adds a link in the chain with the new value immediately before the item pointed to by the iterator.
- The call returns an iterator that points to the newly added element. Other variants of the insert function are also available in the full STL specification.
- Insert is $O(1)$ for `list` and $O(n)$ for `vector`.
- Thus, both `insert` and `erase` should be *AVOIDED or USED SPARINGLY* with large `vectors`.

9.10 Common Confusions / Mistakes with STL Iterators

NOTE: The example syntax below is the same for STL vector and STL lists.

```
std::vector<int> data;
std::vector<int>::iterator itr,itr2,itr3;
//std::list<int> data;
//std::list<int>::iterator itr,itr2,itr3;

data.push_back(100);  data.push_back(200);
data.push_back(300);  data.push_back(400);  data.push_back(500);

itr = data.begin(); // itr is pointing at the 100
++itr;              // itr is now pointing at 200
*itr += 1;          // 200 becomes 201

// itr += 1;      // NOTE: this syntax only works for vector/vector iterator
                  //       but it does not compile for list/list iterator
                  //       list iterators cannot be advanced like this

itr = data.end(); // itr is pointing "one past the last legal value" of data
itr--;            // itr is now pointing at 500;
itr2 = itr--;     // itr is now pointing at 400, itr2 is still pointing at 500
itr3 = --itr;     // itr is now pointing at 300, itr3 is also pointing at 300

// dangerous: decrementing the begin iterator is "undefined behavior"
//  (similarly, incrementing the end iterator is also undefined)
//  it may seem to work, but break later on this machine or on another machine!
itr = data.begin();
itr--;   // dangerous!
itr++;
assert (*itr == 100);  // might seem ok...  but rewrite the code to avoid this!
```

9.11 More STL `vector` vs. STL `list`: What's the same? What's different?

- Although the interface (public member functions) of lists and vectors and their iterators are quite similar, their implementations are VERY different. Clues to these differences can be seen in the operations that are NOT in common, such as:
- STL `vectors` (& C-style arrays) allow indexing / subscripting (`[ ]`), a.k.a. "random-access". That is, we can immediately jump to an arbitrary location within the vector / array. STL `lists` have no subscripting operation (we can't use `[]` to access data in a `list`). The only way to get to the middle of a list is start at the beginning (or end) of the list and follow pointers one link at a time.
- Random access also means that we can add (or subtract) an integer to a `vector` iterator, which will jump the iterator immediately, in $O(1)$ constant time, to the specified location. We can do this jump because the slots of a vector are guaranteed to be contiguous/adjacent and the compiler can perform simple arithmetic on the memory addresses to calculate the relative position of any other slot in the same vector. For example:

```
std::vector<double> even_more_data(1000,3.14);
std::vector<double>::iterator v_itr = even_more_data.begin() + 42;
*v_iter = 6.02;       // same result as typing:  even_more_data[42] = 6.02;
v_itr = v_itr + 100;  // v_itr is now 'pointing' to:  even_more_data[142]
```

We cannot perform these integer jumps with a list iterator, because the memory allocations for each data value in a list are separate and are not guaranteed to contiguous or adjacent.

- Pointer arithmetic also allows us to compare `vector` iterators using `<`, `>`, `<=`, `>=`. These comparisons are not available for `list` iterators – which can only be compared with `=` and `!=`.
- STL `list`s have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions available for both vectors and lists.
- Both containers can be sorted efficiently! STL provides built-in $O(n\ log\ n)$ sorting for both containers. However the syntax to sort the containers is different:

```
std::vector<int> my_vec;
std::list<int> my_lst;
// ... put some data in my_vec & my_lst
std::sort(my_vec.begin(),my_vec.end(),optional_compare_function);
my_lst.sort(optional_compare_function);
```

  We can provide an optional compare function for our data whether we use an STL `vector` or an STL `list`.

- Several operations invalidate the values of vector iterators, but not list iterators:
  - `erase` invalidates all iterators after the point of erasure in vectors;
  - `push_back` and `resize` invalidate ALL iterators in a vector

  The value of any associated vector iterator must be re-assigned / re-initialized after these operations.

9.12 Implementing `Vec<T>` Iterators

- So, how do we add iterators to our `Vec<T>` class declaration from Lecture 7?

```
public:
  // TYPEDEFS
  typedef T* iterator;
  typedef const T* const_iterator;

  // MODIFIERS
  iterator erase(iterator p);
  iterator insert(iterator p, const T &element);

  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
```

- First, remember that `typedef` statements create custom, alternate names for existing types.

  `Vec<int>::iterator` is an iterator type defined by the `Vec<int>` class. It is just a `T *` (an `int *`). Thus, internal to the declarations and member functions, `T*` and `iterator` **may be used interchangeably**.

- Because the underlying implementation of `Vec` uses an array, and because pointers *are* the "iterator"s of arrays, the implementation of vector iterators is quite simple. *Note: the implementation of iterators for other STL containers is more involved!*
- Thus, `begin()` returns a pointer to the first slot in the `m_data` array. And `end()` returns a pointer to the "slot" just beyond the last legal element in the `m_data` array (as prescribed in the STL standard).
- Furthermore, dereferencing a `Vec<T>::iterator` (dereferencing a pointer to type `T`) correctly returns one of the objects in the `m_data`, an object with type `T`.
- And similarly, the `++`, `--`, `<`, `==`, `!=`, `>=`, etc. operators on pointers automatically apply to `Vec` iterators.
- The `erase` and `insert` functions are multi-line functions, require a loop over all data after the specified edit position. `insert` may require the allocation be resized (if `m_alloc == m_size`).
- Finally, note that after a `push_back` or `erase` or `resize` call some or all iterators referring to elements in that vector may be *invalidated*. Why? You must take care when designing your program logic to avoid invalid iterator bugs!

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <list>
#include <cassert>


// Enroll a student if there is room and the student is not already in course or on waiting list.
void enroll_student(const std::string& id, unsigned int max_students,
                    std::list<std::string>& enrolled, std::list<std::string>& waiting) {
  // Check to see if the student is already enrolled.
  std::list<std::string>::iterator itr;
  for (itr = enrolled.begin(); itr != enrolled.end(); ++itr) {
    if (*itr == id) {
      std::cout << "Student " << id << " is already enrolled." << std::endl;
      return;
    }
  }
  // If the course isn't full, add the student.
  if (enrolled.size() < max_students) {
    enrolled.push_back(id);
    std::cout << "Student " << id << " added.\n"
         << enrolled.size() << " students are now in the course." << std::endl;
    return;
  }
  // Check to see if the student is already on the waiting list.
  for (itr = waiting.begin(); itr != waiting.end(); ++itr) {
    if (*itr == id) {
      std::cout << "Student " << id << " is already on the waiting list." << std::endl;
      return;
    }
  }
  // If not, add the student to the waiting list.
  waiting.push_back(id);
  std::cout << "The course is full.  Student " << id << " has been added to the waiting list.\n"
       << waiting.size() << " students are on the waiting list." << std::endl;
}


// Remove a student from the course or from the waiting list.  If removing the student from the
// course opens up a slot, then the first person on the waiting list is placed in the course.
void remove_student(const std::string& id,
                    std::list<std::string>& enrolled, std::list<std::string>& waiting) {
  // Check to see if the student is on the course list.
  bool found = false;
  std::list<std::string>::iterator itr = enrolled.begin();
  while (!found && itr != enrolled.end()) {
    found = *itr == id;
    if (!found) ++itr;
  }
  if (found) {
    // Remove the student and see if a student can be taken from the waiting list.
    enrolled.erase(itr);
    std::cout << "Student " << id << " removed from the course." << std::endl;
    if (waiting.size() > 0) {
      enrolled.push_back(waiting.front());
      std::cout << "Student " << waiting.front()
                << " added to the course from the waiting list." << std::endl;
      waiting.pop_front();
      std::cout << waiting.size() << " students remain on the waiting list." << std::endl;
    } else {
      std::cout << enrolled.size() << " students are now in the course." << std::endl;
    }
  } else {
    // Check to see if the student is on the waiting list
    found = false;
    itr = waiting.begin();
    while (!found && itr != waiting.end()) {
      found = *itr == id;
      if (!found) ++itr;
    }
    if (found) {
      waiting.erase(itr);
      std::cout << "Student " << id << " removed from the waiting list.\n"
           << waiting.size() << " students remain on the waiting list." << std::endl;
    } else {
      std::cout << "Student " << id
                << " is in neither the course nor the waiting list" << std::endl;
    }
  }
}

int main() {
  // Read in the maximum number of students in the course
  unsigned int max_students;
  std::cout << "\nEnrollment program for CSCI 1200\nEnter the maximum number of students allowed\n";
  std::cin >> max_students;

  // Initialize the vectors
  std::list<std::string> enrolled;
  std::list<std::string> waiting;

  // Invariant:
  // (1) enrolled contains the students already in the course,
  // (2) waiting contains students who will be admitted (in the order of request) if a spot opens up
  // (3) enrolled.size() <= max_students,
  // (4) if the course is not filled (enrolled.size() != max_students) then waiting is empty
  do {
    // check (part of) the invariant
    assert (enrolled.size() <= max_students);
    assert (enrolled.size() == max_students || waiting.size() == 0);
    std::cout << "\nOptions:\n"
         << "  To enroll a student type 0\n"
         << "  To remove a student type 1\n"
         << "  To end type 2\n"
         << "Type option ==> ";
    int option;
    if (!(std::cin >> option)) { // if we can't read the input integer, then just fail.
      std::cout << "Illegal input.  Good-bye.\n";
      return 1;
    } else if (option == 2) {
      break; // quit by breaking out of the loop.
    } else if (option != 0 && option != 1) {
      std::cout << "Invalid option.  Try again.\n";
    } else { // option is 0 or 1
      std::string id;
      std::cout << "Enter student id: ";
      if (!(std::cin >> id)) {
        std::cout << "Illegal input.  Good-bye.\n";
        return 1;
      } else if (option == 0) {
        enroll_student(id, max_students, enrolled, waiting);
      } else {
        remove_student(id, enrolled, waiting);
      }
    }
  }
  while (true);

  // some nice output
  enrolled.sort();
  std::cout << "\nAt the end of the enrollment period, the following students are in the class:\n\n";
  std::list<std::string>::iterator itr;
  for (itr = enrolled.begin(); itr != enrolled.end(); ++itr) { std::cout << *itr << std::endl;  }
  if (!waiting.empty()) {
    std::cout << "\nStudents are on the waiting list in the following order:\n";
    for (itr = waiting.begin(); itr != waiting.end(); ++itr) { std::cout << *itr << std::endl; }
  }
  return 0;
}
```