CSCI-1200 Data Structures — Fall 2024
Lecture 8 — Iterators & STL Lists

Review from Lecture 7

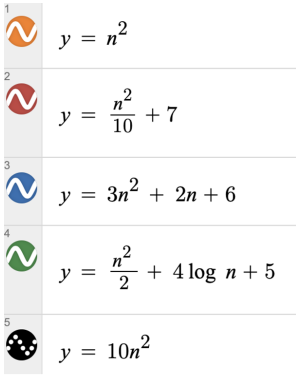
- Algorithm Analysis, Formal Definition of Big O Notation
- Simple recursion, Visualization of recursion, Iteration vs. Recursion, "Rules" for writing recursive functions.
- Examples!

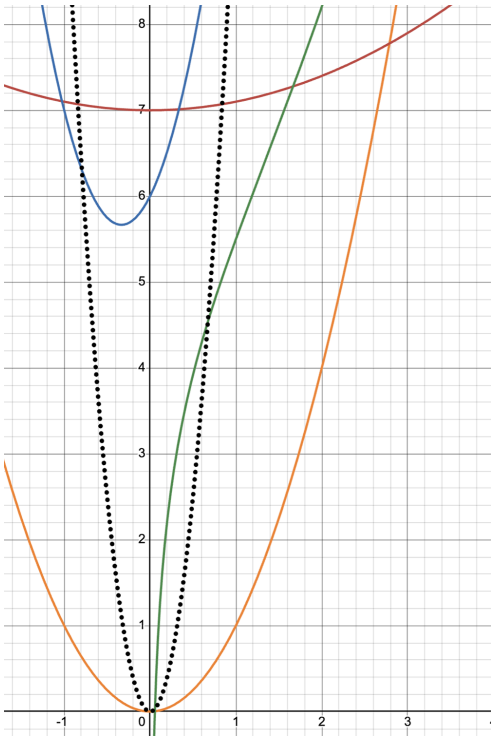
Today

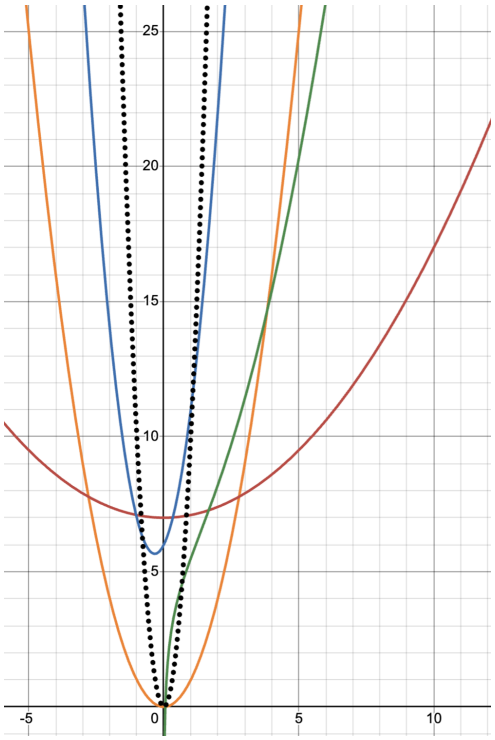
- Constructors, Assignment Operator, and Destructor
- Big 'O' Notation for `push_back`
- Another `vector` operation: `pop_back`
- *Erasing items* from vectors is inefficient!
- Iterators and iterator operations
- Preview: STL `lists` are a different sequential container class.

8.1 Review: Formal Definition of Big O Notation – An Example in Pictures!

- Definition: Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- Crossing out the non-dominant terms and the constant coefficient in front of the dominant term, the red, blue, and green functions (left image below) are all simplified to be part of the same Big O Notation $= O(n^2)$.
- Plotting these functions together (middle image below) it is not clear how they are related to or bounded by the orange function $y = n^2$.





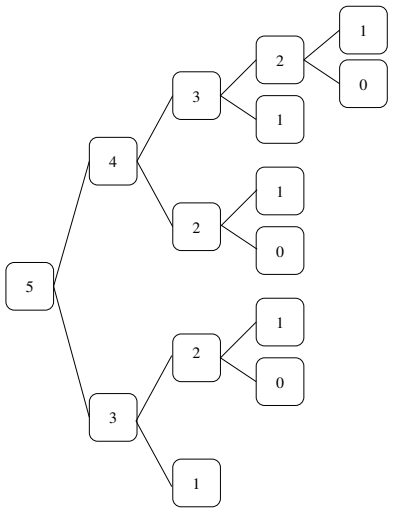


- For this specific group of functions, it is sufficient to set k to 10, and zoom out a little bit (right image above). We can see that the dotted black line bounds the other functions from above for $n > n_0 = 2$.
- And for any other specific function that belongs in $O(n^2)$, we can find a k and n_0 such that $k * n^2$ bounds that function from above for values $n > n_0$. Just increase k and/or n_0 and zoom out as needed.

8.2 Review: Fibonacci Optimization: Big O Notation of Time vs. Space

- Mathematical definition:

$$\text{Fib}_n = \begin{cases} 1 & n == 0 \\ 1 & n == 1 \\ \text{Fib}_{n-1} + \text{Fib}_{n-2} & n > 1 \end{cases}$$



- The *simple* recursive version of Fibonacci that is a direct translation of the mathematical definition of the Fibonacci sequence:

```
int fib ( int n ) {
  assert ( n >= 0 );
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fib (n-1) + fib (n-2);
  }
}
```

- A revised algorithm that is *iterative*, and uses an STL vector:
  NOTE: Often when re-writing a recursive function to be iterative, we have to use extra memory to explicitly store intermediate values.

```
int ifib_vec ( int n ) {
  assert ( n >= 0 );
  std::vector<int> v;
  v.push_back(1);
  v.push_back(1);
  for (int i = 2; i <= n; i++) {
    v.push_back(v[i-1] + v[i-2]);
  }
  return v[n];
}
```

  NOTE: Usually when re-writing a recursive function to be iterative, they both have the same Big O Notation!

- A further revision of the *iterative* algorithm that doesn't use an STL vector:

```
int ifib_novec ( int n ) {
  assert ( n >= 0 );
  if (n < 2) return 1;
  int prevprev = 1;
  int prev = 1;
  for (int i = 2; i < n; i++) {
    int tmp = prevprev + prev;
    prevprev = prev;
    prev = tmp;
  }
  return prevprev + prev;
}
```

  NOTE: Not every algorithm can be revised to remove all non-constant memory usage!

- What is the Big O Notation for the *running time* for each version?
- What is the Big O Notation for the *memory usage* for each version *ignoring* the memory used to store the recursive function calls on the stack?
- What is the Big O Notation for the *memory usage* for each version *including* the memory used to store the recursion function calls on the stack?

8.3 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

8.4 Exercise: Big 'O' Notation Analysis of STL vector / Vec `push_back`

- What is the Big O Notation of `Vec` and STL Vector `push_back`?

```
// Add an element to the end, resize if necesssary.
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    // Allocate a larger array, and copy the old values
    // Calculate the new allocation.  Make sure it is at least one.
    m_alloc *= 2;
    if (m_alloc < 1) m_alloc = 1;
    // Allocate and copy the old array
    T* new_data = new T[ m_alloc ];
    for (size_type i=0; i<m_size; ++i)
      new_data[i] = m_data[i];
    // Delete the old array and reset the pointers
    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;
}
```

8.5 Another STL `vector` operation: `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:** What is the Big O Notation of `pop_back`?

8.6 Review: Erasing Items from a `Vec` or STL `vector`

- In Lab 5 we wrote:

```
template <class T>  void Vec<T>::erase_at_index(int n) {
  assert (n >= 0 && n < int(m_size));
  for (unsigned int i = n; i < m_size-1; i++) {
    m_data[i] = m_data[i+1];
  }
  m_size--;
}
```

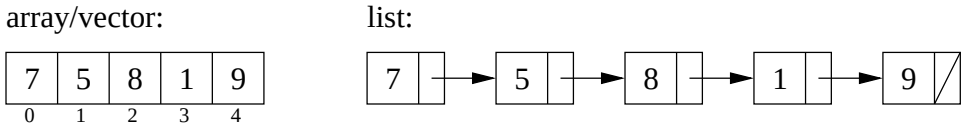
- If we are working with STL `vector`, we would have to write this as a non-member function (since we can't directly add member functions to STL containers).

```
void erase_from_vector(unsigned int n,  vector<string>& v) {
  assert (n >= 0 && i < v.size());
  for (int i = n; i < v.size()-1; i++) {
    v[i] = v[i+1];
  }
  v.pop_back();
}
```

- **Exercise** What is the Big O Notation for `erase_at_index` / `erase_from_vector`?

8.7 What To Do About the Expense of Erasing From a Vector?

- When items are continually being inserted and removed, vectors are not a good choice for the container.
- Instead we need a different sequential container, called a *list*.
  - This has a "linked" structure that makes the cost of erasing independent of the size.



- Switching from a vector to a list may improve the performance of programs that make significant use of insert and erase in the middle of sequences.
- . . . but before we do that, we need to learn about iterators. Iterators are standardized way to interact with the data elements stored in all of the different STL containers.

8.8 Iterators

- 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 container classes have many operations in common. This often makes the switch between containers fairly straightforward from the programer's viewpoint.
- Iterators in many ways are generalizations of pointers: many operators / operations defined for pointers are defined for iterators. You should use this to guide your beginning understanding and use of iterators.

8.9 Iterator Declarations and Operations

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

```
vector<string>::iterator p;
vector<string>::const_iterator q;
```

  defines two (uninitialized) iterator variables.

- The *dereference operator* is used to access the value stored at an element of the container. The code:

```
p = enrolled.begin();
*p = "012312";
```

  changes the first entry in the `enrolled` vector.

- 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 3:

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

  Notes:
  - This operation would be illegal if `i` 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 is exactly equivalent:

```
vector<StudentRec>::iterator i = students.begin();
i->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.
- Iterators can be compared using the `==` and `!=` operators.
- Iterators can be assigned, just like any other variable.
- Vector iterators have several additional operations:
  - Integer values may be added to them or subtracted from them. This leads to statements like

```
enrolled.erase(enrolled.begin() + 5);
```

  - Vector iterators may be compared using operators like `<`, `<=`, etc.
  - For most containers (other than vectors), these "random access" iterator operations are not legal and therefore prevented by the compiler. The reasons will become clear as we look at their implementations.

8.10 Motivating Example Program: Course Enrollment and Waiting List

- This program (last page of the handout) maintains the class list and the waiting list for a single course. The program is structured to handle interactive input. Error checking ensures that the input is valid.
- Vectors store the enrolled students and the waiting students. The main work is done in the two functions `enroll_student` and `remove_student`.
- The invariant on the loop in the main function determines how these functions must behave.

8.11 Exercise: Revising the Class List Program to Use Iterators

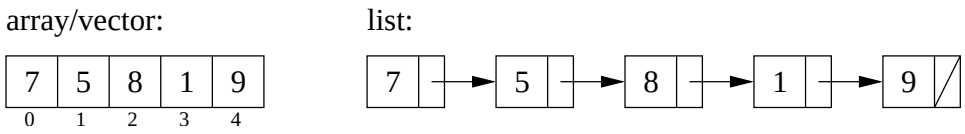
- Now let's modify the class list program to use iterators.
  We want to remove all usage of the STL `vector` subscript operator, `[ ]` .
- This includes rewriting `erase_from_vector` to use iterators:

```
void erase_from_vector(vector<string>::iterator itr,  vector<string>& v) {




}
```

- *Note:* The STL `vector` class has a member function that does just this... called `erase`! More on that soon...
  - Give a sample call to our homemade `erase_from_vector` function:
  - Give the equivalent statement that instead uses the `erase` member function:

8.12 The `list` Standard Library Container Class

- Lists are our second standard-library container class. (Vectors were the first.) Both lists & vectors store sequential data that can shrink or grow.
- 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.



- 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 `vector`s / arrays allow "random-access" / indexing / `[]` subscripting. We can immediately jump to an arbitrary location within the vector / array.
  - STL `list`s have no subscripting operation (we can't use `[]` to access data). The only way to get to the middle of a list is to follow pointers one link at a time.
  - Lists have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions availble for both vectors and lists.
  - `erase` and `insert` in the middle of the STL `list` is very efficient, independent of the size of the list. Both are implemented by rearranging pointers between the small blocks of memeory. (We'll see this when we discuss the implementation details of STL `list` next week!)
  - We can't use the same STL `sort` function we used for `vector`; we must use a special `sort` function defined by the STL `list` type.

```
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);
```

    Note: STL `list sort` member function is just as efficient, $O(n \log n)$, and will also take the same optional compare function as STL `vector`.
  - 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.

8.13 Exercise: Revising the Class List Program to Use Lists (& Iterators)

Now let's further modify the program to use lists instead of vectors. Because we've already switched to iterators, this change will be relatively easy. And now the program will be more efficient!

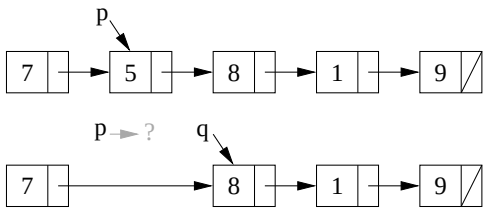
8.14 STL list (and STL vector) has an `erase` member function

- STL `list`s and `vector`s each have a special member function called `erase`. In particular, given list of ints `s`, consider the example:

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

- After the code above is executed:
  - The integer stored in the second entry of the list has been removed.
  - The size of the list has shrunk by one.
  - The iterator `p` does not refer to a valid entry.

– The iterator `q` refers to the item that was the third entry and is now the second.



- To reuse the iterator `p` and make it a valid entry, you will often see the code written:

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

- Even though the `erase` function has the same syntax for vectors and for list, the vector version is $O(n)$, whereas the list version is $O(1)$.

8.15 Insert

- Similarly, there is an `insert` function for STL lists 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. Variants on the basic insert function are also defined.

8.16 Exercise: Using STL `list` Erase & Insert

Write a function that takes an STL `list` of integers, `lst`, and an integer, `x`. The function should 1) remove all negative numbers from the list, 2) verify that the remaining elements in the list are sorted in increasing order, and 3) insert `x` into the list such that the order is maintained.

8.17 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!
```

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <assert.h>
using namespace std;

void erase_from_vector(unsigned int i, vector<string>& v) {
  /* EXERCISE: IMPLEMENT THIS FUNCTION */
}

// Enroll a student if there is room and the student is not already in course or on waiting list.
void enroll_student(const string& id, unsigned int max_students,
                    vector<string>& enrolled, vector<string>& waiting) {
  // Check to see if the student is already enrolled.
  unsigned int i;
  for (i=0; i < enrolled.size(); ++i) {
    if (enrolled[i] == id) {
      cout << "Student " << id << " is already enrolled." << endl;
      return;
    }
  }
  // If the course isn't full, add the student.
  if (enrolled.size() < max_students) {
    enrolled.push_back(id);
    cout << "Student " << id << " added.\n"
         << enrolled.size() << " students are now in the course." << endl;
    return;
  }
  // Check to see if the student is already on the waiting list.
  for (i=0; i < waiting.size(); ++i) {
    if (waiting[i] == id) {
      cout << "Student " << id << " is already on the waiting list." << endl;
      return;
    }
  }
  // If not, add the student to the waiting list.
  waiting.push_back(id);
  cout << "The course is full.  Student " << id << " has been added to the waiting list.\n"
       << waiting.size() << " students are on the waiting list." << 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 string& id, unsigned int max_students,
                    vector<string>& enrolled, vector<string>& waiting) {
  // Check to see if the student is on the course list.
  bool found = false;
  unsigned int loc=0;
  while (!found && loc < enrolled.size()) {
    found = enrolled[loc] == id;
    if (!found) ++loc;
  }
  if (found) {
    // Remove the student and see if a student can be taken from the waiting list.
    erase_from_vector(loc, enrolled);
    cout << "Student " << id << " removed from the course." << endl;
    if (waiting.size() > 0) {
      enrolled.push_back(waiting[0]);
      cout << "Student " << waiting[0] << " added to the course from the waiting list." << endl;
      erase_from_vector(0, waiting);
      cout << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << enrolled.size() << " students are now in the course." << endl;
    }
  } else {
    // Check to see if the student is on the waiting list
    found = false;
    loc = 0;
    while (!found && loc < waiting.size()) {
      found = waiting[loc] == id;
      if (!found) ++loc;
    }
    if (found) {
      erase_from_vector(loc, waiting);
      cout << "Student " << id << " removed from the waiting list.\n"
           << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << "Student " << id << " is in neither the course nor the waiting list" << endl;
    }
  }
}

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

  // Initialize the vectors
  vector<string> enrolled;
  vector<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);
    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 (!(cin >> option)) { // if we can't read the input integer, then just fail.
      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) {
      cout << "Invalid option.  Try again.\n";
    } else { // option is 0 or 1
      string id;
      cout << "Enter student id: ";
      if (!(cin >> id)) {
        cout << "Illegal input.  Good-bye.\n";
        return 1;
      } else if (option == 0) {
        enroll_student(id, max_students, enrolled, waiting);
      } else {
        remove_student(id, max_students, enrolled, waiting);
      }
    }
  }
  while (true);

  // some nice output
  sort(enrolled.begin(), enrolled.end());
  cout << "\nAt the end of the enrollment period, the following students are in the class:\n\n";
  for (unsigned int i=0; i<enrolled.size(); ++i) { cout << enrolled[i] << endl;  }
  if (!waiting.empty()) {
    cout << "\nStudents are on the waiting list in the following order:\n";
    for (unsigned int j=0; j<waiting.size(); ++j) { cout << waiting[j] << endl; }
  }
  return 0;
}
```