Day 27 (Fri 04/01)

- exercise 26 review
- day 26 recap questions
- exercise 27

Announcements/reminders

- Submit midterm project survey by Friday 4/8
  - On Gradescope
  - Do **NOT** expect that late submissions will be accepted

- HW5: due Wednesday 4/6 by 11pm

Exercise 26

Part 2: make_cumulative

```
void make_cumulative(std::vector<double> &dist) {
  for (std::vector<double>::iterator i = dist.begin() + 1;
       i != dist.end();
       i++) {
    *i += *(i - 1);
  }
}
```

Note: vectors have random-access iterators, so "pointer arithmetic" with iterators is possible

Fun fact: in most STL implementations, vector iterators actually ARE pointers

Exercise 26

Part 4: naive_find_last_iterator

```
  std::vector<double>::const_iterator best = begin, i = begin;
  while (i != end && *i <= v) {
    best = i;
    i++;
  }
  return best;
```

Exercise 26

fast_find_last_iterator

```
  size_t n = end - begin;
  if (n == 1) {
    return begin;
  }

  std::vector<double>::const_iterator mid = begin + n/2;
  if (*mid > v) { return fast_find_last_iterator(begin, mid, v); }
          else { return fast_find_last_iterator(mid, end, v); }
```

Note: this is *slightly* different than the standard recursive binary search, because we do not eliminate the middle element from consideration in the recursive case

Day 27 recap questions

1. What is object-oriented programming?
2. What is the difference between a public and a private field/member function?
3. Do class fields and member functions default to public or private?
4. Can we define member functions in a struct in C? How does C++ handle structs? Can we do that in C++?
5. What is a default constructor?
6. Why is using an initializer list in a class constructor a better choice than not using one?

1. What is object-oriented programming?

An object is an instance of a class (or struct) type. The class or struct type can have *member variables* and *member functions*.

Member variables (a.k.a. fields): define the data contained in an object.

Member functions (a.k.a. methods): define the *behaviors* of an object.

In an object-oriented program, computations are done by calling member functions on objects.

2. What is the difference between a public and a private field/member function?

public member: can be directly accessed from code outside the member functions of the class.

private member: can only be directly accessed from member functions of the class.

General rules:

- member variables (fields) should be private
- member functions to be used by the program as a whole should be public (these are sometimes called "API functions")
- member functions that are only needed internally (helper functions for the class) should be private

The idea that internal implementation details (fields, helper functions) are private is known as *encapsulation*.

3. Do class fields and member functions default to public or private?

For class types: default is private

For struct types: default is public

Otherwise, there is no difference between class types and struct types!

4. Can we define member functions in a struct in C? How does C++ handle structs? Can we do that in C++?

In C, a struct type cannot have member functions.

In C++, a struct can have member functions.

The only difference between C++ struct types and class types is that the members of struct types are public by default, and the members of class types are private by default.

5. What is a default constructor?

A default constructor is a constructor which requires no arguments. It is invoked to initialize an object if no other constructor is invoked. Like all constructors, its main job is to initialize the fields of fields of the object being created.

Assume that Foo is a class.

Foo f; // default constructor called to initialize fields of f

Foo arr[10]; // default constructor called on each element of arr

Foo *p = new Foo[5]; // default constructor called on each element of
// dynamically-allocated array

6. Why is using an initializer list in a class constructor a better choice than not using one?

It avoids an initialization of a field by its default constructor, followed by an *assignment* to change the value of the field.

E.g.:

```
struct Foo {
  std::string s;
  Foo();
};

// best way to initialize s
Foo::Foo()
  : s("initial value for s")
{ }

// not as good
Foo::Foo(const std::string &val) {
  s = "initial value for s";
}
```