Individual Testing, Big-O, C++

Bryce Boe

2013/07/16 ☺

CS24, Summer 2013 C

Outline

- IRB Consent Forms
- Project 1 Questions
- Individually Testing ADTs
- Big-O Examples
- C++ Introduction

PROJECT 1 QUESTIONS?

INDIVIDUAL TESTING

I expect that you

- are able write your own C program from scratch and compile the program
- are able to include a library and use its functions and compile the library
- can think of and write test cases for those functions

Review: The simplest C Program

```
int main() {
    return 0;
}
```

Save as: simple.c (or something else)

Compile via: clang simple.c

Review: A simple program to utilize the List ADT

```
#include "array_list.h"
int main() {
        struct List *tmp = list_construct();
        // Insert testing code here (adding/removing items)
        list_destruct(list);
        return 0;
}
```

Save as: my_test_list.c

Compile as: clang my_test_list.c array_list.c

Testing the list ADT

- <In class creation of my_test.c>

BIG-O REVIEW AND EXAMPLES

Recall: Common Ordered Complexities

- $O(1)$ – constant time
- $O(\log(n))$ – logarithmic time
- $O(n)$ – linear time
- $O(n\log(n))$ – linearithmic time
- $O(n^2)$ – quadratic time
- $O(2^n)$ – exponential time
- $O(n!)$ – factorial time

O(?)

```
int a[1024];  // assume allocated

if (a[0] == a[1])
        return a[2];
else
        return a[0];
```

O(1)

O(?)

```
int a[4];  // assume allocated
int n = sizeof(a) / sizeof(int);
int sum = 0;
for (int i = 0; i < n; ++i)
        sum += a[i];
return sum;
```

O(n)

O(?)

```
int a[4];  // assume allocated

return a[0] + a[1] + a[2] + a[3];
```

O(1)

O(?)

```
int a[1024];  // assume allocated
int n = sizeof(a) / sizeof(int);
int dups = 0;
for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
                if (a[i] == a[j])
                        ++dups;
```

$O(n^2)$

O(?)

```
int a[1024];  // assume allocated
int n = sizeof(a) / sizeof(int);
int dups = 0;
for (int i = 0; i < n; ++i)
        for (int j = i; j < n; ++j)
                if (a[i] == a[j])
                        ++dups;
```

$O(n^2)$

C++ INTRODUCTION

Why C++?

- Problems with C
  - Has a single global namespace
  - Cannot use the same name for functions with different types (e.g., min(int, int) and min(double, double)) – called *overloading*
  - Difficult to minimize source-code repetition for similar functions with different types

Some Differences

- #include <stdio.h> → #include <iostream>
  - Or if you want fprintf, etc use #include <cstdio>
- printf("Hello\n"); → cout << "Hello\n";
- Rather than defining a **struct** which only contains data, define a **class** which contains data and methods on the data
- **throw** exceptions rather than use return values to represent error cases

Classes

- Provide encapsulation
  - Combining a number of items, such as variables and functions, into a single package, such as an object of some class (or instance of the class)

Scope Resolution Operator

- ClassName::method_name
- Used to identify the scope, class in this case, that the method belongs to as there may be more than 1 instance of method_name
- Scope resolution isn’t necessary if you are also a member of that class

Data Hiding

- Declaring member (instance) variables as private, why?
  - Assists in separation of implementation and interface
  - Allows for input validation and state consistency

Declaring Private attributes

```
class Date {
    int day;        // this section is private by default
    int month;  // though you should be explicit
public:
    void output_date();
private:
    int year;
};
```

Accessor methods

- Sometimes called getters
- Instance methods that return some data to indicate the state of the instance
- Typically prefixed with get_

```
int Date::get_day() { return day; }
```

Mutator methods

- Sometimes called setters
- Instance methods that update or modify the state of the instance
- Typically prefixed with set_

```
void Date::set_day(int d) { day = d; }
```

Overloading Instance Methods

- Defining methods of a class with the same name, but different parameters

```
void Date::update_date(int d, int m, int y) {…}
void Date::update_date(Date &other) {…}
```

Class Constructors

- A constructor is used to initialize an object
- It must:
  - Have the same name as the class
  - Not return a value
- Constructors should be declared public
  - To ponder: what does it mean to have a non-public constructor?
- Always define a default constructor

Example

```
class Date {
  public:
    Date(int d, int m, int y);
    Date();  // default constructor
  private:
    int day, month, year;
};
```

Two ways to initialize variables

- From the constructor declaration (implementation)
- Method 1: Initialize in the constructor initialization section

```
Date::Date() : day(0), month(0), year(0) {}
```

- Method 2: In the method body

```
Date::Date() {
   day = 0; month = 0; year = 0; }
```

Example Constructor Usage

Date a (10, 10, 11); // use the 3 param constructor

Date b; // correct use of default constructor

~~Date c();~~ // incorrect use of default constructor
// This is actually a function definition

Date d = Date(); // valid, but inefficient

Anonymous Instances

- An instance that is not bound to a variable

Date d = Date();

- In the above example there are actually two instances of class Date
  - The first is represented by d
  - The second is the anonymous instance represented by Date()
- The assignment operator is used to transfer information from the anonymous instance to d

Abstract Data Types

- A formal specification of the separation of implementation and interface
- Developer can use ADTs without concern for their implementation
- Using classes, you can define your own ADTs
  - This allows for reusable code

Tips for writing ADTs

- Make all the member variables private attributes of the class
- Provide a well defined public interface to the class and **don't** change it
- Make all helper functions private