COMP 2160 - Programming Practices

Course Description

Calendar entry

Introduction to issues involved in real-world computing. Topics will include memory management, debugging, compilation, performance, and good programming practices. Prerequisite: COMP 1020 (C+) or COMP 1021 (C+). Pre- or corequisite: COMP 2140.

General Course Description

By this point you know *how* to code (pick one or more of Python, Processing, Java), but you probably don't know how to code **well**.

In this course we're going to be looking at tools and methods that you can use to improve your coding skills, *regardless* of the language that you're writing in. We're going to be using C and Unix. This is *not* a C/Unix course. It is a course focusing on good programming practices that form the foundation you need to become a successful software developer.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Write code that makes use of instantiation, objects in memory, and classes.
- Write code that deals with large sets of data using files stored on disk.
- Design and implement iterative and recursive algorithms.
- Design and implement algorithms using arrays and basic linked lists.
- Implement simple searching and sorting algorithms.

Course Goals

By the end of this course students will:

- Write code using an unfamiliar programming language idiomatically.
- Write code that makes identifying and fixing problems easier.
- Verify and validate that code meets a set of well-defined expectations.
- Modularize code via well-defined functional units.
- Describe how memory is used and safely managed within code they write.
- Identify systemic performance issues and provide mitigating solutions.

Learning Outcomes

Design by Contract

Students should be able to:

1. Define the interface to an Abstract Data Type (ADT).
2. Define the implementation of an ADT using a private data structure.
3. Define the pre- and post-conditions for a routine.
4. Define the invariants that encapsulate the valid states of an ADT.
5. Use preconditions, postconditions, and invariants to validate the run-time behaviour of an ADT being used in an application.

Testing

Students should be able to:

1. Explain the purpose of testing code.
2. List classifications of test data (general, edge, leaks).
3. Create general case test data (inputs, expected outputs) for an ADT.
4. Create *edge case* test data (inputs, expected outputs) for an ADT.
5. Manually test an ADT with test data.
6. Explain the purpose of automated testing.
7. Implement a test harness that automates the testing of an ADT.

Programming Practices

Students should be able to:

1. Recognize potentially risky programming techniques and how they differ in different high-level languages.
2. Write "safe" code in a programming language that makes it difficult to write safe code.
3. Apply "good" programming techniques to produce readable and modifiable code in a programming language that makes it easy to write unreadable code.
4. Divide code from a complex project into higher-level modules, for separate development and compilation.
5. Describe the benefits of modularity and use simple metrics to specify the degree of modular independence, such as coupling and cohesion.
6. Given an existing solution, identify and explain where the use of appropriate data structures, algorithms, and/or techniques (such as caching and lookup tables) provide better paths to optimization than low-level code.

Memory and Pointers

Students should be able to:

1. Write, test, and debug programs in a high-level language that exposes low-level details of data types and memory addresses.
2. Describe concepts of in-application memory management such as first-fit memory allocation, the run-time stack and the heap, and garbage collection.
3. Implement a simple systems-level solution to one of the in-application memory management techniques.
4. Use function pointers to parameterize behaviours.

Tools

Students should be able to:

1. Build and execute programs from a command-line environment.
2. Write code that takes advantage of its environment through a) simple command-line options to define run-time behaviour, and b) the redirecting of standard input and output for file I/O.
3. Use an automated build tool, such as `make` along with a pre-defined `Makefile`, to build a complex project.
4. Define the build of their own project by modifying an existing `Makefile`.
5. Use a source-level debugger, such as `lldb`, to inspect program state and step through code line-by-line to determine the causes of errors in a program.