CS 240 Final Exam Review

Linux

- I/O redirection
- Pipelines
- Standard commands

C++ Pointers

- How to declare
- How to use
- Pointer arithmetic
- new, delete
- Memory leaks

C++ Parameter Passing

- modes
  - value
  - pointer
  - reference
- arrays
  - default: by pointer
- structs/objects
  - default: by value

C++ Classes

- how to write a complete class
  - Constructors
    - default
    - copy
    - others
  - Destructor
  - operator=
  - operator==
  - other operators
  - instance variables
  - public vs. private vs. protected

C++ Inheritance

- Purposes of inheritance:
  - code reuse
  - polymorphism
- Syntax used for inheritance
- Method binding
- Static vs. dynamic binding
- Ordering of method calls
  - constructor
  - destructor
  - operator=
  - sub class, super class call ordering

C++ Polymorphism

- what it is
  - subclass substitutable for superclass
- how it works
  - subclass method gets called
- how it is implemented
  - v-tables
    - what are v-tables
    - how do they get used
    - what overhead is associated with them

C++ Memory Management

- C++ memory model
  - stack, heap, static data
- what data goes on stack, heap, or static area?
- heap layout
- when to use stack vs. heap
- reference counting
- smart pointers
- Mark-and-Sweep garbage collection
- 2D arrays on the heap (new [][], spine-with-ribs, manual indexing)

C++ Multi-file projects

- Header files (.h)
- Implementation files (.cpp)
- Source tree
- Compiling/linking separate files
- Phases
  - preprocess
  - compile
  - assemble
  - link

C++ Libraries

- Static
  - Linker copies the code to the executable
  - Executable is stand-alone
  - `ar -rcs ../lib/libcs240utils.a *.o`
- Dynamic
  - Loader loads the needed library files at run-time
  - Executable is not stand-alone
  - `g++ -shared -o ../lib/libcs240utils.so *.o`
- Advantages and disadvantages?

C++ Miscellaneous

- Operator overloading
- Templates
- STL
  - Containers
  - Iterators

Software Design

- Managing complexity
  - divide-and-conquer
  - abstraction (create abstractions that model application domain)
  - information hiding (hide implementation details)
  - minimize dependencies (between parts of the system)

- Finding/creating abstractions
  - Classes are nouns, methods are verbs
  - Read the specification
  - Use cases

Software Design

- Coupling
  - Level of dependency between classes
- Cohesion
  - How closely related one class's or method's responsibilities are
- Layering
  - Used for organizing abstractions

Software Design

- Data structure selection
  - Data abstraction
  - Information hiding
  - The right data structure for the application
    - Efficiency
    - Access time
    - Complexity
    - Implementation, debugging costs

Software Design

- Class descriptions
  - What are the classes and their responsibilities?
  - What are the member variables and methods?

- Runtime interactions between objects
  - How do the various objects collaborate at runtime to achieve the program's function

Software Implementation

- Code a little, test a little, code a little, test a little, ...
  - "big bang" doesn't work
  - The role of "unit testing"

- Two main reasons for creating routines
  1. top-down decomposition of algorithms
  2. avoid code duplication

- Choose good names for
  - classes
  - methods
  - variables
  - etc.

Software Implementation

- Avoid long parameter lists
- Properly initialize data
- Principles of code layout
  - be consistent
  - use whitespace to enhance readability
    - blank lines between paragraphs, etc.
  - over-parenthesize expressions
  - avoid deep nesting
    - how? Create more routines
  - wrap long lines effectively
  - one statement per line

Code Example

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr,
  EMP_DATA empRec, float & estimRevenue, float ytdRevenue,
  int screenX, int screenY, COLOR_TYPE & newColor,
  COLOR_TYPE & prevColor, STATUS_TYPE & status,
  int expenseType )
{
for ( int i = 1; i <= 100; ++i ) {
  inputRec.revenue[ i ] = 0;
  inputRec.expense[ i ] = corpExpense[ crntQtr, i ];
  }
UpdateCorpDatabase( EmpRec );
estimRevenue = ytdRevenue * 4.0 / (float)crntQtr ;
newColor = prevColor;
status = Success;
if ( expenseType == 1 ) {
                        for ( int i = 1; i <= 12; ++i )
                        profit[ i ] = revenue[ i ] - expense.type1[ i ];
                        }
else if ( expenseType == 2 )  {
                              profit[ i ] = revenue[ i ] - expense.type2[ i ];
                              }
else if ( expenseType == 3 )
  {
  profit[ i ] = revenue[ i ] - expense.type3[ i ];
  }
}
```

Defensive Programming

- Asserts
  - Check the correctness of the assumptions we are making
- Parameter checking
  - Make sure the parameters we are receiving are “correct”

Error Handling

- Return codes
  - Return error information in the method return value
- Error state
  - Store error information in an object's state
- Exceptions
  - try
  - throw
  - catch

Testing

- Unit testing
  - module by module
  - designing test cases
    - line coverage
    - branch coverage
    - condition coverage
- System testing

Debugging

- Techniques for debugging:
  - Code reading
  - Tracing
  - Debuggers
- Find point where bug is manifest
  - crash
  - bad data
  - etc.
- Smallest possible input that will reproduce the bug
- Process of elimination to home in on offending code

Memory Errors

- Types of memory errors
  - Memory leak
  - Dangling pointer
  - Bounds error
- Strategies for debugging memory errors
  - Memory tracker
  - Memory watcher
  - Valgrind-style tools

Tools

- Preprocessor
- Compiler
- Linker
- Debuggers
- Make
- Profilers

Code Tuning

- Where to look for possible optimizations
- Optimize after the implementation - not as you go
- Find true bottlenecks
- Sources of inefficiency
  - unnecessary I/O operations
  - paging
  - system calls
  - interpreted languages
  - errors

Code Optimizations

- Logic
  - Stop when you know the answer
  - Order tests by frequency
  - Compare performance of similar logic structures
  - Substitute table lookups for complicated expressions
  - Use lazy evaluation

Code Optimizations

- Loops
  - Unswitching
  - Combining Loops
  - Unrolling
  - Minimizing the work inside loops
  - Sentinel Values
  - Putting the busiest loop on the inside
  - Replace multiplications that depend on the loop index with addition

Code Optimizations

- Data Transformations
  - Use integers rather than floating-point numbers
  - Use the fewest array dimensions possible
  - Minimize array references
  - Store sizes of variable-length data structures rather than computing on demand
  - Sort key indexes rather than complete objects
  - Use caching

Code Optimizations

- Expressions
  - Exploit algebraic identities
  - Use strength reduction
  - Initialize at compile time
  - Be wary of system routines
  - Use the correct type of constants
  - Precompute results
  - Eliminate common subexpressions