

What You Learn in CS 70

1 Style

You should be able to

- Discuss the value of good style, including
 - The impact (if any) of good style on program and programmer efficiency
 - The perils of programmer efficiency
- Relate the following concepts to programming style
 - Elegance
 - Consistency
 - Correctness
 - Organization
 - Idiom
 - Extensibility
- Name two C++ indentation styles and explain their differences
- Determine what aspects of a program require comments
- Place comments appropriately so that they are highly readable
- Devise appropriate variable names, based on context
- Apply strategies to reduce code complexity and amount of coding (“laziness”)
- Convert code to use idiomatic looping constructs
- Specify key design decisions and implementation ideas using pseudocode

Programming style is discussed in *The Practice of Programming*, by Kernighan and Pike (on reserve in Sprague) and also on a handout available on the course web site.

2 C++

2.1 C++ vs. JAVA

You should be able to

- Contrast C++ and JAVA with respect to
 - Terminology
 - Program layout (decomposition into source files)
 - Program safety
 - Language features, such as
 - * Colon initializers
 - * Code and data outside classes

- * C preprocessor
 - * Enumerated types
 - * Templates
 - * Explicit pointers
 - * Explicit memory management
 - User community
- Port simple text-based JAVA applications to C++

2.2 C++ Memory Model

You should be able to

- Express the memory layout of a program diagrammatically
- Describe and apply C++ scoping rules for local variables
- Determine when objects are allocated on the stack, and when on the heap
- Compare and contrast the heap and the stack
- Give a rationale for providing a stack as well as a heap
- Describe and apply **new** and **delete** for
 - Single objects
 - Arrays of objects
- Contrast and explain the rationale for both kinds of **new/delete**
- Explain the benefits and risks of aliasing via pointers
- Describe and detect the following coding errors
 - Double deletion
 - Memory leaks
 - Dangling pointers
 - Null-pointer dereferences
 - Pointer-to-object/pointer-to-array-of-object confusion
- Describe and use the **&**, *****, and **[]** operators
- Describe and use references
- Explain and contrast
 - Pass by value
 - Pass by constant reference
 - Pass by reference

including when it is appropriate to use each technique, and the lifetimes of the names and objects involved

- Apply and explain pointer arithmetic
- Use and explain primitive arrays
- Contrast primitive arrays with the STL's vector type

2.3 Basic C++ Object Programming

You should be able to

- Determine (for a given class declaration) whether the compiler will create default versions of the following constructs, and whether this default code will be okay
 - Default constructor
 - Copy constructor
 - Assignment operator
 - Destructor
- Enumerate the occasions that a the following functions will be called even though no explicit call has been written by the programmer
 - Default constructor
 - Copy constructor
 - Assignment operator
 - Destructor
- Apply “laziness” techniques to avoid writing duplicating code between a class's copy constructor and its assignment operator
- Explain and apply the technique used to disable copy constructors and/or assignment operators

2.4 C++ Language Features

You should be able to

- Use the C preprocessor to control the source code seen by the compiler proper, including
 - Including source lines from other files
 - Conditionally excluding source lines
 - Defining simple macros
- Explain any drawbacks of using the above features

- Describe and employ overloading
 - With operators implemented inside the class
 - With operators implemented outside the classincluding any restrictions that always (or should) apply
- Describe and employ type conversion
- Describe **friendship**, and determine when **friendship** is appropriate
- Determine and describe when and where **const** should be used
- Resolve problems that may occur when **const** is used
- Determine when member functions and data should be declared **static**
- Define and implement iterators
- Specify iterator invalidation semantics, and explain and abide by the iterator invalidation rules for standard STL types
- Contrast different iterator designs (e.g., STL style with isValid style)
- Define and use templated functions and classes
- Explain why a different file organization is usually needed for definitions of templated functions and classes as compared to non-templated code
- Describe and use important STL algorithms (e.g., nth_element, sort)
- Define and use function objects
- Explicitly instantiate a class template

2.5 Designing Classes

You should be able to

- Describe and employ encapsulation
- Determine which operations should be placed in the public interface
- Specify the behavior of a class from a users (interface) perspective
- List and contrast strategies for handling errors
- Employ nested classes
- Employ inheritance
- Describe and apply the rules for substitutability
- Define and use abstract base classes
- Determine when member functions should be declared **virtual**

3 Computational Complexity

You should be able to

- Define and relate $O(\dots)$, $\Omega(\dots)$, $\Theta(\dots)$, $o(\dots)$, $\omega(\dots)$
- Determine the statement(s) executed most in a code fragment
- Informally analyze code to determine its asymptotic behavior
- Determine and express the performance of code involving loops using Σ -notation
- Determine and express the performance of recursive code using recurrence relations
- Apply transformations to make code easier to analyze
- Improve algorithms to remove obvious inefficiencies
- Perform simple amortized-time analyses

4 Program Development With Standard UNIX Tools

4.1 Compiling

You should be able to

- Enumerate and explain the stages of compilation
- Use the g++ compiler tools to create
 - An executable program from a single source file
 - An object-code file from a single source file
 - An assembly-code file from a single source file
 - An executable program from multiple-object code files
 - A list of the files a source file depends on
- Explain and create Makefiles that include
 - Necessary and sufficient description(s) of file dependencies
 - Standard macro names (e.g, CXX)
 - Standard targets
- Describe and apply the algorithm used by make to rebuild files based on dependencies

GCC (which includes g++) and GNU Make are described in the online manual pages on turing, and also on the GNU Project's webpage at www.gnu.org.

4.2 Debugging

You should be able to

- Describe and employ strategies for reducing the amount and difficulty of debugging work
- Develop testing strategies
- Enumerate, rank, and order debugging approaches
- Employ assert statements to catch errors

5 Data Structures

5.1 Stacks, Queues, and Steques

You should be able to

- List the operations that are fundamental to stacks, queues, and steques
- Suggest appropriate asymptotic complexity for these operations
- Determine when each data structure is an appropriate choice
- Implement stacks, queues, and steques using
 - A static array
 - A dynamic array
 - A linked list
 - A circular linked list (queues and steques only)
 - A linked list of fixed-size chunksincluding an iterator that can traverse the structure

5.2 General Lists

You should be able to

- Suggest operations for generalized singly-linked and doubly-linked lists
- Contrast singly-linked and doubly-linked lists
- Implement
 - Singly-linked lists
 - Circular singly-linked lists
 - Doubly-linked lists
 - Circular doubly-linked listsincluding an iterator that can traverse the structure

5.3 Associative Containers: Sets and Maps

You should be able to

- Explain and contrast sets and maps
- Suggest operations that a set or map class should support
- Explain how (if sets and maps are appropriately defined) a set can be used to represent a map, and a map can be used to represent a set
- Enumerate and contrast potential representations for sets and maps

5.4 Hash Tables

You should be able to

- Explain and implement the following hash-table representations
 - Separate chaining
 - Linear probing
 - Quadratic probing
 - Double hashing

including deletion and assuming no upper limit on the number of items inserted

- Analyze the complexity of the above techniques for a “lightly loaded” hash table
- Explain perfect hashing and contrast it with the above techniques
- Discuss the desirable properties in a hash function
- Relate hash-table problems to classical problems in probability theory (e.g., the birthday problem, the coupon-collector problem)
- Describe when and where prime numbers may be useful in providing hash tables

5.5 General Trees

You should be able to

- Define and explain the following tree concepts:
 - Height
 - Depth
 - Ancestors
 - Descendants

- Path length
 - Pre-order, post-order, in-order, and level-order traversals
 - Perfect binary trees
 - Complete binary trees
- Represent trees and other graphs using the following representations:
 - Bit matrix
 - Adjacency list
 - Dominance drawing
 - Linked data structure
- Represent an arbitrary n -ary tree using a binary tree
- Represent a 2-3-4 tree elegantly using a binary tree

5.6 Binary Search Trees

You should be able to

- Describe the order condition for a BST
- Describe and implement insertion and deletion in a BST
- Describe and implement finding the n -th smallest value in a BST
- Explain the rationale for balancing BSTs, including when doing so is unnecessary
- Explain and apply left and right rotations
- Explain and contrast normal double rotations and splay double rotations
- Implement root insertion in a binary tree
- Provide and explain high-level pseudocode for
 - Randomized binary search trees
 - Splay trees
 - AVL trees
 - 2-3-4 trees
 - B-trees
- Contrast the trade-offs and performance differences of each of the above approaches

5.7 Heaps

You should be able to

- Distinguish between *the* heap and heap-ordered data structures
- State the heap-order and heap-structure conditions
- Explain how heaps can be represented as an array
- Explain and implement insert and deleteMin for a heap-ordered array

5.8 Disjoint-Set Union

You should be able to

- Implement a maze-drawing program given a disjoint-set-union data structure
- Explain the rationale for path-compression and union-by-rank (in general terms)