CS 180 Final Exam Terms and Concepts

Last modified: 27 April 2020

Chapter 1

- hardware components: main memory, secondary memory, central processing unit (CPU)
- main memory: random-access memory (RAM), address
- kinds of values stored in memory: integers, real numbers (floats), characters (using ASCII), program instructions
- secondary memory: hard disk, floppy disk, CD, flash drive
- programming languages: machine language, assembly language, high-level languages, syntax vs. semantics

Chapter 2

- Components of a program: explanatory comment, preprocessor includes, namespace statement, constants, struct definitions, function prototypes with Javadoc, main function definition, other function definitions
- use of the cout statement, including stream insertion << and endl
- variables: programmer-defined named storage locations in memory
    - must be declared
    - must have a type
    - must have a name that follows valid identifier rules
    - may be initialized with a value at declaration time
- data types: A data type is a set of values and a set of operations defined on those values. Know which is appropriate for each kind of data. Data types so far include:
    - integer types: short, int, long, long long; each may be unsigned. Know about binary (base 2), hexadecimal (base 16) and octal (base 8) notation for literals
    - char: character data type for single characters using single quotes for literals
    - string: old-fashioned C-strings that we will use for string literals and the C++ string class that we will use for variables
    - floating-point types: meaning float, double, long double. Literals can use scientific notation or conventional notation; always use a decimal point.
    - bool: meaning a Boolean type, with true and false values

- scope: is the regions of the program in which the variable exists and its name can be legally used. Declaration should be close to where the variable is first used and in as small a scope as possible.
- assignment operation: uses the = sign. Distinguish left-hand side (lhs, resolves to lvalue aka address) from right-hand side (rhs, resolves to rvalue aka a value). One form is initialization, can only be done once per variable
- unary minus: negates a numeric value
- addition, subtraction, and multiplication: + - * work the way you would expect
- division and modulus: division is straightforward for floats and works like on your calculator. With integer types, division gives the quotient. Modulus gives the integer remainder. Modulus can distinguish even from odd, pick out digits, etc.
- parentheses are used to override operator precedence

Chapter 3

- stream extraction reads data from standard input and converts it to the type of the variable
- Mathematical operators use precedence and associativity the way you learned in your algebra class.
- Many more mathematical operations are available from the <cmath> library, including pow
- Overflow and underflow can result from arithmetic operations. With integers they just wrap around without causing errors. With floating point types overflow can cause problems.
- Mixed type arithmetic operations are possible, but discouraged. Use type casting instead: static_cast<double>(foo) casts integer foo to a double
- Combined assignment: x += 1; y *= 5;
- Formatting Output: setw(x), fixed, setprecision(x), showpoint

Chapter 4

- relational operators: >, <, >=, <=, ==, !=, what can and cannot be compared, how to handle comparison for floating point types
- bool type: true, false, boolalpha IO manipulator
- if statement: conditional execution, else, else if, nested if statements, always use braces

- logical operators: && (and, binary), || (or, binary), ! (unary not), operands are Boolean, return value is bool
- truth tables: to explain logical operators
- relops and characters: use ASCII values for comparison. Know order of ASCII table.
- relops and C++ strings: character by character; cannot compare C-strings with relops
- conditional operator: var = x < 0 ? -x : y
- flag variables: Boolean variables that store a condition value.
- scope: of a variable, the region of the program in which the variable exists and can be referred to

Chapter 5

- increment and decrement operators: ++, --, prefix vs. postfix form, be able to evaluate expressions that use these
- while loop: a pretest loop, body executes zero or more times
- counters and accumulators: incrementing vs. accumulating, both used frequently, both must be initialized before the loop
- do while loop: posttest loop, will always execute at least once, compare and contrast with while loop; often used for input validation
- loop control with flags: while (!done) for example
- for loop: for (initialization; test; update), another pretest loop. Know the number of loop iterations
- nested loops: know purposes, number of total loop iterations
- never use break or continue with loops. Never return from a loop
- text file IO: what text files are, streams for reading and writing, opening streams, using streams, closing streams, stream extraction for numbers vs. getline for C++ strings

Chapter 6

- functions: modularization, reuse, return type, parameter list, special void return type
- function prototypes: declare them
- parameters: formal parameters vs. actual parameters (aka parameters vs. arguments), formal: type and name, actual: value (could be variable, expression, or literal)

- naming functions: functions do something so names should contain verbs
- return statements: where they can appear, make sure non-void functions always return a value
- global variables: never use these. Global constants are acceptable if used in more than one function
- local variables: defined inside functions, scope, parameters are like pre-initialized local variables
- pass-by-value parameters: argument is copied into formal parameter when function is called, formal parameter can be used as a variable, but it is just a copy
- pass-by-reference parameters: reference variable is a reference to another variable, declared using ampersand (&), an alias for another variable, changes actually change real variable, arguments must be variables
- function design: function should do only one thing
- Javadoc: for documenting function prototype, including function purpose, parameters using @param tag, return value (if non-void) using @return tag

Chapter 7

- arrays: various ways to declare and initialize arrays, distinguishing type of index (always an unsigned type, usually size_t) versus type of data (can be any type, access elements by position using square brackets, size fixed at compile time, importance of bounds checking
- range-based for loop: aka the foreach loop, loops through array of values automatically, use a reference variable to modify elements of the array, mostly used for vectors (later)
- whole array assignment and comparison: only way is item by item, usually using a loop
- common array algorithms: print contents, sum contents, compute average, find min or max element, find position of min or max or some arbitrary element
- parallel arrays: same position used for data about the same entity, but stored in different arrays, perhaps of different types
- arrays as function parameters: generally always pass-by-reference (actually pass by pointer, explained later), need to pass the size separately, should be const if array is unchanged in function
- multi-dimensional arrays: relevant to lots of real-world situations, double-subscripting for two-dimensional arrays, nested for loops are especially helpful, you need to have a mental image of the data to know how to use the array, use

dimension parameter names to help with this, passing multi-dimensional arrays as parameters, must specify all dimensions except the leftmost
- array problems: static size, size must be known at compile time, arrays don't know their size, lack of bounds checking
- vectors: from the Standard Template Library (STL), how to declare, how to initialize, how to add values (.push_back), use of size_t, .size() to determine size, .at(index) to access elements, always pass by reference or const reference

Chapter 8

- searching: linear search is pretty much the only option for unsorted data, usually implemented with a while loop, main operation is comparison, return position of matched element or size (as opposed to -1) to indicate item is not found, analysis: requires n/2 comparisons on average when item is found, n comparisons when item is not found
- binary search: useful for searching sorted lists, know the algorithm, be able to specify what elements are examined during a binary search, be familiar with implementation, analysis: takes log (base 2) of n comparisons, cuts search space in half at each step, know powers of 2
- sorting: put values in an array (or vector) in nondecreasing order, understand bubble sort and selection sort algorithms and be able to work through them yourself, understand how code works

Chapter 9

- variable addresses: be familiar with address-of operator (&) and sizeof operator
- pointer variables: variable to hold an address, similar to, but different from reference variables, declaring a pointer (int* pointer = &value;), usually draw memory pictures with pointers as arrows to address they point to, dereference a pointer using * operator
- pointers and arrays: array names are const pointers, can use them as pointers or use standard array syntax, pointer arithmetic allows code to dereference the next address using pointer + 1 syntax, compiler does this because it knows the size of things, can compare pointers, since array names are pointers, arrays are passed by pointer very similarly to pass by reference
- dynamic variable allocation: compile time variables are allocated from the runtime stack, dynamically allocated variables are allocated from the heap, new operator is used for dynamic allocation, new returns the address

of the allocated memory, so pointer variable is necessary, normally involves dynamic allocation of arrays whose size is determined at allocation time
- memory management: new operator allocates memory, delete operator deallocates memory, it is important that every new must eventually be followed by a delete, otherwise your program has a memory leak

Chapter 10

- character functions: be familiar with these, such as isalpha, isdigit, along with conversion functions toupper and tolower
- C++ string: including functions to convert strings to numbers, such as stoi, stoul, stod, use of to_string to convert all numeric types to a string representation (note the function overloading), concatenation with +, comparison with standard comparison operators (==, <= . . .), use of .at() to access single characters, .length() and .substr() methods, use of string::npos as the largest possible size_t value.

Chapter 11

- abstraction: defines the common characteristics of a thing, capturing the essence of the thing, distinct from particular instances of the thing.
- abstract data type (ADT): combines the idea of abstraction with data types. ADTs are data types defined by the programmer, consisting of one or more data fields and operations that can be performed on instances of the ADT.
- structs: primary C++ mechanism for building ADTs. Know how to define, initialize, define variables of the struct type, pass as parameters to function, store in vectors/arrays. Be familiar with the use of the dot operator. Know where to define structs in the C++ file, according to our style.
- struct operations: You can copy structs using regular assignment. But comparing requires you to define your own comparisons, depending on what fields you want to compare. Printing is generally done by defining a to_string function, built up from the to_string functions of the component fields.
- nested structures: a member field can be of any data type, including a programmer-defined ADT type. Can be initialized with nested initializer lists and accessed with nested dot notation.
- pointers to structs: require use of parentheses for dereferencing: (*mptr).title for example, so instead we use the dereference-then-select operator ->, for

example: mptr->title. With pointers we can dynamically allocate structs, but we won't really do so in this class.

- overloading: a topic from 6.14 that we skipped. Idea is multiple functions with the same name and same intended functionality but with different signatures meaning the data types of the functions parameters in order. So a function name can be overloaded if the types in the parameter list differ, meaning different number or different arrangement of types. Note that merely having a different return type is not enough to allow overloading. And note potential ambiguity when type coercion happens.
- In addition, we have provided checkpoint exercise numbers, review question numbers for this chapter that you can use to best prepare for the final