CS 13001  Computer Science I: Programming and Problem Solving  4 credit hours

Instructor’s Name: ___________________________________________________

(Textbook Title, Author, Year)

(Other Supplemental Material)

Course Content:
Computer science concepts, including algorithm development and problem-solving strategies focused on procedural abstraction. High-level programming concepts, including data types, expressions, program structures, functions, parameter passing, scope, extent, arrays, introduction to recursion and an introduction to object oriented-concepts.

Prerequisites or co-requisites: Minimum C grade in MATH 11010 or MATH 12001. Required (BS-CS, BS-GP, BS-IS, BA-CS, CS-Minor)

Topics to be Covered:

Tier 1: SDF/Algorithms and Design [11 hours]:
- The concept and properties of algorithms
- Informal comparison of algorithm efficiency (e.g., operation counts)
- The role of algorithms in the problem-solving process
- Problem-solving strategies
- Iterative and recursive mathematical functions
- Iterative and recursive traversal of data structures
- Divide-and-conquer strategies
- Fundamental design concepts and principles
- Abstraction
- Program decomposition
- Encapsulation and information hiding
- Separation of behavior and implementation

Tier 1: SDF/Fundamental Programming Concepts [10 hours]
- Basic syntax and semantics of C++
- Variables and primitive data types (e.g., numbers, characters, Booleans)
- Expressions and assignments
- Simple I/O including file I/O
- Conditional and iterative control structures
- Functions and parameter passing
- The concept of recursion
Tier 1: SDF/Fundamental Data Structures [12 hours]

- Arrays
- Records/structs (heterogeneous aggregates)
- Strings and string processing
- Classes
- Abstract data types and their implementation
- Vectors
- References and aliasing

Tier 1: SDF/Development Methods [5 of the 10 required hours]

- Program comprehension
- Program correctness
- Types of errors (syntax, logic, run-time)
- The concept of a specification
- Defensive programming (e.g. secure coding, exception handling)
- Code reviews
- Testing fundamentals and test-case generation
- The role and the use of contracts, including pre- and post-conditions
- Unit testing
- Simple refactoring
- Modern programming environments
- Programming using library components and their APIs
- Debugging strategies
- Documentation and program style

Learning Outcomes:

Tier 1: SDF/Algorithms and Design [11 hours]:

1. Discuss the importance of algorithms in the problem-solving process. [Familiarity]
2. Discuss how a problem may be solved by multiple algorithms, each with different properties. [Familiarity]
3. Create algorithms for solving simple problems. [Usage]
4. Use a programming language to implement, test, and debug algorithms for solving simple problems. [Usage]
5. Implement, test, and debug simple recursive functions and procedures. [Usage]
6. Determine whether a recursive or iterative solution is most appropriate for a problem. [Assessment]
7. Implement a divide-and-conquer algorithm for solving a problem. [Usage]
8. Apply the techniques of decomposition to break a program into smaller pieces. [Usage]
9. Identify the data components and behaviors of multiple abstract data types. [Usage]
10. Implement a coherent abstract data type, with loose coupling between components and behaviors. [Usage]
11. Identify the relative strengths and weaknesses among multiple designs or implementations for a problem. [Assessment]

Tier 1: SDF/Fundamental Programming Concepts [10 hours]
1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs variables, expressions, assignments, I/O, control constructs, functions, parameter passing, and recursion. [Assessment]
2. Identify and describe uses of primitive data types. [Familiarity]
3. Write programs that use primitive data types. [Usage]
4. Modify and expand short programs that use standard conditional and iterative control structures and functions. [Usage]
5. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing. [Usage]
6. Write a program that uses file I/O to provide persistence across multiple executions. [Usage]
7. Choose appropriate conditional and iteration constructs for a given programming task. [Assessment]
8. Describe the concept of recursion and give examples of its use. [Familiarity]
9. Identify the base case and the general case of a recursively-defined problem. [Assessment]

**Tier 1: SDF/Fundamental Data Structures [12 hours]**
1. Discuss the appropriate use of built-in data structures. [Familiarity]
2. Write programs that use each of the following data structures: arrays, records/structs, strings, classes. [Usage]
3. Compare alternative implementations of data structures with respect to performance. [Assessment]
4. Describe how references allow for objects to be accessed in multiple ways. [Familiarity]
5. Compare and contrast the costs and benefits of dynamic and static data structure implementations. [Assessment]
6. Choose the appropriate data structure for modeling a given problem. [Assessment]

**Tier 1: SDF/Development Methods [5 hours]**
1. Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
2. Explain why the creation of correct program components is important in the production of high-quality software. [Familiarity]
3. Identify common coding errors that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors. [Usage]
4. Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Usage]
5. Apply a variety of strategies to the testing and debugging of simple programs. [Usage]
6. Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers. [Usage]
7. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]
Abet Learning Outcomes

1. Analyze a complex computing problem and to apply principles of computing and other relevant disciplines to identify solutions.
2. Design, implement, and evaluate a computing-based solution to meet a given set of computing requirements in the context of the program’s discipline.
3. Recognize professional responsibilities and make informed judgments in computing practice based on legal and ethical principles.
4. Function effectively as a member or leader of a team engaged in activities appropriate to the program’s discipline.
5. Apply computer science theory and software development fundamentals to produce computing-based solutions.

Learning Outcome Assessments:

1. Multiple small programming projects
2. Weekly assignments and quizzes
3. 3 exams