PRACTICE FINAL CMPUT 201

ROADMAP

Start with the VOCAB & THEORY section of your practice final. These are guaranteed multiple-choice questions and cover the core terminology and definitions that show up in everything else. Work through 10 to 15 questions at a time. For each one you get wrong or aren't confident about, write down why the wrong answers are incorrect and what concept the right one is testing. Build flashcards or a cheat sheet based on key terms like dangling pointers, command redirection, struct passing, and memory allocation. Don't just memorize—run small programs that demonstrate how things like `malloc`, `argv`, or `sizeof` work in real code.

Next, move into the LECTURE CODE QUESTIONS. These problems test your ability to reason through C code and predict its behavior. Open a C file and try typing each example out, compiling it with `-Wall -Werror`, and running it. Focus on undefined behavior like printing uninitialized variables or modifying string literals. Practice writing your own versions of functions like `print_binary`, `strlen`, and `replace`. These aren't hard to write but will show up on the exam in subtle forms. Make sure you understand pointer mutation, scope, function parameters, and what actually changes in memory.

After that, go through the LABS. Don't just reread them—rebuild the core logic of each one. Start with Lab 3 and write a small script that runs an executable with redirected input. Then do Lab 5's Hamming distance logic, both decimal and binary. Try reimplementing the core of Lab 6 (Bulls and Cows) with test cases. Lab 7 is critical for practicing heap allocation and string handling. Lab 8 is good for practicing grid logic and boundary checking. And Lab 9 gives you realistic practice with file input, parsing, and editing text in-place. Focus on memory safety and error handling as you rebuild these.

Once labs are refreshed, review the ASSIGNMENTS. Read the problem descriptions and try rebuilding the core function or logic in a fresh file. For Assignment 2, write `luhns.c` again without using arrays or pointers. For Assignment 3, practice dynamic set handling and how to grow arrays safely. Reimplement the `sequences.c` arithmetic logic, then sketch the sepia filter formula from Assignment 3's image transformer. For Assignment 4, review how the Maze ADT is structured and what functions interact with it. Focus on how state is stored, how special tiles are handled, and how you prevent invalid moves.

Then turn to the PRACTICE PROBLEMS FROM SLIDES. These are quick-hit problems that train implementation skills. Pick one or two per study session. Fully write and test them. Start with `str_to_int`, `argmax`, and `str_replace`. Practice writing `primes`, a basic `stack` with push/pop, and a simplified `wc` clone. For each, sketch the function, write the implementation, and test it with dummy input. Think through memory use, parameter design, and how the function should behave if inputs are missing or invalid.

Use your FINAL NOTES and custom questions to tie everything together. Go through each of the "Explain" or "Write" questions and answer them in your own words or code. Treat them like mini short-answer prompts. Practice explaining memory layout, how array decay works, why passing structs

by value is slow, or how a subshell works in bash. For advanced questions, like function pointers or 2D array performance tradeoffs, try a small working example or sketch it out in plain English.

(Also: watch Youtube videos to explain key concepts that you're having trouble with.)

VOCAB & THEORY

1. Which of the following best describes a shell in Unix-like systems?
   A) A graphical interface for file management
   B) A command-line interpreter that executes user commands.
   C) A background process manager.
   D) A system logging service.

2. In bash, what does the shebang (#!) line at the top of a script specify?
   A) The version of bash required.
   B) The script's execution priority.
   C) The location of configuration files.
   D) The absolute path to the interpreter that will run the script.

3. Which variable determines the search path for executable files in bash?
   A) PATH
   B) HOME
   C) SHELL
   D) USER

   ? To search a path, using PATH variable
   e.g. PATH="${PATH}:/home/rob/cmput201/cmput201_bin"

4. Which symbol is used to redirect standard output to a file in bash?
   A) <
   B) >
   C) |
   D) 2>

5. In bash, "globbing" refers to:
   A) Combining multiple commands with pipes.
   B) Expanding wildcard patterns to match filenames.
   C) Redirecting error messages to a file.
   D) Running a command in a subshell.

6. What is the primary purpose of the main() function in a C program?
   A) To declare variables.
   B) To serve as the program's entry point.
   C) To allocate dynamic memory.
   D) To define custom data types.

7. Which statement best distinguishes a compiled language (e.g., C) from an interpreted language?

A) Compiled code is executed directly without translation.
B) C code is transformed into machine code before execution.
C) C uses runtime interpretation to execute commands.
D) Interpreted languages do not require source code.

8. Which of the following is a valid preprocessor directive in C?
A) #define MAX 100
B) int MAX = 100;
C) #include <stdio.h>
D) printf("MAX");

9. What does it mean that C is "statically typed"?
A) Variables can change their type at runtime.
B) All variable types are known and checked at compile time.
C) Variables are declared without types.
D) Types are determined by the runtime environment.

10. Which function from the standard library is used for printing output to the console?
A) scanf()
B) puts()
C) printf()
D) print()

11. What is the purpose of the malloc() function in C?
A) To allocate memory dynamically on the heap.
B) To free allocated memory.
C) To declare a global variable.
D) To initialize static arrays.

12. Which function must be paired with malloc() to avoid memory leaks?
A) release()
B) free()
C) dealloc()
D) delete()

13. A memory leak in a C program occurs when:
A) Memory is allocated but never freed.
B) Memory is allocated on the stack.
C) The program uses global variables.
D) Memory is accidentally overwritten.

14. What is a dangling pointer?
A) A pointer that correctly points to allocated memory.
B) A pointer that references freed or invalid memory.

C) A pointer declared without an initial value.
D) A pointer that holds multiple addresses.

15. Which practice helps prevent dangling pointers?
A) Reassigning pointers without freeing memory.
B) Immediately setting pointers to NULL after freeing memory.
C) Avoiding the use of pointers in functions.
D) Allocating memory on the stack only.

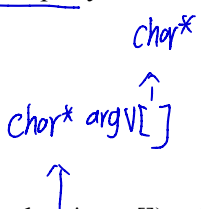
16. Which operator in C is used to obtain the address of a variable?
A) *
B) &
C) %
D) $

17. In C, what does it mean to "dereference" a pointer?
A) To change the pointer's address.
B) To access the value stored at the pointer's address.
C) To declare a pointer variable.
D) To free the pointer's memory.

18. Pointer arithmetic is especially useful for:
A) Iterating through elements in an array.
B) Changing variable types.
C) Converting strings to integers.
D) Preventing memory leaks.

19. Which of the following correctly declares a pointer to an integer?
A) int ptr;
B) int *ptr;
C) int ptr;
D) pointer int;

20. To assign the address of variable x to pointer ptr, you would write:
A) ptr = x;
B) ptr = &x;
C) *ptr = x;
D) &ptr = x;



21. In the function signature int main(int argc, char *argv[]), what does argc represent?
A) The number of command line arguments passed (including the program name).
B) The array of command line arguments.
C) The exit code of the program.
D) The size of argv in bytes.

22. In a C program, argv is best described as:
    A) A pointer to a single string.
    B) An array of pointers to strings.
    C) A two-dimensional integer array.
    D) A pointer to an integer.

23. Command line arguments in C are typically accessed using:
    A) Global variables.
    B) The argv array passed to main().
    C) Standard input functions.
    D) The getenv() function.

24. What is generally stored in argv[0]?
    A) The first user-supplied argument.
    B) The program's executable name or path.
    C) The total count of arguments.
    D) An error code.

25. What does "array decay" mean when passing an array to a function?
    A) The array's size decreases with each function call.
    B) The array automatically converts into a pointer to its first element.
    C) The array becomes read-only in the function.
    D) The function creates a copy of the array.

26. Which operator returns the size (in bytes) of a variable or array in C?
    A) len()
    B) count()
    C) sizeof → determine the size of a variable or array
    D) sizeof() → determine the size of a data-type or a type name

27. How would you correctly declare an array of 10 integers?
    A) int arr(10);
    B) int arr[10];
    C) int arr = new int[10];
    D) int arr{10};

28. When passing an array to a function, why is it important to pass the array's length as well?
    A) Because the function needs to know how many elements are available.
    B) Because the array's memory address is lost.
    C) Because the compiler automatically resizes the array.
    D) Because arrays are passed by value by default.

29. Which best describes a pointer-to-pointer in C?

A) A pointer that stores the address of another pointer.
B) An array of pointers.
C) A double-sized pointer.
D) A pointer used exclusively with two-dimensional arrays.

30. What is the primary difference between an array and a pointer?
A) An array name is a constant pointer to its first element, whereas a pointer is a variable holding a memory address.
B) Arrays can be reassigned; pointers cannot.
C) Pointers always allocate dynamic memory; arrays do not. It can also point to statically-allocated memory
D) There is no difference; they are completely interchangeable.

31. In C, what does mutation refer to when working with pointers?
A) Changing a variable's value using its pointer.
B) Converting a pointer to an integer.
C) Copying an array into a new variable.
D) Declaring a constant pointer.

32. How can a C function effectively return multiple values?
A) By using global variables.
B) By returning an array.
C) By passing pointers as parameters to update caller variables.
D) By using recursion.

33. What is the primary purpose of a double pointer in mutation functions?
A) To hold two independent integer values.
B) To allow a function to update the caller's pointer variable.
C) To create two-dimensional arrays only. X
D) To prevent a pointer from being changed.

34. Which of the following best describes a potential problem when using mutation with pointers?
A) Creating a new variable on each function call.
B) The possibility of generating dangling pointers.
C) Automatic conversion of pointers to arrays.
D) Returning data by value instead of address.

35. When updating a dynamic array through mutation, a common operation is:
A) Halving the allocation size.
B) Doubling the allocation size when capacity is reached.
C) Moving the array from the heap to the stack.
D) Changing the array's element types.

36. Why is it important to check if a pointer is NULL before using it in a mutation function?
A) To force the pointer to become an array.

B) To prevent segmentation faults by ensuring valid memory access.
C) To automatically free the pointer's memory.
D) To convert the pointer to an integer.

37. Which statement best describes using pointers for mutation?
A) It creates a complete copy of the data in the function's scope.
B) It allows the function to modify the caller's data directly via memory addresses.
C) It restricts the function to only reading data.
D) It avoids the need for dynamic memory allocation.

38. In a mutation function, why might passing the array's length be critical?
A) It determines how many elements need to be processed.
B) It triggers automatic pointer conversion.
C) It allocates memory for new elements.
D) It prevents the array from decaying.
A more rigorous statement can be: "It ensures that only valid entries of the array are processed, preventing out-of-bounds access and memory errors.

39. When using a double pointer to update an array, which operation is essential?
A) Incrementing the pointer's value without dereferencing.
B) Dereferencing the double pointer to access and update the original pointer's value.
C) Casting the pointer to an integer.
D) Freeing the pointer before updating.

40. Which error might occur if a mutation function improperly frees memory still in use?
A) Memory leak.
B) Dangling pointer leading to segmentation fault.
C) Buffer overflow.
D) Compile-time error.

41. What is a safe approach for a function to return multiple computed values?
A) Returning a pointer to a local variable.
B) Using pointer parameters to store results so the caller's variables are updated.
C) Allocating memory on the stack for the results and returning it directly.
D) Using global variables exclusively.

42. Why is it problematic to pass pointers to local stack variables for mutation purposes?
A) The local variables are copied automatically.
B) The lifetime of stack variables ends when the function returns, leading to undefined behavior if accessed later.
C) They cannot be dereferenced.
D) They are automatically converted to NULL.

43. What is the main goal of separate compilation in C?
A) To compile all source code into one file for faster execution.
B) To compile each source file independently and then link them together.

C) To avoid using header files.
D) To prevent runtime errors.

44. A header guard is used to:
A) Speed up code execution.
B) Prevent multiple inclusions of the same header file during compilation.
C) Link object files together.
D) Allocate memory for variables in headers.

45. Why are forward declarations important?
A) They automatically free memory.
B) They allow functions and variables to be recognized before their full definitions appear.
C) They compile source code faster.
D) They enable code optimization by the linker.

46. An object file in C is:
A) A text file containing source code.
B) A binary file resulting from compiling a source file without linking.
C) A header file for a library.
D) A script for the preprocessor.

47. What does the linker do in the build process?
A) Checks for syntax errors.
B) Combines object files into a single executable.
C) Allocates dynamic memory during runtime.
D) Interprets preprocessor directives.

48. Which flag in gcc tells the compiler to compile source files into object files without linking?
A) -o
B) -c
C) -l
D) -D

49. How can compile-time definitions be passed to the compiler?
A) Using the -c flag.
B) Using the -D flag to define identifiers and values.
C) Including them in the source code only.
D) Through linker scripts.

50. What error might occur if a header file is included multiple times without proper guards?
A) Segmentation fault at runtime.
B) Multiple definition errors during compilation.
C) Memory leaks in the executable.
D) A warning that can be safely ignored.

51. How do header files facilitate separate compilation?
    A) They allow function implementations to be compiled repeatedly.
    B) They provide declarations and macros that ensure consistency across multiple source files.
    C) They are only used for documentation purposes.
    D) They automatically link object files.

52. Which of the following is NOT a benefit of separate compilation?
    A) Faster compile times due to modular builds.
    B) Improved code organization.
    C) Reduced risk of runtime segmentation faults. Separate compilation only helps organize and modularize
    D) Independent development of project modules. the code.

53. What is a struct in C?
    A) A built-in data type like int or char.
    B) A user-defined composite data type that groups related variables.
    C) A function for handling dynamic memory.
    D) A pointer type for arrays.

54. How do you access a member of a struct through a pointer?
    A) Using the dot operator (.)
    B) Using the arrow operator (->)
    C) Using the ampersand (&)
    D) Using the asterisk (*) operator

55. Which of the following is the correct way to declare a struct variable after defining a struct?
    A)
    ```
    struct Person { char name[20]; int age; };
    Person p;
    ```
    this is correct for C++ but not for C
    B)
    ```
    struct Person { char name[20]; int age; };
    struct Person p;
    ```
    C)
    ```
    Person { char name[20]; int age; } p;
    ```
    D)
    ```
    struct Person { char name[20]; int age; } p;
    ```

56. What is one advantage of passing a struct by pointer instead of by value?
    A) It creates an independent copy for the function to work with.
    B) It avoids copying large amounts of data, reducing overhead.
    C) It prevents the function from modifying the original struct.
    D) It automatically frees the memory of the struct.

57. In a struct declaration, why must you include a semicolon after the closing brace?

A) It is optional in C.
B) It indicates the end of the struct definition.
C) It allocates memory for the struct.
D) It separates the struct from function declarations.

58. What disadvantage might result from passing a large struct by value to a function?
A) The function cannot access the struct members.
B) It can lead to increased memory usage and slower performance due to copying.
C) It automatically converts the struct into a pointer.
D) It causes the struct to be read-only within the function.

59. How can you simulate encapsulation with structs in C?
A) By declaring all members as public.
B) By using pointers and providing functions to access or modify the data.
C) By avoiding the use of header files.
D) By using global variables to store struct members.

60. Which is a reason to use pointers to structs rather than nesting structs directly?
A) Pointers reduce the amount of code needed.
B) Pointers can help prevent large copies and allow sharing of data between functions.
C) Nested structs are not supported in C.
D) Pointers automatically manage memory.

B, D, A, B, B, B, B, A, B, C, A, B, A, B, B, B, B, A, B, B, A, B, B, B, B, C, B, A, A, A, A, C, B, B, B, B, B, B, A, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B

LECTURE CODE QUESTIONS

1. What is problematic about the following code, and what behavior might you expect?

```
#include <stdio.h>

int blah() {
  int z;
  printf("z: %d\n", z);
}
int main() {
  blah();
  int x;
  printf("%d\n", x);
  int y;
  printf("%d\n", y);
  blah();
}
```

In int blah(), variable z is not initialized before being printed; no values to return

The program can be successfully compiled but undefined behaviors will occur for int blah() when it is called and uninitialized int variables x and y.

2. Examine these three programs. What will the shell variable $? contain after each program runs, and why?

```
// three.c
int main() {
 return 3;
}
```

Shell variable $? will contain 3
When the program finishes executing, it will exit with a status code of 3, and using gcc to compile and then running the executable will enable shell variable to capture the status code of the program

```
// five.c
int main() {
 return 5;
}
```

Shell variable $? will contain 5
The reason is the same as mentioned above.

```
// big_status_code.c
int main() {
 return 258;
}
```

8-bit binary number from 00000000 ~ 11111111
(0) (255)
Shell variable $? will contain 2
The reason is that when the value to return exceeds 255, a circulation will be formed

3. Write a function that reads a character from standard input and converts lowercase letters to uppercase: char ch; ch = ch + ('A' - 'a');
<0

4. What's wrong with this binary printing function, and how is it fixed in the correct version?

```
// broken_print_binary.c
void print_binary(unsigned int num) {
 while (num != 0) {
  if (num % 2) {
   printf("1");
  } else {
   printf("0");
  }
  num = num/2;
 }
 printf("\n");
}
```

19 19/2=9 19%2=1 9/2=4 9%2=1
11
19
10011
1
1x2=2<19
2x2=4<19
4x2=8<19
8x2=16<19
16x2=32>19
32/2=16

2 | 19
2 | 9 ... 1
2 | 4 ... 1
2 | 2 ... 0
2 | 1 ... 0
0 ... 1

5. Implement a function that correctly prints the binary representation of an unsigned integer:

6. Why doesn't the following function modify the value of x in the main function?

```
#include <stdio.h>

void times2(int x) {
 x = x*2;
```

```
}

int main() {
  int x = 10;
  times2(x);
  printf("%d\n", x);
}
```

7. Explain the concept of variable scope using this example:

```
#include <stdio.h>

int main() {
  for (int i = 0; i < 10; ++i) {
    printf("%d\n", i);
  }

  int x;
  int y;
  for (x = 0, y = 0; x*y < 100; x = x+2, y = y+3) {
    printf("(%d, %d)\n", x, y);
  }

  printf("x: %d, y: %d\n", x, y);
  printf("i: %d\n", i);  // Will this compile? Why or why not?
}
```

8. Explain how command line arguments are accessed in C using this example:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("We received %d cmd line arguments\n", argc);
  for (int i = 0; i < argc; ++i) {
    printf("argv[%d]: %s\n", i, argv[i]);
  }
}
```

9. Write a program that reads integers from standard input and prints them to standard output:

10. What's the difference between writing to stdout and stderr as shown in these programs?

```
// print_stderr.c
#include <stdio.h>

int main() {
```

```
  int x;
  while (scanf("%d", &x) == 1) {
    fprintf(stderr,"%d ", x);
  }
  fprintf(stderr,"\n");
}
```

11. Implement a program that reads 10 integers and prints them in sorted order:

12. How can you determine the length of an array in C? Explain using code:

13. Write a function that replaces all occurrences of a value in an array:

14. Explain the difference between these data types and their sizes:

```
#include <stdio.h>

int main() {
  int x = 5;
  char c = 'a';
  char *p = &x;
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int *z = arr;
  printf("sizeof(x): %lu\n", sizeof(x));
  printf("sizeof(c): %lu\n", sizeof(c));
  printf("sizeof(p): %lu\n", sizeof(p));
  printf("sizeof(arr): %lu\n", sizeof(arr));
  printf("sizeof(z): %lu\n", sizeof(z));
  printf("sizeof(*p): %lu\n", sizeof(*p));
  printf("sizeof(*z): %lu\n", sizeof(*z));
}
```

15. Why does this function not work correctly, and how should it be fixed?

```
#include <stdio.h>

void printArray(int arr[]) {
  unsigned int len = sizeof(arr)/sizeof(arr[0]);
  for (unsigned int i = 0; i < len; ++i) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}

int main() {
  int a[] = {1, 2, 3, 4, 5};
```

```
  int b[] = {7, 3, 2};
  printArray(a);
  printArray(b);
}
```

16. What is the correct way to pass an array to a function? Show using code:

17. Explain the relationship between arrays and pointers in C using this code:

```
#include <stdio.h>

int main() {
  int arr[4] = {1, 2, 3, 4};
  int *p = arr;
  printf("sizeof(arr): %lu\n", sizeof(arr));
  printf("sizeof(p): %lu\n", sizeof(p));
  printf("p: %p\n", p);
  printf("arr: %p\n", arr);
  printf("&p: %p\n", &p);
  printf("&arr: %p\n", arr);
}
```

18. How would you implement your own version of the strlen function?

19. What is the difference between using char myStr[] and const char *s when working with strings in C?

20. Explain why the following code crashes and how to fix it:

```
char *p = "On text/data";
p[1] = 'o';
```

21. What is the purpose of the null terminator (\0) in C strings? How does it affect string functions?

22. What is the potential problem with the following code and how would you fix it?

```
char myS[] = {'a', 'b', 'c', 'd', '\0'};
const char *s = " copy me";
strcat(myS, s);
```

23. Why does comparing strings with < or > operators not work as expected in C? What should you use instead?

24. What makes a character array a valid C string? How can a string's length differ from its allocated size?

25. How would you implement a dynamic growing array (similar to a vector) in C?

26. Write a function to add an element to the end of a dynamic array, growing the array when necessary.

27. Implement a function to remove and return the last element from a dynamic array.

28. What are the implications of using assert in functions like pop and ith in the list implementation?

29. Why is properly freeing memory important in data structures like linked lists? Identify potential memory leaks in the provided code examples.

30. How could you modify the linked list implementation to prevent users from directly accessing the internal structure fields?

31. Write a function to create an empty linked list structure in C.

32. How would you implement a function to add an element to the front of a linked list?

33. Write a function to retrieve the ith element in a linked list.

34. Implement a function to remove the ith element from a linked list and return its value.

35. Write a function to destroy/free a linked list using a tail-recursive approach.

36. Write a function to destroy/free a linked list using an iterative approach.

37. Explain the difference between recursive and tail-recursive implementations for destroying linked list nodes.

38. What function would you write to move a rectangle to a new position by adding offsets to its current coordinates?

39. How would you define a structure to represent a rectangle with position and dimensions?

40. Write code to create a rectangle at position (0,0) with width 5 and height 3, then move it to position (2,-3).

41. Why might the size of a structure differ from the sum of its individual member sizes? Provide an example structure where this occurs.

42. Write a program that allocates a 2D array (using an array of pointers) to form an identity matrix, prints the matrix using a helper function `printArray`, and then frees all allocated memory.

43. Write a program that implements dynamic array growth using functions `push` (to append an element) and `pop` (to remove the last element). The array should double in capacity when full. Demonstrate reading integers from input and printing the resulting array.

PRACTICE PROBLEMS FROM SLIDES

1. Write a program that takes in two command line arguments that represent integers. Your program should multiply those two numbers together and print out the result. Remember your command line arguments are provided as strings!

2. Update the word count program we wrote earlier to mimic support of the -l, -w, and -c flags that the built-in wc command provides.

3. Write a function void primes(int n) that takes as a parameter one int and prints out (in order) all of the prime numbers smaller than that int.

4. Write a function int str_to_int(const char *s) that takes in a string that is a numeric only string and returns the integer that string represents.

5. Write a function digits that takes in an unsigned int and returns a pointer to an array that stores the digits of the int in order from most significant to least significant. For example digits(257) would return an array that contained 2, 5, and 7 in that order. digits(72519) would return an array that contained 7, 2, 5, 1, and 9 in that order.

6. Write a C program that mimics the behaviour of the wc tool we've used when learning about the shell. For now only worry about the behaviour of wc when it receives no command-line arguments.

7. Write a function argmax that returns the index of the maximal item in an array of integers. What parameters does your function need to have?

8. Write a function replace which takes in an array and is parameterized by two integers tar and repl and replaces every instance of tar in the given array with repl.

9. Write the function str_replace (str_replace replaces every instance of find in string s with the character repl and returns the number of instances replaced) as described below:

```
int str_replace(char *s, char find, char repl);
```

10. Write a function pop that takes in an array of integers and removes the last item from the array. The function should have a void return type and any changes it needs to make to the caller's data should be done through mutation.

11. Implement a Stack ADT using structs:
    Create a struct called Stack that stores an array of integers and any necessary metadata (such as size and capacity)
    Implement the push function: void push(struct Stack* s, int value) that adds a value to the stack
    Implement the pop function: int pop(struct Stack* s) that removes and returns the top value from the stack
    Your implementation should handle basic error cases (such as pushing to a full stack or popping from an empty stack)

12. Header File Implementation:
    You are creating a math library for a large project. Create a proper header file (math_utils.h) that includes:
        A function declaration for calculating the factorial of a number
        A function declaration for calculating the nth Fibonacci number
        A constant MAX_CALCULATION_SIZE
    Implement the corresponding implementation file (math_utils.c)
    Add appropriate header guards to prevent multiple inclusion issues
    Explain which components should be exposed in the header file and which should remain private in the implementation file

13. Separate Compilation
    You have a project with three files:
        geometry.h: Contains declarations for geometric calculations
        geometry.c: Contains implementations of those calculations
        main.c: A program that uses the geometry library
    Write the necessary gcc commands to:
        Compile geometry.c into an object file
        Compile main.c into an object file
        Link the object files into an executable named "geometry_calc"
    Include appropriate forward declarations and header includes in each file
    Implement a Point struct that should be exposed to users and a Distance struct that should remain private to the implementation

14. Write a function print binary that has one unsigned int parameter and prints out the binary representation of that number.

LABS

(Refer to actual labs posted to 201-W25 for more information & extra files)

1. Lab 3 – Bash Script for Test Case Execution
   Write a bash script named produceOutputs that takes two command line arguments: (a) a working executable and (b) a test set file. For each test case stem in the file, do the following:

   If a corresponding .in file exists, run the executable with input redirection.

   If a corresponding .args file exists, run the executable with command-line arguments.

   If both files exist, run the executable using both input and arguments.

   Redirect the output to a file with the same stem and a .out extension.
   Also, display a usage message if not given exactly two parameters and handle file names containing spaces.

2. Lab 5 – Hamming Distance Programs
   Write two C programs:

   decimal_hamming.c: Read two unsigned integers with the same number of digits from standard input, compare them digit by digit, and print their base-10 Hamming distance.

   binary_hamming.c: Read two unsigned integers, convert each into a fixed 32-bit representation, compare them bit by bit, and print their base-2 Hamming distance.
   Explain how you extract and compare the digits and bits.

3. Lab 6 – Bulls and Cows Game
   Create a C program that plays Bulls and Cows. Use a codeword provided via the command line. The program should:

   Allow the player up to six guesses (or stop earlier if the correct guess is made or EOF is encountered).

   For each guess, count and display the number of Bulls (letters in the correct position) and Cows (correct letters in the wrong position without double counting).
   Describe how you count bulls and cows, especially when duplicate letters appear.

4. Lab 7 – String Zipping and Safe String Reading
   Write two C functions:

   zipStrings: Accept two equal-length strings and return a new dynamically allocated string that interlaces their characters (e.g. "abc" and "xyz" produce "axbycz").

readString: Read the next string from standard input into a heap-allocated buffer. If the first non-whitespace character is a double quote, read until the next double quote; otherwise, read until whitespace or EOF.
Describe how you manage dynamic memory and avoid memory leaks.

5. Lab 8 – Minesweeper Board Query
   Develop a C program that reads a Minesweeper board from standard input. The input format is as follows:

   Two integers indicating the grid's width and height.

   A series of width×height characters (either 'X' for a bomb or 'O' for an open space) possibly separated by extra whitespace.

   A sequence of coordinate pairs (x, y) until EOF.
   For each coordinate, print "BOMB!" if the cell has an 'X'; if it has an 'O', count and print the number of bombs in adjacent cells.

   Explain how you parse the grid and check adjacent cells.

6. Lab 9 – File I/O: Integer Sorting and Redaction
   Write two C programs:

   filesort.c: Read a file path from standard input using an array of 256 characters. Open the file and print all integers in the file in ascending order without storing all integers in a large array (re-read the file as needed).

   redact.c: Accept a file name as a command line argument and then read up to 10 strings from standard input. Open the file and replace every occurrence of each given string with a sequence of uppercase 'X's matching the original word's length.
   Describe your approach to sorting without large arrays and editing the file in place.

ASSIGNMENTS

(Refer to actual labs posted to 201-W25 for more information & extra files)

1. Assignment 1 – Repository Setup and Bash Test Scripts
   Your task is to build several bash scripts to support testing of your code, all while using a pre‑provided course repository. In detail, address the following:

   a. Repository Clone and File Verification:
      • Clone the course Git repository from the provided URL (https://github.com/rsh-ua/201-W25.git).

• After cloning, locate the file hello.txt within the “a1” directory and include it as one of your deliverables.

b. Script: testDescribe:

• Write a bash script called testDescribe that takes exactly one command line argument—a filepath to a “test set file.” This file contains a series of file stems separated by whitespace (for example, “test1 /home/rob/foo/test2 ./test3 test4”).

• For each file stem in the test set file, your script must:

Print a usage message to stderr if no argument is supplied.

Check whether a file named with the stem plus a “.desc” extension exists.

If it exists, output the full contents of that file; if not, print a message in the format: stem: No test description.

• Make sure your script correctly handles both relative and absolute paths.

c. Script: runInTests:

• Develop a script named runInTests that expects two command line arguments: a command to run and a test set file (with file stems as above).

• For each file stem in the test set file, execute the following:

Run the provided command while redirecting its input from the file named “stem.in”.

Compare the command’s output against the content of the corresponding “stem.out” file, using a tool like diff (checking the exit status is recommended).

If the outputs match, print “Test stem passed”. Otherwise, print “Test stem failed” and follow on subsequent lines with “Expected output:” plus the content of stem.out and “Actual output:” plus the output generated by your test run.

• Any temporary files you create must be removed prior to the script’s termination.

d. Script: runTests (Updated):

• Modify your earlier solution for runInTests to produce a script called runTests. In this version, when running each test case:

Redirect input from “stem.in” and, additionally, pass any command line arguments provided in “stem.args” (if that file exists) to the executable.

Be careful to read the contents of the .args file and incorporate them into the command line dynamically without hardcoding them.

(Extra exercise, not for marks) Optionally, improve the script so it only passes input or arguments if the corresponding file exists, and prints a meaningful error if the expected "stem.out" file is missing. Also consider displaying the description from testDescribe along with each test's output.

2. Assignment 2 – Rotations, Credit Card Check, and Roman Numerals
   This assignment consists of three separate C programming tasks. Make sure your programs compile with the flags gcc -Wall -Wvla -Werror and work against the provided sample executables.

   a. Rotating Numbers (rotate.c):
      • Write a program that reads a single integer from standard input.
      • The program must print every "rotation" of that integer on a new line. A single rotation is defined by taking the least significant digit of the current integer and moving it to the front of the number (for example, rotating 5347 produces 7534, then 4753, then 3475).
      • You may assume that every rotation will be representable as an integer.

   b. Credit Card Verification (luhns.c):
      • Implement a program that reads digits one by one from standard input until encountering the first non-digit character (this forms the account number).
      • Apply Luhn's algorithm with the following procedure:

         The last digit is the check digit.

         Moving left from the check digit, double every alternate digit; if doubling produces a number ≥ 10, subtract 9 from the product.

         Sum all these processed digits.

         Multiply the sum by 9, and the remainder modulo 10 must equal the check digit.
         • Print "Valid" if the verification passes, or "Invalid" otherwise. • Important constraint: Do not use arrays or pointers anywhere in this program. The challenge is to perform the computation without storing the digits in an array.

   c. Roman Numerals (numerals.c):
      • Write a program that reads one Roman numeral (ignoring any leading whitespace) from standard input and prints its equivalent integer value (the Arabic numeral).
      • Your program must handle the simplified Roman numeral system with the symbols I, V, X, L, C, D, and M representing 1, 5, 10, 50, 100, 500, and 1000 respectively.
      • Implement the rules where a numeral that follows a larger or equal numeral is added, and if a numeral precedes a larger one, its value is subtracted (for example, IV equals 4, XLI equals 41, and XLIV equals 44).

3. Assignment 3 – Integer Sets, Linear Sequences, and Image Translation
   This assignment involves three programs that each work with dynamic data and require careful memory management (use the doubling strategy for dynamic arrays when applicable).

   a. Integer Sets (int_set.c):
      • Implement a program that maintains two dynamic sets of integers referred to as x and y.
      • The program should continually read and execute commands until receiving ‘q’. The supported commands are:

         a <targ> <int>: Add the integer to the specified set (x or y), ensuring no duplicates.

         r <targ> <int>: Remove the integer from the specified set.

         p <targ>: Print all integers in the specified set in increasing order, space-separated (print nothing if the set is empty).

         u: Calculate and print the union of the two sets in increasing order.

         i: Calculate and print the intersection of the two sets in increasing order. • The input may have arbitrary amounts of whitespace between commands and their arguments.

   b. Linear Sequences (sequences.c):
      • Write a program that expects exactly one command line argument: the initial value of a sequence.
      • Then, read from standard input a series of commands until EOF. These commands will either be:

         An arithmetic operation in the form of an operator string (add, sub, mul, or div) followed by an integer operand.

         The string n which signals that the program should compute and print the next number in the sequence.
         • Each time an operation command is received, append that operation to a sequence of operations. When n is read, apply the stored operations (in the order they were read) to the current number to compute and output the next number; update the current number with this computed value.

   c. Image Translation (transformer.c):
      • Develop a program that reads in an image file in “Plain or Raw PPM” format from

standard input. The PPM file format is structured as:

First line: “P3”

Second line: two integers representing the image’s width and height

Third line: the maximum color component value (assume always 255)

Following lines: rows of pixel data, with each pixel represented as three integers (R, G, and B values).
• The program must optionally accept command line arguments:

-f to flip (mirror) the image horizontally (i.e. reverse the order of pixels in each row).

-s to apply a sepia filter to each pixel. The sepia transformation is defined by:
• newR = min(255, R * 0.393 + G * 0.769 + B * 0.189)
• newG = min(255, R * 0.349 + G * 0.686 + B * 0.168)
• newB = min(255, R * 0.272 + G * 0.534 + B * 0.131)

(Extra exercise: Optionally implement a -r flag to rotate the image 90 degrees.)
• After applying the specified transformations (if any), output the resulting PPM image to standard output.

4. Assignment 4 – Maze Game ADT
   In this assignment, you will implement an Abstract Data Type (ADT) for a maze. Note that you are provided with the files main.c and maze.h, which must not be modified. Your task is to implement all required functions in maze.c.

   a. Maze Representation:
      • Design the Maze ADT to represent a two-dimensional grid of tiles. Tiles are represented by specific characters:

      Start Tile ('S'): Indicates the player’s starting position (exactly one per maze).

      Goal Tile ('G'): One or more of these indicate exit points.

      Open Tile ('O'): Traversable space.

      Wall Tile ('X'): Impassable obstacles.

Teleporter Tiles (digits '0'–'9'): Each digit appears exactly twice in the maze; stepping on one teleports the player to the matching digit tile.

Icy Tile ('I'): Traversable but causes the player to continue sliding in the current direction until a boundary, wall, or non-icy tile is reached.

b. Required Functions in maze.c:
• *struct Maze readMaze():

Read the maze from standard input line by line until a completely blank line is encountered. Each non-empty line represents one row of the maze grid.

Validate that there is exactly one start tile and at least one goal tile. If not, return NULL; otherwise, allocate and initialize your Maze structure. • *struct Pos makeMove(struct Maze , char):

Accepts a Maze pointer and a direction command ('n', 'e', 's', or 'w').

Move the player according to maze rules, handling special behavior for teleporter and icy tiles.

Update the Maze to reflect the new player position and return a Pos structure with the new coordinates. If the move results in landing on a goal tile, return a Pos with both x and y set to -1. • *void reset(struct Maze):

Reset the Maze such that the player's position is restored to the starting tile. • *void printMaze(struct Maze):

Print the maze with a border (top and bottom enclosed by '=' and sides by '|').

Do not print the underlying tile at the player's location; instead, print a 'P' to indicate the player. • **struct Maze destroyMaze(struct Maze):

Free all memory associated with the Maze and return NULL.

c. Additional Requirements:
• Define your own Maze structure in maze.c to support all necessary operations.
• Compile your implementation with the provided main.c (using a command like gcc main.c maze.c -o myMaze) to verify that the ADT functions correctly.

MIDTERM 1 & 2

Unfortunately because of Academic integrity policies I can't share the midterm 1 and 2 questions, but look at the 201 eClass under Week 6 and Week 10 for Midterm 1 and 2 and review them. It is likely that the final exam will feature similar formats and questions.

Extra Notes and Questions

1. C Language Fundamentals

    Compilation Process
    - C code goes through preprocessing, compilation, assembling, and linking.
    - Use gcc with flags like –Wall and –Werror to catch errors early.

    Static vs. Dynamic Typing
    - Variables must be declared with types that never change.
    - The main() function is the fixed entry point.

    Preprocessor Directives & Header Guards
    - Directives (like #include and #define) are processed before compilation.
    - Header guards (using #ifndef, #define, #endif) prevent multiple inclusions.

2. Pointers, Arrays, and Memory

    Pointers vs. Arrays
    - A pointer stores a memory address; arrays provide contiguous storage.
    - When passed to functions, arrays "decay" into pointers.

    Pointer Arithmetic
    - Incrementing a pointer moves it by the size of its data type.
    - Useful for iterating over arrays without indexing.

    Dynamic Memory Allocation
    - Use malloc/free for heap memory; check for NULL returns from malloc.
    - When growing arrays, a doubling strategy minimizes reallocation time.
    - Always set freed pointers to NULL to avoid dangling references.

3. Structures and Aggregate Data

Defining and Accessing Structs

- Structs group related variables. Use the dot (.) for variables and arrow (→) for pointers.
- Large structs are best passed by pointer to avoid copying overhead.

ADT Design and Header Files

- Expose only the interface (function prototypes) in header files while keeping implementation details hidden.
- Separate compilation allows modular development using custom header files with guards.

4. Command-Line Arguments and Multi-dimensional Arrays

Using argc/argv

- main() receives command-line arguments as the argument count and an array of strings.
- argv[0] is the program name; the rest are passed parameters.

Dynamic 2D Array Allocation

- Two methods: allocate an array of pointers (each row separate) or allocate a contiguous block and simulate 2D indexing.
- Consider cache locality when performance is critical.

5. Shell and Bash Scripting Basics

Command Redirection and Pipes

- Use > for stdout, 2> for stderr, and < for stdin.
- The pipe operator (|) channels output from one command to the input of another.

Variables, Subshells, and PATH

- Variables are strings; use $ or ${} for expansion.
- Subshells (using parentheses) allow you to capture command output.
- The PATH variable determines where executables are found; modify it to run scripts from anywhere.

1. Header Guards and Modular Design

Explain the purpose of header guards in your own words and provide a short code example that uses them in a custom header file.

2. Pointer Arithmetic Scenario

Consider an integer array and a pointer initialized to its first element. Describe what happens (in terms of memory addresses) when you increment the pointer by 3. Provide a brief example.

3. Dynamic Array Growth

   Write a function in C that takes a pointer to an integer array, its current length, and capacity. Show how to double the array's capacity, copy over the contents, and free the old memory. Explain why doubling is preferred over incremental growth.

4. Handling Command‑Line Input

   Compare two approaches to processing file input in a C program: one using command‑line arguments to pass file names and the other using input redirection. Describe a scenario where one might be more appropriate than the other.

5. Struct vs. Array Passing

   Describe the difference between passing a struct by value and passing a pointer to the struct when calling a function. Why might you choose one method over the other, especially when dealing with large aggregates?

6. Bash Scripting – Subshells and Pipes

   Provide a small bash script snippet that uses a subshell to capture the output of a command and then pipes that output into another command. Briefly explain each step.

7. Memory Debugging

   Explain what a dangling pointer is and outline two different strategies to prevent them. Include how you would use a debugging tool to identify related issues.

8. Multi‑dimensional Array Allocation Pitfalls

   Discuss two key performance or memory management considerations when allocating a 2D array with an array-of-pointers versus a contiguous 1D block. Explain how cache locality might influence your choice.

9. Function Pointers (Advanced)

   Although not covered extensively in your notes, outline how a function pointer is declared and used in C. Write a small example that stores a function’s address and then calls the function through the pointer.

10. Real-World Debugging Challenge

    Imagine a C program that sporadically crashes due to a memory error. What steps would you take to debug this issue? List the tools and coding practices you would employ to locate and fix the problem.