Some questions may be difficult or tricky. If you can answer these, the exam will be a breeze. Expect Open ended questions like this and some file in the blank. The second half will be programming questions (i.e., you write code). Some questions will be easier than what is below.

01. What is the scope of a variable?

The scope is the block of which a variable is "seen". This means that you can read or set the value of it in a particular block of code.

Textual Example: Variable Z may be available in block C which is embedded in block B, which is embedded in block A, but variable Z is not visible in block B. We have variable Y in block B. We have variable X in block A. In block C, variables X, Y, and Z are visible. In block B, variables X and Y are visible. In block C, variable Z is visible.

For example:

(assume we are in `main`)

```
int a = 10;
/// `a` IS IN SCOPE
if (a == 10) {
     int b = 20;
     /// `a`, `b` ARE IN SCOPE
     {
          int c = 30;
          /// `a`, `b`, `c` ARE IN SCOPE
          /// CHANGES TO `a`, `b` WILL BE PROPRAGATED TO BELOW
LINES.
     }
     /// `a`, `b` ARE IN SCOPE // Note: `c` is now out of scope
}
/// `a` IS IN SCOPE // Note: We no longer see `b` because it is
out of scope.
```

For detailed info, visit (C) https://learn.microsoft.com/en-us/cpp/c-language/lifetime-scope-visibility-and-linkage?view=msvc-170. (C++) https://learn.microsoft.com/en-us/cpp/cpp/scope-visual-cpp?view=msvc-170.

02. What is the purpose of a function? What makes up a function?

A function is used to organize, allow for code reuse (i.e., a repetitive process), and for portability. (There are many applications, if you have questions, reach out.)

A function is composed of a return type, name, parameters, and a body. We have a prototype and a declaration. The prototype declares the return type, name, and parameter types (optionally, declare names). The declaration has the return type, name, parameter types with names (a type must be declared with every comma), and a body.

If the function's return type is not void, that explicit type must be returned. Failure to do so may result in undefined run-time behaviors. Be sure that all parameters have a name and a type in the declaration, separating multiple arguments by commas. If you declare names for parameters in the prototype, the same name should be reflected in your declaration (no error or undefined behavior, but good coding practice).

03. Do we want to comment our programs? Why/why not?

We want to comment our code enough that it helps the reader understand what is happening. If someone who has never coded before can look at your code and figure it out, then there are excellent comments. They should be concise, clear, and relevant.

In-line comments are not as necessary in real-world applications unless it is very complicated and there is no easier way. We also like comments to provide credit and create traceability of code modifications. For example, if some code breaks after Joe modifies a thought of unrelated code, then the programmer who is patching the bug can reach out to Joe and ask what he did and come up with a better solution than hoping his fix wont then make Joe's patch ineffective. Comments are a means of communication.

04. Is a datatype the same as a function? Why/why not?

A datatype is a TYPE that is returned in a function, but it is NOT a function. A function is a named block of code. A type can be declared within a method, struct, union, class (C++), or globally (outside of any method in a file).

05. In C, what is a stream? Why do we want to use streams?

A stream is a flow of data. In C, we interact with streams with the FILE* type. This is referred to as a file pointer. The stream is the actual part which provides the in-between of a file and your program. These streams are unidirectional. We can either read or write. (Technically we can create bidirectional streams, but focus on unidirectional for now)

Examples of streams: keyboard (I), mouse (I), printer (O), network (I/O), modify files (I/O), etc.

See https://www.gnu.org/software/libc/manual/html_node/I_002fO-on-Streams.html for more info.

06. How do we control the flow of our program using logic?

We use an: if, switch, for, for-each, while, do-while statement.

If, if-else, if-elif-...-else: This is when we want to run the segment ONCE. Executing runs from line 0 to N, it will never jump back to line X using an if statement. We can add as many else if statements as we want; at a certain point, we should use a switch statement. We can add an else clause which is a catch all for any condition that was not true before. *Note: an else if or else clause must be connecting to a previous if or else if clause.*

Switch: a statement that allows you to use `case` statements. Use a default statement which is the same as an else in an if statement. Always terminate the end of a `case` with `break;`.

For, for-each: Iterate over a block of code for a finite number of times. For each will iterate through an array (this is more relevant to C++, but there is a version in C).

While: Iterate over something a finite (or improperly an infinite) number of times. There is a condition that can be changed and is less practical than a for loop for counting from 0 to N. More practical for advanced loop conditions.

Do-While: Run the block of code once then check the condition. After first iteration, same was while loop.

07. Can we compare a character type with an integer directly? Why/why not?

Technically yes, they are both of a numerical type so it will work. We should not, though, because we should only compare types that are the same. It is good practice to always use the same types with each other and explicitly cast as needed.

08. What type of arguments do we have in a function? What is each type?

Actual: The value we pass into a function.

Formal: The variable we declare to use the actual argument within the function.

09. Should we use floats or doubles more?

We like doubles more since they have higher precision (8 bytes minimum) whereas floats only have 4 bytes minimum. This means that floats have half the precision, and it will cause data loss if we cast a double to a float. (Since we lose data with a single type, it shows that it is not ideal for large numbers or for numbers with fractional parts.

10. Is 'a' or 'Z' larger? How do you know?

'a' is larger. We know from the ASCII table.

'a' > 'Z' === 97 > 90

11. What types of errors do the compiler catch?

Syntax errors. It will catch errors such as `nit` where we meant `int`, but it will not catch errors such as `if (x > 100)` whereas we wanted `if (x < 100`. It will also catch incorrect assignments (mismatching types) and other syntactical errors. These are also referred to as static errors.

12. What types of errors does your IDE catch?

Your IDE itself does not catch any errors; it is an environment. It is composed of a lot of different software that come together and each have their own part in diagnosing errors. Sometimes it can identify logic errors and suggest other improvements that the compiler would otherwise not be able to identify. They also include debuggers to help find bugs at runtime.

13. When are logic errors caught?

Logic errors are only caught by the developer, you. It is typically discovered when we run the program and get unexpected results. It may cause the program to crash, act erratically, or otherwise not perform or do what we want.

14. When will you know if you dereference NULL?

(If you could not answer this, it is ok.) If we dereference NULL, then the program will crash. There is no recovery from it once we attempt to read at 0x0. We would ideally be able to catch this when we write our code, but it is likely this detail will slip. Sometimes our IDE can logically analyze the code and state when we will encounter this error, but is should not be relied on. We need to check if the variable is non-null before using.

15. Is it better to use fscanf or scanf?

We should use fscanf when working with files and scanf when working with user input. We can however use fscanf to read from the user. It is not possible to read from a file using scanf.

16. What is the purpose of main()?

a. What is/are the return type(s)? List all possible outcomes.

int

b. What is/are the parameter(s)? List all possible outcomes.

(1) (void) (2) (int argc, char *argv[])

c. Do we need main() in every C program?

Yes, absolutely. If there is no entry point, then we cannot run anything.

17. How does `FILE * infile` differ from `FILE infile`? Is one better over the other?

FILE * is a type we work with. We will never work with FILE since fopen, fscanf, fprintf, and fclose all work with a FILE *. This means that we work with points without fully knowing what the pointer is. This is a difficult questions since we never will work with FILE, only FILE *

18. What is the star operator? When do we use it?*

It indicates that we have a pointer. We use it to declare a variable an we use it to deference a variable. Dereferencing a variable means that we see the value at a particular memory location. We want to use it when working with large data types or when we need to save space in memory (such as recursive calls). A recursive call is when a function calls itself one or more times. See below for an ex. Do not get stuck on this concept; we will go over it soon, but it should not be on this exam other than FILE *.

```
int * intPtr; // int pointer
... (some code here)
*intPtr = 100; // dereference intPtr and set it to 100.
```

19. What are define statements?

We use define statements to declare a macro whose name is replaced by its value at compile time.

20. What are include statements?

This is a statement that allows us to use functions, variables, datatypes, and other info from a separate file.

21. What all do we need to include for a function prototype?

RETURN_TYPE NAME(P_1_TYPE P_1_NAME, ..., P_N_TYPE P_N_NAME);

If we have no parameters, then we want to use void in place of the p_n_types and names.

* Indicates complete comprehension is not required for Exam 1. We only need to understand what it means for FILE*.

What is wrong with this code?

MAIN.C:

```
#include "stdio.h" #include <stdio.h> This will technically work, but system headers should always be in <>, not "".

#include <myHeader.h> #include "myHeader.h" This will not work since it will be looking in the system headers and myHeader.h is defined locally.

int main(void) {

     char input = getCharFromUser(); We need to store this value somewhere.
     int valid = validateChar(input); We need to store this value somewhere and pass in the value from before. Bool is also acceptable, but we must include stdbool.h

     if (valid == 1) { {} are not required for a single line. Since there are two printf statements, we must use curly braces.
          printf("SUCCESS! Good Entry!\n");
          printf("valid = %lfd", valid); We need %d for an int and it is valid for bool if you go that route.
     } else (valid == 0)
          printf("FAILURE! Bad Entry!\nvalid = %lfd", valid); We need %d for an int and it is valid for bool if you go that route.

}
```

Snippet from MYHEADER.H:
(Assume this file is proper)

```
char getCharFromUser(void);

int validateChar(char);
```

Scoping

Call: myMethod(10, 20, 30);

```
static int d = 40;

void myMethod(int a, int b, int c) {
      // POSITION 1
      d = 4444;
      if (a + b == 12) {
         int a = 1;
         // POSITION 2
      } else if (c - b == 23) {
         int c = 3;
         // POSITION 3
         {
              c = 33;
              b = 2;
              // POSITION 4
         }
      } else {
         int d = 4;
         {
              d = 44;
                   {
                        extern int d;
                        d = 444; // This references the
global variable per `extern int d;`
                        // POSITION 5
                   }
              // POSITION 6
         }
         // POSITION 7
      }
      // POSITION 8
}
```

(Questions on next page)

What is the value of a, b, c, and d at the given positions?

POSITION 1) a = **10** b = **20** c = **30** d = **40**
POSITION 2) a = **1** b = **20** c = **30** d = **4444**
POSITION 3) a = **10** b = **20** c = **3** d = **4444**
POSITION 4) a = **10** b = **2** c = **33** d = **4444**
POSITION 5) a = **10** b = **20** c = **30** d = **444**
POSITION 6) a = **10** b = **20** c = **30** d = **44**
POSITION 7) a = **10** b = **20** c = **30** d = **4**
POSITION 8) a = **10** b = **20** c = **30** d = **444**

```
1: 10 20 30 40
2: 1 20 30 4444
3: 10 20 3 4444
4: 10 2 33 4444
5: 10 20 30 444
6: 10 20 30 44
7: 10 20 30 44
8: 10 20 30 444
```

Write a program that:

1) Opens a file called "input.csv" for reading.
2) Opens a file called "output.log" for writing.
3) Read the first 20 characters [assume all characters are lowercase] $CHAR is the character read.
    a. If it is a vowel (a, e, i, o, u), then print "$CHAR is a vowel" to output.log.
    b. Otherwise, print "$CHAR is NOT a vowel" to output.log.
    c. Do not write 20 different statements to read and output.
4) Clean up variables as needed.
5) Exit with code 0 if clean, exit with code 1 if there was a problem.

- Declare any functions directly above main(). [AKA no protocols are needed]
- No comments needed, only code
- Assume stdio.h is included. Do not use any other header files.

```
int isVowel(char c) {
     return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c ==
'u';
}

int main(void) {
     int exitCode = 0;
     FILE * input = fopen("input.csv", "r");
     FILE * output = fopen("output.log", "r");

     if (input && output) {
          /* Option 1 (Easy) */
          for (int i = 0; i < 20; i += 1) {
               char currentC = (char)fgetc(input);
               if (isVowel(currentC)) {
                    fprintf(output, "%c is a vowel\n", currentC);
               } else {
                    fprintf(output, "%c is NOT a vowel\n",
currentC);
               }
          }

          /* Option 2 (Advanced) */
          int count = 0;
          while (count < 20 && feof(input)) {
               char currentC = (char)fgetc(input);
               if (isVowel(currentC)) {
                    fprintf(output, "%c is a vowel\n", currentC);
               } else {
                    fprintf(output, "%c is NOT a vowel\n",
currentC);
               }
          }

          /* Option 3 (Advanced) */
          char list[20];
          fgets(list, 20, input);
          for (int index = 0; index < 20; index += 1) {
               if (isVowel(list[index])) {
                    fprintf(output, "%c is a vowel\n",
list[index]);
               } else {
                    fprintf(output, "%c is NOT a vowel\n",
list[index]);
               }

          }
```

```
            /* All Options: */
            fclose(input);
            fclose(output);
        } else {
            exitCode = 1;
        }

        return exitCode;
}
```

Write a function that checks if a number is an Armstrong Number.
That is, $153 = 1^3 + 5^3 + 3^3$. Each digit of the number cubed is the given number.

- Use scanf to get an input, In this example 153.
- Then, go digit by digit and sum each cube to determine if it is an Armstrong Number.

```
int main(void) {
     int input;

     // Define variables as needed

     scanf("%d", &input);

     // Implement

     // Print if it is an Armstrong Number or not

}
```

```
int main(void) {
     int input;

     scanf("%d", &input);

     double sum = input;
     int backup = input;

     while (input != 0) {
          sum -= pow(input % 10, 3);
          input %= 10;
     }

     if (sum == 0) {
          printf("%d is an armstrong number", backup);
     } else {
          printf("%d is NOT an armstrong number", backup);
     }

     return 0;
}
```

Write a function that checks if a number is a Perfect Number. e.g., N = 28; N's divisors are 1, 2, 4, 7, 14. 28 = 1 + 2 + 4 + 7 + 14. All divisors of the input sums to that input.

- Use scanf to get an input, in this example 28.
- Then, check every number from 1 to N if it is a divisor.
    - A divisor can be determined by (N % i == 0) where i varies from 1 to the N.

```
int main(void) {
     int input;

     // Define variables as needed

     scanf("%d", &input);

     // Implement

     // Print if it is a Perfect Number or not

}
```

```
int main(void) {
     int input;

     scanf("%d", &input);

     int sum = 0;

     for (int i = 1; i < input; i += 1) {
          if (i % input == 0) {
               sum += i;
          }
     }

     if (sum == input) {
          printf("%d is a perfect number", input);
     } else {
          printf("%d is NOT a perfect number", input);
     }

     return 0;
}
```