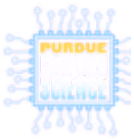
CS 240 - Programming in C

Spring 2025



CS 240 Example Exam 1 Questions

Revised: Monday, February 24, 2025

NOTE: You MAY see some of these questions on the exam. YOU WILL ALSO SEE QUESTIONS ON THE EXAM THAT ARE NOT LISTED HERE.

```
Compilation and Linking:
========================
 - Write the command to compile a single C file named "hello.c" into an object
   file called "hello.o".


 - Write the command to link two object files named "hello.o" and "goodbye.o"
   into the executable called "application".


 - Can you "run" an object file if it contains the "main()" function?


 - Can you "run" an executable that contains a single function called "main()"?


 - Can you "run" an executable that does not contain a function called "main()"


 - What does the "-Wall" flag do?


 - What does the "-g" flag do?


 - What does the "-ansi" flag do?


 - What does the "-c" flag do?


 - What does the "-o" flag do?

File Operations:
```

==========

- Given the following FILE pointer variable definition, write the code
  that will open a file named "hello.txt" for read-only access and print
  a message of your choice if there was an error in doing so.

```
FILE *my_file = 0;
```

- Write code that will, without opening any file, check if a file named
  "hello.txt" can be opened for read access. Put the code inside the
  'if' predicate:

```
if (                                                    ) {
    /* Yes, we can open the file... */
}
```

- Write code that will, without opening any file, check if a file named
  "hello.txt" can be opened for write access. Put the code inside the
  'if' predicate:

```
if (                                                    ) {
    /* Yes, we can open the file... */
}
```

- Write a function called read_and_print() that will do the following:
  . Open a text file called "hello.txt" for read-only access.
  . Read a word that is terminated by a newline from the file into the
    character array called "my_string".
  . Read an integer terminated by a newline into the int variable called
    "my_int".
  . Print the string and the integer value.
  . Return the my_int value.
  - If the file cannot be opened for reading, return -1.
  - If an error occurs while reading from the file, return -1.

- Write a function named print_reverse that will open a text file
  named "hello.txt" and print each character in the file in reverse.
  i.e. print the first character last and the last character first.
  The function should return the number of characters in the file.
  Upon any error, return -1.   HINT: Use fseek() a lot to do this.

- Write a function that defines a structure, initializes it, writes it to a file called "struct.out", closes the file, re-opens the file for read-only access, reads a single structure into a new struct variable and then closes the file. Print the structure contents to the screen. On any error, return -1. Otherwise return 0.

Typedef:
========

- Declare a type called "my_array_t" that is an array of 15 floats.

- Declare a type called "struct_arr_t" that is an array of 10 structs of the format

```
struct str {
  int x;
  int y;
};
```

- Define a variable called my_str_arr of type struct_arr_type.

Structures:
===========

- Can two elements within a structure have the same name?

- Can you initialize a structure like this?

```
struct my_str {
  int   x;
  float y;
} mine = { 0, 0.0 };
```

- Can you initialize a structure like this?

```
struct my_str {
  int   x;
  float y;
};
void my_func(int n) {
  my_str mine = { n, 0.0 };
}
```

- Declare a structure that contains an integer element named i, a floating point element named f, and an array of 20 characters named str (in that order). Name it anything you want.

- Define a variable called "my_new_struct" of the type in the previous question.

- Define a variable called "my_array_of_structs" that is an array of 40 structures of the type in the prior two questions.

- Define a function called bigger_rectangle() that will accept one argument of the structure type rectangle (declared below) and will multiply the width dimension by 1.5, the height dimension by 2.5 and the length dimension by 3. The function should return the new structure. Define a temporary local variable if you want to.

```
struct rectangle {
  float height;
  float width;
  float length;
};
```

- Write a function named sum_rectangles that will open a binary file named "rect.in" for reading and read the binary images of rectangle structures from it. For each rectangle structure, add it's elements to those of the first structure read. e.g. sum the height fields of all the structures, sum the width fields of all the structures, etc... Return a structure from sum_rectangles where each element represents the sum of all structures read from the file. i.e. the height field should be the sum of all of the height fields of each of the structures. On any file error, return the structure { -1.0, -1.0, -1.0 }.

assert()

- Under what circumstances would you place an assert() into your code?

- What will be the result of the following code:

```
int my_func() {
  int count = 0;
  int sum = 0;

  for (count = 0; count < 100; count++) {
    assert(sum > 0);
    sum = sum + count;
  }
  return sum;
}
```

- What might you do the previous code to make it do a "better" job?

String Operations:

- Write a function called do_compare() that will prompt the user for two strings of maximum length 100. It should compare them and print one of the following message:

```
    The strings are equal.
    The first string comes before the second.
    The second string comes before the first.
```

  The function should always return zero.

Variables:

- What is the different between initialization of a variable and assignment to a variable.

- What is the difference between a declaration and a definition.

- What is the difference between a global variable and a local variable.

- For the following questions, assume that the size of an 'int' is 4 bytes, the size of a 'char' is one byte, the size of a 'float' is 4 bytes and the size of a 'double' is 8 bytes. Write the size of the following expressions:

```
struct my_coord {
  int x;
  int y;
  double altitude;
};

struct my_line {
  struct my_coord first;
  struct my_coord second;
  char name[10];
};

struct my_coord var;
struct my_coord array[3];
struct my_line one_line;
struct my_line two_lines[2];

sizeof(struct my_coord) = _______

sizeof(var)             = _______

sizeof(array[1])        = _______

sizeof(array[2])        = _______

sizeof(array)           = _______

sizeof(struct my_line)  = _______

sizeof(two_lines)       = _______

sizeof(one_line)        = _______
```

- Draw the memory layout of the prior four variables; var, array, one_line, and two_lines on a line of boxes. Label the start of each variable and clearly show how many bytes each element within each structure variable consumes.

- Re-define the two_lines variable above and _initialize_ it's contents with the following values:

```
first my_line structure:
    first my_coord structure:
       x = 1
       y = 3
       altitude = 5.6
    second my_coord structure:
       x = 4
       y = 5
       altitude = 2.1
    name = "My Town"
second my_line structure:
    first my_coord structure:
       x = 9
       y = 2
       altitude = 1.1
    second my_coord structure:
       x = 3
       y = 3
       altitude = 0.1
    name = "Your Town"
```

- How many bytes large is the following definition?

```
struct my_coord new_array[] = { { 0,0,3.5 }, { 1,2,4.5}, { 2,0,9.5} };
```

Basic Pointer Operations:

- What is printed by the following three pieces of code:

```
int x = 0;                  int x = 0;                  int x = 0;
int y = 0;                  int y = 0;                  int y = 0;
int *p = NULL;              int *p = NULL;              int *p = NULL;
                            int *q = NULL;              int *q = NULL;
p = &x;
*p = 5;                     p = &x;                     p = &y;
p = &y;                     q = p;                      q = &x;
*p = 7;                     *q = 7;                     p = 2;
                                                        q = 3;
printf("%d %d\n", x, y);    printf("%d %d\n", x, y);    printf("%d %d\n'
```

- Consider the following variable definitions:

```
int x = 2;
int arr[10] = {4, 5, 6, 7, 1, 2, 3, 0, 8, 9};
int *p;
```

And assume that p is initialized to point to one of the integers in arr.
Which of the following statements are legitimate? Why or why not?

```
p = arr;      arr = p;      p = &arr[2];   p = arr[x];     p = &arr[x];

arr[x] = p;   arr[p] = x;   &arr[x] = p;   p = &arr;       x = *arr;

x = arr + x;  p = arr + x;  arr = p + x;   x = &(arr+x);   p++;
```

```
  x = --p;      x = *p++;     x = (*p)++;    arr++;          x = p - arr;

  x = (p>arr);  arr[*p]=*p;   *p++ = x;      p = p + 1;      arr = arr + 1;
```

- Given the following definitions:

```
  int arr[] = { 0, 1, 2, 3 };
  int *p = arr;
```

  are the following two statements equivalent?

```
      p = p + 1;                p++;
```

  What can you say about the result of adding a pointer to an integer?

- Write a function called 'swap' that will accept two pointers to integers and will exchange the contents of those integer locations.

  . Show a call to this subroutine to exchange two variables.

  . Why is it necessary to pass pointers to the integers instead of just passing the integers to the Swap subroutine?

  . What would happen if you called swap like this:

```
      int x = 5;
      swap(&x, &x);
```

  . Can you do this: (why or why not?)

```
      swap(&123, &456);
```

- What does the following code print:

```
int func() {
  int array[] = { 4, 2, 9, 3, 8 };
  int *P = NULL;
  int i = 0;

  p = &array[2];
  p++;
  printf("%d\n", *(p++));
  *(--p) = 7;

  (*p)++;
  for (i = 0; i < (sizeof(array)/sizeof(int)); i++) {
    printf("%d ", array[i]);
  }
}
```

- Write a subroutine called clear_it that accepts a pointer to integer and an integer that indicates the size of the space that the pointer points to. clear_it should set all of the elements that the pointer points to to zero.

- Write a subroutine called add_vectors that accepts three pointers to integer and a fourth parameter to indicate what the size of the spaces that the pointers point to. add_vectors should add the elements of the first two 'vectors' together and store them in the third 'vector'.
  e.g. if two arrays of 10 integers, A and B, were to be added together and the result stored in an array C of the same size, the call would look like

```
add_vectors(a, b, c, 10);
```

and, as a result, c[5] would be the sum of a[5] and b[5]