Lab 12: Structs, Pointers, and Memory Allocation
CPSC 1011 – Fall 2018

CLEMSON School of COMPUTING

Introduction

This week's lab involves taking several concepts you've learned over the course of the semester to create a program that helps a user calculate all the perfect numbers between two values. A perfect number is an integer that is equal to the sum of its divisors (except itself).

Lab Objectives

By successfully completing today's lab, you will be able to:

- Implement a C program that processes command-line arguments
- Use typedef to create a structure (e.g., struct) that holds values
- Dynamically allocate memory

Prior to Lab

- This lab corresponds to zyBooks chapter 7, 8.3-8.6, and 15.

Deadline and Collaboration Policy

- This assignment is due by 11:00 PM on Friday (11/30/2018) via Canvas.
    - More instructions on what and how to submit are included at the end of this document.
- You should write your solutions to this lab by yourself. In this lab, **you should not talk about specific code to anyone but a course instructor or lab teaching assistant. If you speak with a TA at the TA help desk, you should document the TA's name in the academic honesty code header.**
- Your zyBooks chapters and lecture slides are available resources you can use to assist you with this lab.

Lab Instructions

Today's lab involves creating a program where a user indicates a range of numbers (e.g., a lower and upper number), and you will write a program to calculate whether each number in between the range is perfect. Recall, a number that divides evenly into a given number is called a *divisor*. For a number 12, we would determine the divisors to be 1, 2, 3, 4, 6, and 12. To determine whether a number is a perfect number, we sum up all the divisors (but exclude the original number itself). Thus, the sum of divisors for 12 would be:

$$1 + 2 + 3 + 4 + 6 = 16$$

The following table gives examples for three different types of numbers: *perfect* (where the sum of divisors excluding the original number itself turns out to be the same as the original number); *abundant* (where if the sum of the divisors is greater than the original number); and *deficient* (when the sum of divisors excluding the original number itself is less than the original number):

Number	Divisors	Sum of Divisors	Type of Number
6	1 2 3	6	perfect
15	1 3 5	9	deficient
20	1 2 4 5 10	22	abundant

Part of this week's lab is to determine a strategy for calculating the sum of divisors for a given value. This may seem challenging, but it's only a few lines of code. You'll be able to spend a few minutes with your neighbor to discuss strategies for calculating this value, and you can write a function to do this for any integer number.

However, how will we get the number? Traditionally, you've used `printf()` and `scanf()` as ways to prompt a user for input and to receive values from the keyboard. While this is certainly a valid way to obtain values, this lab will require you to use command-line arguments. If you haven't covered this in class yet (or have gotten a little rusty), we will give you a refresher. Also, recall that zyBooks chapter 15 covers command line arguments.

When you use command-line arguments, your main contains the following header definition:

```
int main (int argc, char *argv[])
```

The first parameter (`argc`) refers to the number of arguments entered at the command-line and the second parameter (`argv[]`) is an array of pointers that point to the value of the argument entered at the command line.

```
prompt % ./a.out 525 arg2 arg3
```

When we execute the line above, four arguments are passed into the main for `a.out`:

0	`a.out` (the name of the executable)
1	`525`
2	`arg2`
3	`arg3`

Within your code, you can use the `sscanf()` function to get values that are already in memory.

```
int myVal = 0;
sscanf(argv[1], “%d”, &myVal);
```

The above code snippet would take the second value of the command-line arguments (which is an integer) and stores it into an integer value `myVal`. Likewise, we could use the access specifier `%s` to read in a string, but would not need to use the & in `sscanf` because C-style strings are character arrays.

Lab 12 will require you to take in a lower-number and upper-number from command-line arguments. So far, not too bad!

Finally, the last part of the introduction is a refresher on structures and type definitions. You've been introduced to structs in lecture, zyBooks, and in Lab 9, but they are one of the concepts first-semester students tend to struggle with. Thus, we will give you an opportunity to gain more practice using structs within this weeks lab.

When you declare a struct, you are essentially creating a new type, and when you declare a variable of that new type, the computer will reserve memory so that its data members can be assigned values.

```
struct month {           //option 1
     int numberOfDays;
     char name[4];
};

typedef struct {         //option 2 with typedef
     int numberOfDays;
     char name[4];
} month;
```

In the code snippets above for option 1, we declare a definition of a struct. However, there's no memory being allocated at that time---we'd need to *declare* variables of type struct month in order to have memory assigned:

```
struct month lastMonth, currentMonth, nextMonth;
```

In option 2, because we use the typedef keyword, we can declare variables using only the word month:

```
month lastMonth, currentMonth, nextMonth;
```

Recall that you can assign values in several different formats:

```
month previousMonth, currentMonth, nextMonth;  // two "month" variables

// perhaps other code here …

currentMonth.numberOfDays = 30;
strcpy(currentMonth.name, "Nov");    // cannot use '=' when assigning a string
                                     // value unless it is at the same time
                                     // as the declaration

// OR - could use a compound literal in place of the above two lines:
// currentMonth = (month) {30, "Nov"};
```

This leads us to the final topic this week's lab will involve: *dynamic memory allocation*. Most of the programs you've worked on had memory allocated by the computer at compile time. That is, when you go to compile your program, the compiler reserves as much memory is needed for the variables of your program. This works well when you know what variables are ahead of time, but does not work so well for if you depend on the user for some sort of input. For instance, if you wanted to create an array for a user-defined class size to calculate grades, you would need to wait until run-time to obtain the information from the user in order to size the array appropriately.

In this week's lab, you will use either `malloc()` or `calloc()` from the `<stdlib.h>` to allocate memory for a struct that you will create for calculating your perfect divisors. A call to either `malloc()` or `calloc()` will return one of two things:

- NULL, indicating that memory could not be reserved from the system

- A void pointer that can be used to point to any data type (thus, you need to cast to the type of pointer being used).

```
#include <stdio.h>
#include <stdlib.h>
// calloc(), malloc(), and exit() functions come from stdlib.h

typedef struct {         // defining a new data type
 int numberOfDays;
 char name[4];
  } month;


int main(void) {

  month * theMonths;     // a pointer that can be used to point to a "month"
  int howManyMonths;  // the number of months, to be initialized from user input

  printf("How many months do you want?");
  scanf("%d", &howManyMonths);

  theMonths = (month *)calloc(howManyMonths, sizeof(month));  // 2 arguments
  //or
  theMonths = (month *)malloc(howManyMonths * sizeof(month));  // 1 argument
```

In the above example, we are requesting memory for how many months the user specified. If the `calloc()` or `malloc()` are successful, the pointer `theMonths` will be pointer to a block of memory where **<u>you can use array subscript notation to access/set the struct values!:</u>**

```
      theMonths[0].numberOfDays = 31;
```

Assignment

In today's lab, you will write a program that will use dynamic memory allocation and structs to calculate whether a number is *perfect*, *deficient*, or *abundant* as described earlier in the document. The number range will be determined via command-line arguments:

```
Prompt % ./lab12.out  15 19 '|'
  15 is Deficient   |||||||||
  16 is Deficient   |||||||||||||||
  17 is Deficient   |
  18 is Abundant    |||||||||||||||||||||
  19 is Deficient   |
```

Lab Exercise

Create a `lab12.c` file in your Lab 12 directory. Your file should meet the following criteria:

- Function prototype and implementation for a `calculateDivisors` function that take in an integer parameter, calculates the sum of its divisors, and returns that sum (as an integer value)
- A structure that represents what is represented on each line of output, e.g.,
  - Line number
  - Sum of the divisors for that line number
  - Character array containing "Perfect", "Deficient", or "Abundant"
- Pointer declared within the main that will be used to point to an area of memory containing a collection of these structs
- Dynamically allocated memory for the number of structs necessary
- You will assign values to each struct via the pointer
- A loop that goes from X to Y, where X and Y are the numbers inputted by the user at the command line argument
  - Note: because the user is entering input here, you will need to check for reasonableness:
    - The value of X must be greater than or equal to 2
    - The value of Y must be greater than the value of X
    - If either one of the above conditions are not met, you should display an error and re-prompt.
- The user also specifies the character used for the histogram, which is simply the sum of divisors excluding the original number.
- The line number column uses a column width of 4; the status of divisors is in a column of width 10.

You may spend 15 minutes during the Tuesday lab working with your immediate neighbors to think through the logic of how to calculate the sum of divisors for a given integer X. During this phase, you should not be writing any code on the computers, but it's okay to use scratch paper. Don't overthink this problem—think of the basic operators that you already know.

Submission Guidelines

- After you've completed your project, please create a tar.gz submission file that's named: `lastname.tar.gz` that contains your well-commented and documented `lab12.c` file. See Lab 11 for directions on how to do this.

General Grading Rubric

- If you are not present and attending lab on Tuesday and Thursday, you will not receive credit for this lab.
- If you do not check in your code with a TA on Thursday, you will not receive credit for this lab.
- If your assignment is not submitted fully to Canvas by the due date, you will not receive credit for this lab.
- Your assignment will be graded out of 100 points. The approximate grading distribution is:

- Program `lab12.c` runs correctly without errors or warnings 85 points
  - Required function is implemented correctly
  - Command-line arguments are used and operate correctly; incorrect input is detected
  - Dynamic memory allocation is used
  - Column output meets requested size
  - Checks for null pointers
  - Uses structs correctly
  - Correct output
- Proper code formatting, commenting, headers using good programming conventions 15 points