# CpSc 1111 Lab 13
# Command-Line Arguments, Structs, & Dynamic Memory Allocation

## Overview

For this week's lab, you will gain some experience with:

- command-line arguments
- using `sscanf()` to get the values entered at the command-line
- defining and using `structs`
- dynamically allocating memory
- working out the logic (i.e. thinking through an algorithm) for what could be a confusing problem

## Background Information

### Command-Line Arguments – review from lab 12

It is often useful to pass arguments to a program via the command-line (see Makefile). For example,

```
gcc -g -Wall -o lab12_b lab12_b.c
```

passes 6 arguments to the gcc compiler:

```
0     gcc       (the first one is always the name of the executable)
1     -g
2     -Wall
3     -o
4     lab12_b
5     lab12_b.c
```

Remember that the `main()` function header, when using command-line arguments, looks like this:

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

where `argc` contains the number of arguments entered at the command-line (including the name of the executable, which is 6 for the above example) and `argv[]` is the array of pointers, each of which points to the value of the argument that was entered at the command-line. The first item in the `argv[]` array is always a pointer that points to the name of the executable (`gcc` in the above example).

---

### sscanf()

The `sscanf()` function is used to extract something that is already in memory. It is often used to get items from the `argv[]` array when command-line arguments are used. For example, if the second item entered on the command-line was an integer, the following could be used to get that value from `argv[1]` and store it into an integer variable called `num1` (which would have been declared already):

```
sscanf(argv[1], "%d", &num1);  // it automatically converts it to an integer and
                               // stores it in the variable called num1
```

If the third command-line argument was a string, the following could be used to store that value into a character array that was declared called `word`:

```
sscanf(argv[2], "%s", word);   // no '&' needed because 'word' is an array
```

---

**structs**

Recall from lecture that a structure in C is a collection of data members, which could be of different ***types***, that logically belong together. When you define a `struct`, you are essentially creating a new ***type***. When you declare a variable of that new type, that is when memory is reserved, and the data members can be assigned values.

An example of a structure is the following:

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

The above is the definition of a struct, not a declaration (so no memory is being reserved yet). You may declare variables of type `struct month`, such as:

```
struct month currentMonth, nextMonth;   // two "struct month" variables
                                        // memory is reserved here for both
```

or an array of struct months, like this:

```
struct month theMonths[12]; // an array of 12 "struct month" variables
                            // memory is reserved here for the array
                            //    of 12 "struct month"
```

---

**using `typedef` with structs**

If you "typedef" the structure, like this:

```
typedef struct {
    int numberOfDays;
    char name[4];
} month;
```

then you may declare variables using only the word `month` (instead of both words `struct month`):

```
month currentMonth, nextMonth;  // two "month" variables
month theMonths[12];            // an array of 12 "month" variables
```

- - > Either way, once you declare variables of your new data type, you may give values to the data members of your variables, which can be done when it is being declared, or later on in your code.

---

**struct declaration + initialization (using typedef'd struct)**

```
month currentMonth = {30, "Nov"};
month nextMonth = {31, "Dec"};

month theMonths[12] =
    { {31, "Jan"}, {28, "Feb"},
      {31, "Mar"}, {30, "Apr"},
      {31, "May"}, {30, "Jun"},
      {31, "Jul"}, {31, "Aug"},
      {30, "Sep"}, {31, "Oct"},
      {30, "Nov"}, {31, "Dec"} };
```

---

**struct declaration first with initialization later in code (using typedef'd struct)**

```
month currentMonth, nextMonth;  // two "month" variables
month theMonths[12];            // an array of 12 "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"};


nextMonth.numberOfDays = 31;
strcpy(nextMonth.name, "Dec");

// compound literals for each element of the "theMonths" array
theMonths[0] = (month){31, "Jan"};
theMonths[1] = (month){28, "Feb"};
theMonths[2] = (month){31, "Mar"};
theMonths[3] = (month){30, "Apr"};
theMonths[4] = (month){31, "May"};
theMonths[5] = (month){30, "Jun"};
theMonths[6] = (month){31, "Jul"};
theMonths[7] = (month){31, "Aug"};
theMonths[8] = (month){30, "Sep"};
theMonths[9] = (month){31, "Oct"};
theMonths[10] = (month){30, "Nov"};
theMonths[11] = (month){31, "Dec"};
```

---

**Dynamic Memory Allocation**

So far up to this point, when we have declared variables, memory has been reserved at compile time – the compiler knows how much memory to reserve, and space is set aside for those variables:

```
int x;  // compiler "pushes" (reserves) 4 bytes of memory on the stack for "x"
float y;  // compiler reserves 4 bytes (on our system) of memory on the stack
char word[10];  // compiler reserves 10 bytes of memory on the stack
```

There are various reasons for not wanting something to be declared statically, i.e. "pushed" onto the stack at compile time. Sometimes, we do not know how big something will be until the user enters some sort of value; or sometimes the data consists of a large data structure and we do not want it to be "pushed" onto the stack, especially if it is repeatedly being sent to a function (each function call copies it onto the stack).

These are some reasons to dynamically allocate memory for some data structure being used in a program. Dynamically allocated memory:

1. uses a pointer to point to the area of memory to be used,
2. and, the memory being used is called "heap" memory – not "stack" memory (a different area of memory than the stack)

There are a couple of functions to choose from to dynamically allocate memory, both coming from <stdlib.h>:

```
calloc()
malloc()
```

They both return a void pointer, which is a pointer that could be used to point to any data type. Thus, the return type is cast (should be cast) to the type being used.

**Example code snippet using month structure from above:**

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

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

  // prompt user to enter the number of months they want to have memory reserved
  // for - maybe this program can print up a calendar consisting of any number
  // of months, like an 18 month calendar, for example
  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

  // notice a few things:
    // 1. the return type from either function call is cast as a "month *"
    //    (a month pointer)
    // 2. if successful, "theMonths" will be pointing to a block of memory (on
    //    the heap) big enough to hold the number of months needed (i.e. the
    //    value of the variable called "howManyMonths")
    // 3. if unsuccessful, the returned pointer will be NULL, so it's a good idea
    //    to add an if statement to check whether "theMonths" is NULL or not; if
    //    it is, you may want to print out an error message and possibly exit
    //    the program

  if (theMonths == NULL) {
     printf(" *** Unable to allocate memory. *** \n*** Exiting program. ***");
     exit(1);
  }


  // ... rest of program ...
  /*
  if that memory allocation was successful, remember that the pointer called
  theMonths is pointing to a block of memory of that number of structs and that
  you can use array subscript notation, e.g.  theMonths[0].numberOfDays = 31;
  */
```

---

**Sum of the Divisors and Perfect/Abundant/Deficient Numbers**

Remember that a number that divides evenly into a given number is a divisor. Divisors for 15, for example, would be 1, 3, 5, and 15. The sum of the divisors for 15 therefore would be: 1 + 3 + 5 + 15 = 24. For this program, however, you will not be adding in the given number, so the sum of the divisors of 15 *not including 15* would be: 1 + 3 + 5 = 9.

By the way, if the sum of the divisors of a number (not including the number itself) turns out to be the same as the original number, it is said to be perfect. If the sum of the divisors is greater than the original number, it is said to be abundant. If it is less than the original number, it is said to be deficient.

| number | divisors | sum of divisors | |
|---|---|---|---|
| 6 | 1 2 3 | 6 | perfect number |
| 15 | 1 3 5 | 9 | deficient number |
| 20 | 1 2 4 5 10 | 22 | abundant number |

## Lab Assignment

For this week's lab, you will write a program called **main.c** that will print a histogram for a series of numbers beginning with 2, up to a value entered by the user at the command line. Each line of the histogram will consist of at least 1 character, also specified at the command line. The number of characters per line will correspond to the sum of the divisors (not including the number itself) of the value for that line. For example, if the user entered the following at the command-line:

```
./main 6 '$'
```

the following output would be printed to the screen:

```
 2 is Deficient   $
 3 is Deficient   $
 4 is Deficient   $$$
 5 is Deficient   $
 6 is Perfect     $$$$$$
```

If the user entered the following at the command-line:

```
./main 19 '*'
```

the following output would be printed to the screen:

```
 2 is Deficient   *
 3 is Deficient   *
 4 is Deficient   ***
 5 is Deficient   *
 6 is Perfect     ******
 7 is Deficient   *
 8 is Deficient   *******
 9 is Deficient   ****
10 is Deficient   ********
11 is Deficient   *
12 is Abundant    ****************
13 is Deficient   *
14 is Deficient   **********
15 is Deficient   *********
16 is Deficient   ***************
17 is Deficient   *
18 is Abundant    ********************
19 is Deficient   *
```

**The following items are required for your program:**

1. a `sumOfDivisors()` function which will compute and return the sum of the divisors for a number sent to it. The prototype should look like this: `int sumOfDivisors(int aNum);`
2. a structure which will define what is represented by each line of output. It should contain three things:
    - the line number
    - the sum of the divisors for that line number
    - and a character array that will contain the value "Perfect", "Deficient", or "Abundant" for the given number
2. a pointer declared within `main()` that will be used to point to an area of memory containing a collection of these structs
3. the use of either `malloc()` or `calloc()` to dynamically allocate memory for the number of these structs (as indicated by the number from the user with the second value entered at the command-line)
4. a loop that goes from 2 to the number of times (the number entered by the user) where each time through the loop:
    - `sumOfDivisors()` function is called
    - assign (initialize) the three data members for the current line number:
        a. the line number
        b. sum of the divisors value for that line number that was returned from that function call
        c. also, initialize the string for whether it is perfect, deficient, or abundant
5. after that loop in the above step where all the values in memory are filled in (initialized), you'll have another loop to print out each line of output printing the value of each data member for each line number
6. notice the line numbers line up as right justified in a column – put them in a column width of 3. The deficient/perfect/abundant should be in a column of width -10.

The logic may take a little bit of effort to work through, so start with a well thought out algorithm, and code the program incrementally. You should start by printing back out to the user the integer value that was entered on the command-line and the character that was entered. This involves extracting the integer value and the character from the `argv[]` array using `sscanf()` and storing those values into variables that you can then print to the screen (don't forget to comment out or remove those print statements once you are sure you are extracting those values correctly).

Next, work on the function for computing the sum of the divisors, and print that value out to the screen. Then add the loop to print that number of characters to the screen.

Finally, make sure that the values for each structure pointed to by the pointer are assigned correctly and the histograms are printed based on those values stored in memory.

## Turn In Work

1. Before turning in your assignment, make sure you have followed all of the instructions stated in this assignment and any additional instructions given by your lab instructor(s). Always test, test, and retest that your program compiles and runs successfully on our Unix machines before submitting it.
2. Show your TA that you completed the assignment. Then submit your `main.c` file to the handin page: http://handin.cs.clemson.edu. ***Don't forget to always check on the handin page that your submission worked. You can go to your bucket to see what is there.***

## Grading Rubric

For this lab, points will be based on the following:

| | | |
|---|---|---|
| Functionality | 85 | including the following:<br>• has sumOfDivisors() function & uses command-line arguments (5 points)<br>• a struct (10 points)<br>• malloc() or calloc() (5 points)<br>• checks for NULL pointer after call to malloc/calloc (5 points)<br>• initializes the data members of each line number using the pointer to memory (25 points)<br>• prints out the correct output using the pointer to memory (25 points)<br>• has the line numbers in a column width of 3 (5 points)<br>• has the perfect/deficient/abundant in a column with of -10 (5 points) |
| Code formatting | 10 | (including comments before each function) |
| No warnings when compile | 5 | (including forgetting return statement at end of main) |