#### ­CMPS 12M

**Introduction to Data Structures Lab**

**Lab Assignment 4**

The purpose of this lab assignment is to get more practice programming in C, including the character functions in the library ctype.h, and dynamic memory allocation using malloc, calloc, and free.

**The Character Library**

The C standard library ctype.h contains many functions for classifying and handling character data. For historical reasons the arguments to these functions are of type int rather than char. In order to avoid a compiler warning (under gcc –std=c99 –Wall), it is necessary to first cast the char argument as int. For instance, ctype.h contains the function:

int isalnum(int ch);

which returns non-zero (true) if ch is an alphanumeric character (i.e. a letter or a digit), and 0 (false) if ch is any other type of character. The following C program reads any number of strings from the command line and classifies each character as either alphanumeric, or non-alphanumeric. Please name this program alphaNum.c when submitting.

#include<stdio.h>

#include<ctype.h>

#include<stdlib.h>

#include<string.h>

int main(int argc, char\* argv[]){

char ch;

int i, j, count;

if( argc>1 )

{

for(i=1; i<argc; i++){

ch = argv[i][0];

count = j = 0;

while( ch!='\0' ){

if( isalnum((int)ch) )

count++;

ch = argv[i][++j];

}

printf("%s contains %d alphanumeric and ", argv[i], count);

printf("%d non-alphanumeric characters\n", strlen(argv[i])-count);

}

}

return EXIT\_SUCCESS;

}

Note that this program behaves oddly when certain non-alphabetic characters are included on the command line, such as ’&’, ’!’, or ’\*’, since these characters have a special meaning to some unix shells. To see a short description of the other character functions in ctype.h, do the unix command % man ctype.h (% man cytpe on Mac OSX). Consider especially the functions isalnum(), isalpha(), isdigit(), ispunct(), and isspace() which will be needed for this assignment. To view the man pages for these functions, run % man <name of command> (ie., % man isalnum).

**Dynamic Allocation of Memory**

There are two types of memory in C: *stack* memory and *heap* (also called *free-store*) memory. Stack memory is what you get when you declare a local variable of some type in a function definition. Stack memory is allocated when the function is called and is de-allocated when it returns. The memory area associated with a given function call is called a *stack frame* or just a *frame*. A frame includes memory for all local variables, formal parameters, and a pointer to the instruction in the calling function to which control will be transferred after the function returns. The *function call stack* is literally a stack data structure whose elements are (pointers to) these so-called frames. The frame at the top of the stack corresponds to the function currently executing. Each function call pushes a new frame onto the stack, and each return pops a frame off the stack.

Heap memory on the other hand, is not associated with the function call stack and must be explicitly allocated and de-allocated by program instructions. Heap memory is often said to be *dynamically allocated*, which means that the amount of memory to be used can only be determined at run time. Storage in the heap is organized into blocks of contiguous bytes, and each block is designated as either *allocated* or *free*. These blocks are chunks of memory controlled by the functions malloc, calloc, and free, which are defined in the library stdlib.h. (Do man stdlib.h, and man malloc etc. for documentation.) Allocated blocks are reserved for whatever data the programmer wishes to store in them. In C, one creates an allocated block of a given size by calling the malloc function that, if successful, returns a pointer to the first byte of the newly allocated block. To do this, malloc first has to find a free block large enough to handle the request and convert all or part of that free block into an allocated block. Free blocks are those blocks which are not currently allocated. It is important to remember that the code you write should never access the contents of free blocks. Most bytes in free blocks contain meaningless garbage, but some bytes contain critical information about the locations and sizes of the free and allocated blocks. If a program corrupts that information, it may crash in a way that is mysterious and difficult to diagnose. The free function is used to recycle an allocated block that is no longer needed. Function free converts an allocated block back into a free block, and, if possible, merges that free block with one or two neighboring free blocks.

The prototype for malloc is

void\* malloc(size\_t num\_bytes);

The data type size\_t is an alias for either unsigned int or unsigned long int, and is also defined in stdlib.h. Thus malloc's argument is the number of bytes to be allocated. Its return type is void\* which means a *generic pointer*, i.e. a pointer to any type of data. This is necessary since malloc does not know what kind of data is to be stored in the newly allocated block. Function malloc is almost always used with the sizeof operator, which returns the number of bytes needed to store a given data type. For example

int\* p = malloc(sizeof(int));

allocates a block of heap memory sufficient to store one int and sets p to point to that block. It is important to remember that the pointer variable p is a local variable (within some function) and as such, belongs to stack memory. The memory it *points to* is heap memory. Memory is a finite resource on all computers, so it is possible that malloc cannot find a free block of sufficient size. When that happens, malloc returns a NULL pointer to indicate failure. One should always check malloc’s return value for such a failure.

if( p==NULL ){

fprintf(stderr, "malloc failed\n");

exit(EXIT\_FAILURE);

}

Another common way to do this check is via the assert function as follows:

assert( p!=NULL );

Function assert is defined in the library assert.h, and has prototype void assert(int exp). (As before look up assert.h and assert in the Unix man pages.) It writes information to stderr and then aborts program execution if the expression exp evaluates to 0. Otherwise assert does nothing. The output of assert is platform dependent. Compiling with gcc on unix.ucsc.edu and then calling assert on a false expression gives:

<object\_file>: <source\_file>: <function\_name>: Assertion <expression> failed.

Abort

As previously mentioned, heap memory is de-allocated by the free function.

free(p);

Note that all this instruction does is convert the block of heap memory pointed to by p from allocated to free. The local variable p still stores the address of this block, and therefore it is still possible to dereference p and alter the contents of the block. *Never do this!* As previously mentioned, such an operation could lead to runtime errors that are intermittent and very difficult to trace. Instead, after calling free, set the pointer safely to NULL.

free(p);

p = NULL;

Now any attempt to follow the pointer p will result in a segmentation fault, which although it is a runtime error, will happen consistently and can be traced more easily. Another common error occurs when one reassigns a pointer without first freeing the memory it points to. Consider the following instructions.

int\* p;

p = malloc(sizeof(int));

\*p = 6;

p = malloc(sizeof(int));

\*p = 7;

Observe that the address of the block of heap memory storing the value 6 is lost. The address cannot be assigned to another pointer variable. The block cannot be de-allocated and therefore cannot be re-allocated at any future time. The block storing the value 6 is therefore completely lost to the program and can be of no further use. This situation is called a *memory leak*.

As we can see, C allows programmers to do bad things to memory. In Java, all these problems are solved by the advent of garbage collection. The operator new in java is roughly equivalent to malloc in C. When one creates a reference (i.e. a pointer) to some memory via new, Java sets up a separate internal reference to that same memory. The Java runtime system periodically checks all of its references, and if it notices that the program no longer maintains a reference to some allocated memory, it de-allocates that memory. Thus to free memory in Java you do precisely what you should not do in C, just point the reference variable somewhere else. Also it is not possible in Java to alter the contents of a free block, as can be done in C, since memory is not freed until the program no longer contains references to it.

There is one more C memory allocation function of interest called calloc (contiguous allocation). We use this function to allocate arrays on the heap. The instructions

int\* A;

A = calloc(n, sizeof(int));

allocates a block of heap memory sufficient to store an int array of length n. Equivalently one can do

int\* A;

A = malloc(n\*sizeof(int));

The only difference in the two examples above is that calloc() sets its allocated memory to zero, while malloc() does not. Note also that in both examples n can be a variable. Recall that one cannot allocate variable length arrays on the stack. For instance int A[n] is not a valid declaration in C. As with any array, the array name is a pointer to its zeroth element, so that the expressions A==&A[0] and \*A==A[0] always evaluate to true (i.e. non-zero). In the above examples, A is itself a stack variable, while the memory it points to is on the heap. Pointers have a special kind of arithmetic. The expression A+1 is interpreted to be, not the next byte after A, but the next int after A[0], namely A[1]. Thus \*(A+1)==A[1], \*(A+2)==A[2], etc. all evaluate to true. This gives an alternative method for traversing an array, which is illustrated in the next example. When submitting this please name the file cArrays.c.

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main(int argc, char\* argv[]){

int i, n;

int\* A;

/\* check number of arguments on the command line \*/

if( argc<2 ){

printf("Usage: %s positive\_integer\n", argv[0]);

exit(EXIT\_FAILURE);

}

/\* check that the command line argument is an integer \*/

/\* and if so, assign it to n \*/

if( sscanf(argv[1], "%d", &n)<1 || n<1 ){

printf("Usage: %s positive\_integer\n", argv[0]);

exit(EXIT\_FAILURE);

}

/\* allocate an array of n ints on the heap \*/

A = calloc(n, sizeof(int));

/\* initialize the array using the standard subscript notation \*/

for(i=0; i<n; i++) A[i] = 2\*i+2;

/\* process the array using pointer arithmetic \*/

for(i=0; i<n; i++) printf("%d ", \*(A+i));

printf("\n");

free(A);

A = NULL;

return EXIT\_SUCCESS;

}

The preceding program uses an IO function we've not seen before called sscanf, which is defined in stdio.h. This function works exactly like scanf and fscanf described in lab3, except that it reads input from a string rather than stdin or a file. For more details do man sscanf. Also read the man pages for the standard IO function fgets().

**What to turn in**

Write a C program called charType.c that takes two command line arguments giving the input and output file names respectively, then classifies the characters on each line of the input file into the following categories: alphabetic characters (upper or lower case), numeric characters (digits 0-9), punctuation, and white space (space, tab, or newline). Any characters on a given line of the input file that cannot be placed into one of these four categories (such as control or non-printable characters) will be ignored. For convenience, assume that a line has at most 256 characters. You can use the C functions fgets to read the line. (Check out online documentation on fgets.)

Your program will print a report to the output file for each line in the input file giving the number of characters of each type, and the characters themselves. For instance if in is a file containing the four lines:

abc h63 8ur-)(\*&yhq!~ `xbv

JKL\*()#$$%345~!@? ><mnb

afst ey64 YDNC&

hfdjs9\*&^^%$tre":L

then upon doing % ./charType in out, the file out will contain the lines:

line 1 contains:

12 alphabetic characters: abchuryhqxbv

3 numeric characters: 638

8 punctuation characters: -)(\*&!~`

5 whitespace characters:

line 2 contains:

6 alphabetic characters: JKLmnb

3 numeric characters: 345

13 punctuation characters: \*()#$$%~!@?><

2 whitespace characters:

line 3 contains:

10 alphabetic characters: afsteyYDNC

2 numeric characters: 64

1 punctuation character: &

6 whitespace characters:

line 4 contains:

9 alphabetic characters: hfdjstreL

1 numeric character: 9

8 punctuation characters: \*&^^%$":

1 whitespace character:

Notice that in these reports the word "character" is appropriately singular or plural. Your program will contain a function called extract\_chars with the prototype

void extract\_chars(char\* s, char\* a, char\* d, char\* p, char\* w);

that takes the input string s, and copies its characters into the appropriate output character arrays a (alphabetic), d (digits), p (punctuation), or w (whitespace). The output arrays will each be terminated by the null character '\0', making them into valid C strings. Function main will call extract\_chars on array arguments that have been allocated from heap memory using either malloc or calloc. Before your program terminates it will free all allocated heap memory using free. It is suggested that you take the example program alphaNum.c as a starting point for your charType.c program, since much of what you need to do is illustrated there. When your program is complete, test it on various input files, including its own source file. Check your program for memory leaks by using the unix program valgrind. Do

% valgrind --leak-check=full charType infile outfile

to run valgrind on your program. Valgrind does not work correctly on all versions of Mac OSX so it is recommended to do this on the Unix timeshare. Do % valgrind –help to see some of the options to valgrind, and see the man pages for further details. Write a Makefile that creates an executable binary file called charType and includes a clean utility. Also include a target called check in your Makefile which runs valgrind on your executable as above, taking infile to be the source file charType.c itself. Use the Makefile from lab3 as a starting point.

The Lab4 directory should contain: charType.c, Makefile, README. As always start early and ask for help if anything is not clear.

**Grading:**

* (10 points) Everything done correctly
* (7 points) Everything correct, but valgrind reports a memory leak.
* (3 points) Program does not crash, but is incorrect on some inputs.