CPE-2600 **FALL 2024**

Systems Programming

Lab 4 Console Formatted IO

Objectives

- Work with a variety of C language syntax and practices for strings and formatted IO.

Topical Concepts

Command Line Arguments

This program will utilize a couple of simple command line arguments to control its behavior. We saw an example of this very early in the course. In order to utilize command line arguments, your main must accept two arguments (passed from the operating system's command shell). The first argument is an integer that will tell you how many arguments were passed and the second argument will be an array of character pointers (char*) referencing each argument. Of special note, there is always at least one argument and that is the name of the program itself. Refer to the following example.

```c
#include <stdio.h>
#include <string.h>

// note, argv is often char**, what is that?  A double pointer?
int main(int argc, char* argv[])
{
    printf("Number of args: %d\n",argc);

    for(int i=0; i<argc; i++)
    {
        printf("Arg %d is %s\n",i,argv[i]);
    }

    // a particular arg
    if(argc>1)
    {
        if(!strcmp(argv[1],"marco"))
        {
            printf("Polo\n");
        }
    }

    return 0;
}
```

printf

We have seen printf used in a variety of examples. For routine usage, printf is easy enough. Just be sure to match the type and number of format specifiers to the type and number of additional arguments.

A feature of printf that is used a little less frequently is that it can establish fields and print justified within that field. This can be used to create nicely formatted tables. Consider the following example:

```
int main(int argc, char* argv[])
{
    int rows = 5;

    // make sure there is an arg
    if(argc>1)
    {
        rows=atoi(argv[1]);
    }

    printf("----------------------\n");
    for (int i=0; i<rows; i++)
    {
        printf("|%10d|%10x|\n",i,i);
    }
    printf("----------------------\n\n");

    printf("----------------------\n");
    for (int i=0; i<rows; i++)
    {
        printf("|%-10d|%-10x|\n",i,i);
    }
    printf("----------------------\n\n");

    return 0;
}
```

When this is executed, you can see the effect of the field width and justification:

```
rothede@MSOE-PF3VZ7NL:~/CPE2600/lecturecode/LabWeek4$ ./a.out 11
----------------------
|         0|         0|
|         1|         1|
|         2|         2|
|         3|         3|
|         4|         4|
|         5|         5|
|         6|         6|
|         7|         7|
|         8|         8|
|         9|         9|
|        10|         a|
----------------------

----------------------
|0         |0         |
|1         |1         |
|2         |2         |
|3         |3         |
|4         |4         |
|5         |5         |
|6         |6         |
|7         |7         |
|8         |8         |
|9         |9         |
|10        |a         |
----------------------
```

In addition to width and justification, there are a number of other options that can be applied to the format specifier.

Arrays

The basic usage and syntax of arrays was covered in lecture. Basic usage is simple enough. Where it can get tricky is working with strings. Often errors will go unnoticed for some time after a program is written. Also, occasionally you may wish to use an array in a less common capacity. The following example explores some less common usage of strings in character arrays.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Some "global" data arrays
const char *help[] = {"Help text line 1","Help text line 2","More help
text",""};
const char info[][4] = {"NUL","BEL","EOR","FUN","END"}; // why 4?

// note, argv is often char**, what is that?  A double pointer?
int main(int argc, char* argv[])
{
    // print out arrays
    int i = 0;
    while (strlen(help[i]))
    {
        // help is an array of char*...
        printf("Line %d: %s\n",i,help[i]);
        i++;
    }

    // if using 2 indexes, get a character
    printf("Info [%d][%d]: %c\n",1,1,info[1][1]);

    // if using 1 index, you will basically get
    // the address to the first element of that
    // row.  In this case, that is actaully char*
    // and can use it like a string.
    printf("Info %d: %s\n",1,info[1]);

    // OK, I said stay away from sizeof, but I cannot resist
    printf("Sizeof Help: %zu\n",sizeof(help));
    printf("Sizeof Info: %zu\n",sizeof(info));

    // Hmm...
    printf("Address of help[0]: %p\n",&help[0]);
    printf("Address of help[1]: %p\n",&help[1]);
    printf("Address of info[0]: %p\n",&info[0][0]);
    printf("Address of info[1]: %p\n",&info[1][0]);

    return 0;
}
```

The Assignment

Write an application which accept command line arguments to produce a variety of output. The following represents minimal expectations and you may embellish as you wish. Also keep in mind the background information above which provides some hints as to efficient implementation.

- When your program is invoked with no arguments, your program should print to the console a nicely formatted table of all 7-bit ASCII codes and characters (that is ASCII values from 0 to 127). At a minimum it should list each ASCII value in decimal, in hex, and the actual character. For non-printable characters, the customary mnemonics should be shown (i.e. NUL, SOH, STX, etc.)

  The example below is an example of what might be a minimal expectation.

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

  The printable characters can be generated directly from printf with a %c specifier. How about the mnemonics for the non-printable characters? You must store them in a data array that can easily be iterated.

- If your program is invoked with a '-c' argument, the argument after the '-c' will be interpreted as the number of columns the table should be. The example above shows eight columns which might be a suitable default. If the user has requested an unreasonable number of columns, your program can report that. Perhaps a range of 1 to 8 would be reasonable with 8 as the default if the -c option is not provided. What if a user supplies a -c but neglects to include a number as the next argument? This would be an error condition you should catch and handle.

- If your program is provided a single printable character as an argument, it should respond with the ASCII value of that character in decimal, hex, and 8-bit binary* presented to the user in an easy to read format.

  * as discussed in lecture, there is not currently a printf specifier for binary conversions, although it is planned for C23 and is apparently available in newer versions of glibc. In any case, you must write a conversion routine yourself. The routine should accept a number as an argument and convert it to a string holding the binary representation. I would imagine the function will benefit from another argument to set the bit-width so that it can apply the proper number of leading zeros to the resulting string. The implementation details will be left up to you.

- Your program should be validating and verifying proper commands. For example, if the program is run with options '-c 100' it is probably not reasonable. If an invalid command is detected, you have a couple of choices. Some programs will display a terse error message and exit. Some will display a general help message, and some programs will display a brief error message and run with default behavior. The choice of behavior is up to you.

- OPTIONAL – want to spiff up your output? Check out the topic of ANSI Escape Codes (ANSI escape code - Wikipedia). This is not a C feature per see, rather, it is how a terminal may respond to special sequences of characters printed to it. In this case, the non-printable character ESC (ASCII 27) kicks things off. After the terminal receives ESC, it will interpret the following characters in a particular format to control color, the cursor, and other effects. Check out the C example in the Wikipedia article and see if you can make your ASCII table pretty.

Implementation Details

To implement this program, I suggest two source (.c) files and at least one header file.

- main.c (or other suitable name) should contain main(), of course, and perhaps basic code to decipher the command line arguments.
- ainfo.c (or other suitable name) should contain code specific to this assignment – generating the table and single character information, etc.

Although not critical for this application, you must also supply a makefile to build your project. Be sure to also comply with the course coding standards.

Deliverable

You will demo your finished program during lab in Week 5. In addition, you will submit source code to Canvas by the end of Week 5.

Grading will be based on adherence to specifications, functionality, coding efficiency, and adherence to the coding standard.