

COIS 3380: Lab 3

ASCII Code

Use your terminal to connect and login to Loki as you did in lab 0 ,1, and 2. Move to your 3380 directory using the `cd` command. Once you are inside your 3380 directory, create a new directory called “lab3” using the `mkdir` command. Navigate to the newly created “lab3” directory.

We are going to analyse how ASCII works in C. ASCII (American Standard Code for Information Interchange) is a character encoding standard used to represent text. Each ASCII character is assigned a unique numerical code ranging from 0 to 127, which can be represented in various forms such as binary, decimal, or hexadecimal.

Take a look at this ACSII table below:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Each character in C is stored as an integer corresponding to its ASCII code. For example:

- 'A' has an ASCII value of 65.

- 'a' has an ASCII value of 97.
- '0' has an ASCII value of 48.

```
#include <stdio.h>

#define ASCIIMAX 122

int main()
{
    int i;

    for( i=97 ; i<= ASCIIMAX ; i++ ) /*ASCII values ranges from 0-127*/
    {
        printf("ASCII value of character %c = %d\n", i, i);
    }

    return 0;
}
```

In the above example code, when we print out an integer (*i*) using the character placeholder (*%c*) instead of using the integer placeholder (*%d*), **C interprets the integer as its corresponding ASCII character.**

Complete the code in `convertASCII.c`. It currently prints out all the lowercase letters. Modify the code to print out the uppercase letters instead.

HINT: Use the ASCII table to find the difference between lowercase and uppercase ASCII values.

Strings in C

In C, a string is a sequence of characters stored in contiguous memory locations, terminated by a special null character `\0` to indicate the end of the string. Strings are essentially arrays of characters.

A string in C can be declared as follows:

```
char str1[] = "Hello";

char str2[20]; // This is a string that can only hold 20
characters
```

To access a single character in a string in C, you can use array indexing. Since strings in C are stored as arrays of characters, each character can be accessed using its index, starting from 0 for the first character

```
printf("%c", str1[0]) // prints out H
```

The `<string.h>` library provides several functions to manipulate strings such as:

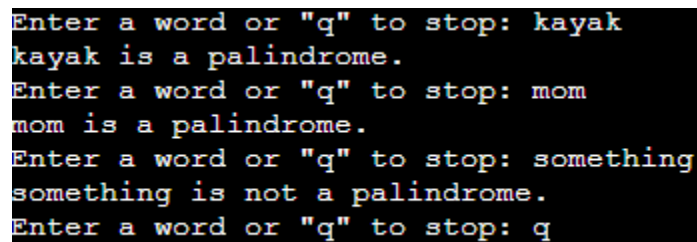
`strlen(str)` : Returns the length of the string.

`strcpy(dest, src)` : Copies a string from `src` to `dest`.

`strcat(dest, src)` : Appends `src` to `dest`.

`strcmp(str1, str2)` : Compares two strings (returns 0 if equal).

A palindrome is a word that reads the same from front and back. Write a program (you can start with `palindrome.c`) to detect if a word is a palindrome or not. Your program should interact with a user as follows:



```
Enter a word or "q" to stop: kayak
kayak is a palindrome.
Enter a word or "q" to stop: mom
mom is a palindrome.
Enter a word or "q" to stop: something
something is not a palindrome.
Enter a word or "q" to stop: q
```

Arrays

An array in C is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store multiple values in a single variable, which is more efficient than declaring separate variables for each value

They can be declared as:

```
int array1[5]; // Option 1
int arr2[5] = {1, 2, 3, 4, 5}; // options 2
int arr3[] = {1, 2, 3}; // Size is inferred as 3.
```

C also supports multi-dimensional arrays.

```
int array2D[3][4] = { {1, 2, 3, 4},
                      {5, 6, 7, 8},
                      {9, 10, 11, 12} };
```

The size of an array is determined by multiplying the size of its data type (e.g., `int` = 4 bytes, `float` = 4 bytes, `double` = 8 bytes, `char` = 1 byte) by the number of elements in the array. For example, an `int array[5]` occupies $5 \times 4 = 20$ bytes. The `sizeof()` operator calculates the size of a data type or an array in memory.

Complete the code `array.c` to find the sum of each row of `array2D`. Add the code in the area shown using `// compute the sum of each row of array2D`

GDB debugger

When compiling your C code, the gcc compiler captures syntax errors, like missing brackets, ‘;’ at the end of statements, and mistakes such as *longe* instead of *long*. It also checks inconsistencies between functions declarations and parameters that are passed to functions. For instance,

```
int swap(int x, int y);

long numX;
long numY;
swap(numX, numY);
```

in the above code, the compiler will display an error message showing a mismatch between the parameters of the function on `swap`, which are declared as `int`, and the type of the variables that were passed to the function, which are declared as `long`.

However, the gcc compiler cannot detect logical errors and errors that may occur during runtime such as division by 0, array out of bounds, and uninitialized variables. To detect such errors, we can use the gdb debugger.

We will use the `average.c` file to test out the gdb debugger. To make file run with the gdb debugger, run the following command while compiling the C code.

```
[username@loki lab3]$ gcc average.c -g -o average
```

The `-g` flag creates a symbol table that allows the executable binary to map back to the source code, including variable names, line numbers, and source file names.

Once you have compiled the code, run the following command:

```
[username@loki lab3]$ gdb average
```

The above command will start the program in compilation mode. You can verify that the code is running using gdb debugger by looking at terminal which should change to `(gdb)` instead of `'[username@loki lab3]$'`

You can start debugging the code by using the `start` command

```
(gdb) start
```

This will start the code for debugging and make the debugger stop at the first line of your code in the `main()`. To continue debugging and move onto the next line, use the `next` command (`n` for short).

```
(gdb) next
```

Keep going to the next line until you reach : `average(x,y) ;`

To look around the code around `average(x,y) ;` use the `list` command (or `l` for short). You can also add a line number after the list command to check the code around a specific line. For instance, `list 34` (or `l 34`) will show you the code lines around line 34.

If you use the next command after reaching `average(x,y) ;`, the debugger will skip the `average()` and move onto the next line. To avoid this and actually see the execution of the `average()` we will use the `step` command

```
(gdb) step
```

This command will take you to the `average()` and allow you to see the parameters and values that are passed to `average()`. Inside this function, keep using the `next` command to reach the for loop on line 65.

The most important use case of using a debugger is to verify the values of the variables. To check the values of all the variables of function `average()`, we can utilize the `info locals` command.

```
(gdb) info locals
```

To view the value of variables `i` and `sum` individually, you can use the `print` command

```
(gdb) print i
```

```
(gdb) print sum
```

However, these values will only be shown once and if they are updated, we have to print the values once again. To avoid repeatedly printing out the value, we can use the `display` command

```
(gdb) display sum
```

This will ensure that the value of the variable will remain and can be seen as it is updated. This display variable can be toggled to be seen or not. To do this, use this command

```
(gdb) info display
```

Use the `Num` value to toggle it. Notice since `sum` is the only variable that is being displayed, it can be disabled by using the following command

```
(gdb) disable display 1
```

It can be enabled once again by using the `enable` command

```
(gdb) enable display 1
```

To completely delete the display variable, use the `delete` command

```
(gdb) delete display 1
```

Using the next command in the for loop, take you back to the same line repeatedly. To avoid this and go outside the loop, use the `finish` command.

Using the `continue` or `'c'` command will terminate the execution of the program.

Breakpoints using GDB

Breakpoints as in any other programming language are commands that instruct the debugger to stop at a particular line of code. We can use `gdb` to set breakpoints and make the debugger stop there. To create a breakpoint, we will use the `break` command.

Set a breakpoint on line 25.

```
(gdb)break 25
```

We will set another breakpoint where the function `average()` is called.

```
(gdb)break average
```

To make the program run and stop at the breakpoints, instead of using `start`, we will use the `run` (`r` for short) command. This will make our debugger stop at the 1st breakpoint on line 25.

We can toggle this breakpoint using the `disable` and `enable` commands. In order to do this, first we need to make sure the which breakpoints are set. We can do this by using the `info breakpoints` command

```
(gdb) info breakpoints
```

This will list all the breakpoints that are set. You will notice that breakpoint 1 is on line 25. To switch off that breakpoint we can use this command

```
(gdb)disable 1
```

And to switch it on back again, we can use this command

```
(gdb) enable 1
```

Switch off that breakpoint on line 25, and use the `run` command. This should make the debugger skip the breakpoint on line 25, and stop at the second breakpoint at the function call of `average()`.

To completely remove a breakpoint, we can utilize the `delete` command

```
(gdb)delete 1
```

The above command should completely remove the breakpoint on line 25. You can verify this by

```
(gdb) info breakpoints
```

Which should not show the breakpoint on line 25 anymore.

Feel free to view some more `gdb` commands on this website:

https://ccrma.stanford.edu/~jos/stkintro/Useful_commands_gdb.html

Demonstration

Please show the lab demonstrators `convertASCII.c`, `palindrome.c`, and `array.c`. Additionally, show the lab demonstrators how you used the `gdb` debugger on `average.c`.