Lecture 03
I/O, Functions, Strings, Arrays

In this lecture

- **Input and output (STDIN and STDOUT)**
- **File I/O**
- **Binary Files**
- **Briefly**
  - **Arrays in C**
  - **Strings in C**
  - **Functions in C**
- **Additional Readings**
- **Exercises**

Input and Output

Input/output(I/O) is fundamental to any programming language. Most programming languages provide built-in I/O routines or provide libraries that can be used to do I/O operations. In C, I/O operations can be performed by using stdio.h. You are encouraged to type at the Unix prompt:

>man stdio.h

To read about this standard buffered input/output routines.

(if you see : press space bar to go to next page. Type Q to quit man pages)

Files are treated as streams of data that can come from terminal or from an external storage device such as tape or disk. Files can be treated as ASCII or can be treated as binary files. All bytes in an ASCII file are expected to be represented using a character in ASCII table and it is possible to edit those files. For example any .txt file is an ASCII file. C compiler provides three standard IO references.

STDIN – refers to Standard input device (typically a keyboard)

STDOUT – defines standard output device (typically a terminal)

STDERR – defines a device where the errors are recorded.

There are two stdio.h functions that can be used for this processing data. fprintf is used for formatted output while fscanf is used for formatted input.

Writing output to STDOUT

Let us look at fprintf first. This is a function with variable number of arguments. Here is an example.

fprintf(STDIN, "This is a test %d %.4f %10.2f %c\n",134,56.455, 3355.5346, 65);

The prototype of fprintf is: int fprintf(FILE* fp, char *S, arg1, arg2, …)

Where S typically called a format string that contains regular characters (eg: "This is ..") and specification of conversion of arguments. For example first specification %d in the

example instructs the compiler to replace %d by the decimal representation of 134, the first argument. The number of format specifications inside the format string must match the number of arguments. Although the program will compile w/o the equal number of matching arguments, the output can be erroneous.
The formatting statements such as %10.2f are used to format your output to 10 total spaces (blanks in front) and 2 decimal places. A list of format characters can be found in Table 7-1 on page 154 on K&R.

If you are interested in outputting the characters to terminal then you can use the function: int putchar(int)
For example if you need to write the character 'c' to terminal then you may write: putchar('c') or putchar(99) where 99 is the ASCII value of character 'c'. putchar returns the ASCII value of the same character or EOF if an error occurs.

Reading input from STDIN
Function scanf can be used to read input from STDIN. For example, if you need to read an integer from STDIN to integer variable x, then you can write; **scanf("%d", &x);**
Note that scanf requires a format statement ("%d") and "address" of the variable that input is read into. Each variable is given a location in the memory and &x indicates the actual memory location for x. You can see this memory location by typing printf("%x", &x); What you will see is a 32 bit address variable for x given as a hexadecimal(base-16) number. Scanf stops when its exhaust the input string. Scanf returns the number of successful matches. Basic scanf conversions can be found on Table 7-2 on page 158.
If you dealing with a stream of characters from keyboard you can use the function: int **getchar(void)**
getchar returns the ASCII value of the next input character or EOF, a constant defined in stdio.h Another function that may be of use is the sscanf. The prototype for sscanf is:
int sscanf(char*S,char* format, arg1, arg2,…)
Where it scans the string S according to the format statement order of args. Blanks within the format statement are ignored. It is important that arguments to scanf statement are addresses of variables.

Reading "Characters" from stdin
You need to be extra careful about reading characters from stdin. If you just want to read one character from the stdin (for example, to get a menu option) one problem may be that the linefeed character (ASCII=10) may still be in the buffer waiting to be read. Although you might think that you can flush the buffer using fflush, fflush(stdin) does not work in some systems. One possibility is to write a dummy function

```
int flush(){
  char ch;
  fscanf(STDIN, "%c", ch);
}
```

To flush just the line feed character. After reading a character from the stdin, just call flush to do the cleanup. Here is a simple program (courtesy Tim Hoffman) that illustrates some of the concepts.

```
/* hello.c
  Illustrates: fprintf, fflush, stdin, stdout
  Author: Tim Hoffman
*/

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

int main( int argc, char *argv[] )
{
        int x,y;
fprintf(STDOUT,"Hello World: Enter two small positive numbers for x and y: ");
        fflush(stdout); /* needed because output streams are buffered */
        fscanf(STDIN, "%d %d", &x, &y);
        fprintf(STDOUT,"\nYou entered %d for x and %d for y\n", x,y);
        return EXIT_SUCCESS;
}
```

File I/O

Any file is treated as a stream of bytes ending with the EOF character. However there is a distinction between text file and a binary file. A text file is considered a file of readable characters with lines ending with newline('\n') character. Here is an example of a text file.

```
10\n
20\n
eof
```

A binary file on the other hand is a sequence of bytes stored in a file. So if your intention is to store 10 and 20 in the file, you can store them very compactly using just two bytes.

```
0000101000010100
```

The trick is that you need to know how to read the data from the binary file since there are no newline or EOF characters or spaces that separates the data.

Reading from a File

A file can be considered a data stream and the first part of reading from a file is to open up a pointer to that file. For example;

FILE* fp; // defines a pointer to any file (input/output/text/binary)

To open a file for reading we simply use the fopen function defined in stdio.

fp = fopen(filename, "r");

This indicates that the file with the filename(a string) is open for read only. We can combine the two statements for example by writing

FILE* fp = fopen("data.txt", "r");

Available file opening modes are "r" = read, "w" = write and "a" = append. Some systems may require binary files to open with "b" mode. So you may write:

FILE* fp = fopen("data.bin", "rb");

If the file cannot be open fopen will return NULL. You need to check this before starting to read data from the file.

There are two functions that can be used to read from a file. fscanf and fprintf. The function prototype for fscanf is

int fscanf(FILE* fp, const char* format, aorg1, arg2, …);

fscanf reads formatted data from a file stream fp and stores them in arguments according to the format statement given.

For example if a file data.txt contains two short ints

```
10
20
```

Then you can read two numbers in the file by;

```
FILE* fp = fopen("data.txt", "r");
int x, y;
fscanf(fp,"%d %d", &x, &y);
```

WRITING TO A FILE

We can open a file as

FILE* fout = fopen(“out.txt”,”w”);

This opens the out.txt in your working directory (it creates a new one if it does not exists) for writing. We can write, for example two integers to the file as

fprintf(fout,”%d %d”, 20, 30);

here is a program (courtesy Tim Hoffman) that illustrates concepts as listed.

```
/* fileio.c
  Illustrates:
                argc, argv
                atoi(), exit()
                printf(), fprintf(), fscanf()
                fopen(), fclose()
   Author: Tim Hoffman
*/

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

int main( int argc, char *argv[] )
{
 FILE *infile, *outfile; /* infile and outfile are pointers to FILE objects  */
 int x,i,n;

 if (argc < 3 )
 {
   printf("must enter two values on cmd line: a small positive integer followed by name
         of output file.\n");
  exit( EXIT_FAILURE );
 }

n = atoi( argv[1] ); /* read the man pages on atoi(). Converts a string to an int */
if (n<=0)
{
 printf("must enter a SMALL POSITIVE INT  followed by name for output file on cmd
       line\n");
 exit( EXIT_FAILURE );
}

 outfile = fopen(argv[2], "wt" ); /* "wt" means we are writing text to the file */
 if (NULL==outfile) /* if the open fails then a NULL pointer was put into outfile */
                {
                printf("Can't open %s for output.\n", argv[2] );
                exit( EXIT_FAILURE );
```

```
}

/* READY TO WRITE A SEQUENCE Of INTS TO THE OUTPUT TEXT FILE */

printf("\nWriting to file %s\n", argv[2] );
for (i=1 ; i<=n ; ++i )
{
  printf("...wrote %d\n",i);
  fprintf( outfile, "%d\n", i );
}
fclose( outfile );

/*RE-OPEN THAT FILE AS INPUT AND READ THEM BACK IN AND ECHO TO STDOUT */

 infile = fopen(argv[2], "rt" ); /* "rt" means we are reading the text file */
 if (infile==NULL)
/* we really don't expect this to happen considering we just wrote it - but we always test */
 {
printf("Can't open %s for input.\n", argv[2] );
 exit( EXIT_FAILURE );
}

printf("\nNow reading from file %s\n", argv[2] );
for (i=1 ; i<=n ; ++i )
{
fscanf( infile, "%d", &x );
printf("...read %d\n",x);
}
fclose( infile );

   return 0;
}
```

READING FROM A BINARY FILE

If the file is binary, then we cannot use fscanf to read the data from the file. Instead we need to use : fread

The function prototype for fread is

For example if we want to read 2 bytes from memory and store them in a short int variable x, we can write;

fread(&x, sizeof(x),1,fp);

The prototype for fread is

size_t fread (void * ptr, size_t size, size_t count, FILE * stream);

We will discuss more on binary files later in the course.

Arrays in C

Arrays are static blocks of memory assigned by the compiler. Arrays can be initialized and implicitly assigned a size. For example

int A[] = {1,2,3};

declares an array of size 3 and assign values to 1, 2, 3. Array indices starts from 0.

An array can also be declared, as

const int n = 10;
int A[n];

The size of the memory assigned is the number of bytes necessary to hold n integers. The name of the array, A in this case, serves as the address of the first byte of the array memory block. For example, the statement

printf("%x\n", A);

outputs the address of the starting byte of the array. As we learn pointers we will be able to manipulate the values of the array using indirect pointers.

C does not check for array boundaries. So a statement such as A[n] = 10; is completely legal and but could have some devastating effects. This is a weakness of C language and a source of hacker attacks on code written in C. At CMU, there are efforts to create "safe" subsets of C such as c0. If you take 15-122, you will be exposed to some of these ideas.
We can define arrays of any standard/primitive types (int, double, char etc) or user defined types(discussed later). Arrays can be defined as dynamic structures once we understand how to allocate memory using malloc or calloc. This will be discussed later.

Two Dimensional Arrays
C can define two dimensional arrays. For example,

char List[100][20];

Defines a two dimensional array of 100x20, where we have 100 rows to store a string of max size 20. We can set the character 2 of string 50 to 'a' by doing

List[50][2] = 'a';

Strings

It is very important to understand how strings are processed in C. Unlike in Java, where String is an object, in C a string is considered to be a sequence of characters ending with the null character '\0'. It is very important to have '\0' at the end a string or else it may throw run time errors (or segmentation faults) if we try to do something like this.

```
char name[4];
name[0]='g'; name[1]='u'; name[2]='n'; name[3]='a';
fprintf(STDIN, "%s", name);
```

It is possible in some systems that the null character may be automatically added. We will talk about C string in detail next week, but until then you can see some of the functions available for C strings from Bb. We just touched on some of the topics of IO, strings and files in C. We will be discussing more of these things later in the course.

Functions

Functions allow programs to be broken into smaller, more manageable components. Functions can also hide details of the operations from the program that uses it. Also using functions allow programmers to debug code more easily. C functions can be included in separate files, compiled and linked later to create the executable. The basic form of a function is:

```
<return_type> function_name(arguments){
    Function_statements
    Return value
}
```

Although functions support modular code development, they are generally less efficient as function execution uses run time stack for function evaluation. All arguments to a C function are passed by value. That is a copy of the variables is placed in the runtime stack so function can perform its operations. As C supports passing addresses of variables into functions, one can effectively passed an argument by "reference" so function can modify the value of a global variable. We will discuss more about functions later in the course.

Further Readings

K & R Chapter 1 – Tutorial Introduction Pages 5-31

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	&#127;	DEL

Source: www.LookupTables.com

ASCII TABLE

Exercises

1. Suppose you want to process a binary file (a sequence of bytes ending with EOF character) and output characters in the file. How would you do that?
2. Write a program to read a file of text and count the words in the file. Output one word per line to STDOUT.
3. Suppose a TAB is 3 characters long. Write a program that will take a string (with tabs) as input and replace each tab with a space.
4. Suppose a file of the following format is given.

   123/n
   34/n
   EOF

Write the binary representation of the file. See if you can write a little program to confirm your output.

Answers

1. We can read one character at a time from any file until we hit EOF character (assuming file has an EOF character).
   char ch;
   FILE* fp = fopen(filename,"r");

while (fscanf(fp,"%c", &ch) != EOF) { ….}

4. 123/n
   34/n
   EOF

123/n = 00110001001100100011001100001010
34/n = 00110011001101000001010
EOF   = There is no ASCII value for EOF