Lecture 02
C Strings and File IO

In this lecture

- **Interactive and File IO**
- **C strings**
- **Strings and functions**
- **String.h**
- **ASCII Table and Operator Precedence Table**
- **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 is not part of the language, but by using stdio.h libraries one can use I/O routines in C. Whether the I/O comes from terminal or external device such as a tape or disk, C deals with it in a standard way. External files can be treated as text files or binary files. STDIN and STDOUT are the terminal input and output. There are two stdio.h functions that can be used for this task. printf is used for standard output while scanf is used for standard input.

Writing output to STDOUT

Let us look at printf first. This is used to output to STDOUT (or terminal) and is a function with variable number of arguments. Here is an example.

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

The prototype of printf is: int printf(char *s, arg1, arg2, …)
Where S typically called a format string contains regular characters (eg: "This is ..") and specification of conversion of arguments. For example first specification %d instructs the compiler to replace that by the decimal representation of 134, the first argument. The number of format specifications inside the format string must match the number of arguments.
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;
  scanf("%c",&ch);
}
```

To flush just the line feed character. After reading a character from the stdin, just call flush to do the cleanup.

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, arg1, 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);

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.

C Strings

Before we begin to define, how C treats strings, let us try to understand the concept of a pointer or address. We will discuss in detail what pointers mean shortly, but for now we want to start with a definition as follows.

char* s;

The above statement simply indicates that s is a pointer to (or address of) a character. A String is simply defined as an array of characters and s is the address of the first character (byte) of the string. We recall that in Java, a String is an object that inherits many methods. [A complete description of Java strings can be found in its API].

But in C, a string is just an array of characters that does not have any inherited properties. A valid C string ends with the null character '\0' [slash zero]. Therefore the amount of memory required for a C string is 1 + length of the actual string. Failure to make sure that a string ends with '\0' may result in unpredictable behavior in your code.

Initializing a C String

A constant string s can be simply initialized as

char* S = "guna\0";

However no memory is allocated for s. If we try to write to s, then it may cause a segmentation fault since memory has not been allocated explicitly. So it is important to allocate memory first and then copy the string to the location. To allocate a block of memory to hold a string, we use the malloc function. The malloc(int n) returns a pointer to (or an address of) a block of n bytes. Note that a string with n characters requires n+1 bytes (n for the string AND 1 byte to store the '\0' character). The following code allocates 5 characters to store the string "guna" + '\0'.

```
char *S = malloc(5*sizeof(char));
strcpy(S,"guna");
```

Alternatively we can also write

```
char s[5];
strcpy(S,"guna");
```

Reading a String from stdin

fscanf function can be used to read a string from stdin or any external input stream infile.

char s[10];
fscanf(stdin,"%s",s);

Reading a string using fscanf is somewhat dangerous. It is possible that the input you enter may be longer than the memory allocated by the character array. For example, if you type something more than 9 characters, enough memory have not been allocated for the string and the program may segfault. For example, enter the following program and see what happens if you enter something significantly longer than 10. The behavior of the program is completely unpredictable.

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

int main(int argc, char* argv[]){
  char S[10];
  fscanf(stdin,"%s",S);
  printf("The input is %s \n", S);
  return (EXIT_SUCCESS);
}
```

You need to be careful about managing memory for strings. This is especially true if you are reading strings of variable lengths and the size of the memory cannot be fixed in advance. One possible way to safely read strings is to use fgets function.

char *fgets(char *s, int size, FILE *stream);

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A '\0' is stored after the last character in the buffer.

There is also another version, gets as follows

char *gets(char *s);
However, DO NOT use gets since we do not know how many characters will be read from the stdin.

Warning: *Never use gets().Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to*

*store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead. [**Source: UNIX Manual**]*

Before using fgets we need to make sure a buffer has been allocated to read in the string. For example

char buffer[50];
fgets(buffer, 40, stdin);

will read 39 characters into the buffer
(max buffer size 50).

Writing a String to stdout

If a string is properly read, then we can write the string to an output stream as follows.

fprintf(stdout,"%s",buffer);

or use formatting such as

fprintf(stdout,"%20s",buffer);

Another useful function for string output is **sprintf**. This is particularly useful if you need to construct a string out of fixed and variable lengths, integers, floating points numbers etc. For example you can think of a CMU student course record in the format

S07,gunadean,Guna,Dean,SCS,CS,2,L,4,15111 ,1 ,,

Given the values of individual fields this can be created using sprintf. The prototype for sprintf is

int sprintf(char *str, const char *format, ...);

where 3 dots as the last argument indicates a variable length argument list(we will learn how to write such functions later in the course). An example of how to use sprintf is given below.

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

int main(int argc, char* argv[]){
  char buffer[256];
  sprintf(buffer,"%s/%s%d.c",
        "/afs/andrew.cmu.edu", "myfile", 34);
```

```
  printf("%s\n",buffer);
  return (EXIT_SUCCESS);
}
```

The output produced by the above code is:

/afs/andrew.cmu.edu/myfile34.c

other library functions that can be used for string output are:

int puts(char* s);
int fputs(char *s, FILE* outfile);

Passing a String(s) to a function

A string or an array of strings can be passed as an argument to a function. Suppose for example we have a character buffer we need to pass to a function.

char buffer[50];

we can write a function as
void foo(char [] , /* other parameters here*/);

The name of the char array (or buffer in this case) is in fact the address of the first character

Passing the address of the buffer allows direct manipulation of the content at the buffer location. We can call the function foo as

foo(buffer,….);

So suppose foo's job is to read a string of length n into buffer from stdin. Then it can be defined as

```
void foo(char[] s, int n){
  fgets(s,n,stdin);
}
```

Returning a String from a function

A string or the address of a string can be returned from a function. Consider the following foo function.

```
char* foo(){
  char* s;
  s = malloc(n);
  return s;
}
```

The function allocates memory for a string, and returns its address back to main.

Swapping Two Strings

Suppose we need to swap two strings. A simple swapstring function can be written as follows.

```
void swapstrings(char A[], char B[]){
   char tmp[10];
   strcpy(tmp,A);
   strcpy(A,B);
   strcpy(B,tmp);
}
```

Note that it is important to do string copy, instead of just assigning addresses of strings, that would only result in a local change.

scanf("%s",buffer) and gets()

These functions are vulnerable to buffer overflows and may pose some security problems. Never use them in your programs. You can use fgets instead.

String.h methods

C strings are supported by the string.h library. The following functions are available.

Copying	
void *memcpy(void *dest, const void *src, size_t n);	The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas may not overlap.
void *memmove(void *dest, const void * src, size_t n);	The memmove() function copies n bytes from memory area src to memory area dest. The memory areas may overlap.
	The strcpy() function copies the string

char *strcpy(char *dest, const char *src);	pointed to by src (including the terminating '\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.
char *strncpy(char *dest, const char *src, size_t n);	The strncpy() function is similar, except that not more than n bytes of src are copied. Thus, if there is no null byte among the first n bytes of src, the result will not be null-terminated.
Concatenation	
char *strcat(char *dest, const char *src);	The strcat() function appends the src string to the dest string over-writing the '\0' character at the end of dest, and then adds a terminating '\0' character. The strings may not overlap, and the dest string must have enough space for the result.
char *strncat(char *dest, const char *src, size_t n);	The strncat() function is similar, except that only the first n characters of src are appended to dest.
Comparison	
int memcmp(const void *s1, const void *s2, size_t n);	The memcmp() function compares the first n bytes of the memory areas s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.
int strcmp(const char *s1, const char *s2);	The strcmp() function compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.
int strncmp(const char *s1, const char *s2, size_t n);	The strncmp() function is similar, except it only compares the first (at most) n characters of s1 and s2.
Searching	
char *strchr(const char *s, int c);	The strchr() function returns a pointer to the first occurrence of the character c in the string s.

char *strrchr(const char *s, int c);	The strrchr() function returns a pointer to the last occurrence of the character c in the string s.
char *strtok(char *s, const char *delim);	The strtok() function can be used to parse the string s into tokens. The first call to strtok() should have s as its first argument. Subsequent calls should have the first argument set to NULL. Each call returns a pointer to the next token, or NULL when no more tokens are found.
char *strstr(const char *haystack, const char *needle);	The strstr() function finds the first occurrence of the substring needle in the string haystack. The terminating '\0' characters are not compared.
Other	
void *memset(void *s, int c, size_t n);	The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.
size_t strlen(const char *s);	The strlen() function calculates the length of the string s, not including the terminating '\0' character.
void *memchr(const void *s, int c, size_t n);	The memchr() function scans the first n bytes of the memory area pointed to by s for the character c. The first byte to match c (interpreted as an unsigned character) stops the operation.

Source: Unix manual

You can find more unix documentation on string.h by typing

- man string.h

Tokenizing a String

A String can be tokenized according to a delimiter. For example, let us assume we need to tokenize string S and the delimiter is the blank. The following code can be used to tokenize the string into tokens.

```
char* tk = strtok(S," ");
do {
    printf("%s\n", tk);
} while ((tk=strtok(NULL," ")) != NULL);
```

Note that the first call to the strtok uses the original string S and the subsequent calls we are passing NULL. When there are no more tokens. Strtok returns NULL and the while loop ends.

Exercises

[1] A typical entry on CMU student records looks like this

S07,gunadean,Guna,Dean,SCS,CS,2,L,4,15111 ,1 ,,

The course datafile entry is a comma delimited file containing the following information: semester, computer id, student's last name, student's first name, college, department, class, grade option, qpa scale, course and section.

Write a function that takes a typical entry as a string and break down its fields into appropriate data types. Return a struct containing the data. [read about structs in K&R page 128]

[2] What is wrong with this code?

```
{char *s1 = "Hello, "; char *s2 = "world!"; char *s3 = strcat(s1, s2);}
```

[3] What happens with this code? Please explain

```
char* s = "guna\0";
char buffer[20];
printf("%s is of length %d\n",s, strlen(s));
strcpy(s, buffer); // strcpy(dest,source)
```

[4] Why is this function is bad? Find all problems you can think of.

```
char* badfunction(int n){
  char A[n];
  strncpy(A,"ghfhhfhhhfhhfdfasfff",n);
  return A;
}
```

[5] Explain why calling this function would not swap two integers A and B.

```
void intswap(int x, int y){
   int temp = x;
   x = y;
   y = temp;
}

int main(){
```

```
  int A=10, B=20;
  intswap(A,B);
}
```

[6] Explain why calling this function would not swap two strings A and B.

```
  void stringswap(char x[], char y[]){
     char temp[10] = x;
     x = y;
     y = temp;
  }

int main(){
  char A[10]="guna\0", B[10]="me\0";
  stringswap(A,B);
}
```

[7] 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?

[8] 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.

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

Operator Precedence

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (change *type*) Dereference Address Determine size in bytes	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

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) { ….}
```

2. 123/n
   34/n
   EOF

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