Files

What is a file?

- Logical collection of 1s and 0s

Two basic types:

- **Text (aka ASCII) files**: plain printable text
  - Generally human readable and editable
  - Examples: `.c`, `.java`, `.txt`, `.tex`, `.sh`, `.html`, `.rst`, `.py`
- **Binary files**: everything else
  - Kind of a misnomer (is ASCII not binary?)
  - Generally not human readable and editable (looks like garbage)
  - Usually generated and interpreted by some program
  - Examples: `.o`, `.exe`, `.jpg`, `.mp3`, `.wmv`, `.doc`, `.xls`, `.ppt`
- Somewhere in between
  - Example: `.ps`, `.pdf`

AN EXAMPLE ASCII FILE (A SIMPLE TEXT FILE)

The Contents Of A File

ex1_ascii_file.txt

```
01000011010010010101010000100000
00110101001110010011001100100000
01001001010100110010000001001101
01011001001000000100110001001001
01000110010001010100100001
```

Appears to be a random bunch of 1s and 0s

The Contents Of A File

ex1_ascii_file.txt

```
01000011010010010101010000100000
00110101001110010011001100100000
01001001010100110010000001001101
01011001001000000100110001001001
010001100100010100100001
```

Unless we assign it meaning!

Let’s break it down byte-by-byte…

The Contents Of A File

ex1_ascii_file.txt

```
0100 0011  0100 1001  0101 0100  0010 0000
0011 0101  0011 1001  0011 0011  0010 0000
0100 1001  0101 0011  0010 0000  0100 1101
0101 1001  0010 0000  0100 1100  0100 1001
0100 0110  0100 0101  0010 0001
```

But we are programmers, so we love hex!

The Contents Of A File

ex1_ascii_file.txt

4 3	4 9	5 4	2 0
3 5	3 9	3 3	2 0
4 9	5 3	2 0	4 D
5 9	2 0	4 C	4 9
4 6	4 5	2 1	

How many bytes is this file?

19 bytes!

How Do We Assign Meaning?

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

Your old friend ASCII is back!

The Contents Of A File

ex1_ascii_file.txt

4 3	4 9	5 4	2 0
3 5	3 9	3 3	2 0
4 9	5 3	2 0	4 D
5 9	2 0	4 C	4 9
4 6	4 5	2 1	

C	I	T	_
5	9	3	_
I	S	_	M
Y	_	L	I
F	E	!	

19 characters (including spaces)

19 bytes!

A text editor interprets each byte of the binary #s as an ASCII character

The fact that the binary #s have been encoded in ASCII is what makes this an ASCII file

ASCII Files (aka Text Files)

Text editors:

- Windows Notepad, Unix vi, EMACS, etc.
  - They look at every byte and attempt to map it to ASCII
  - They only show ASCII representation of the binary underneath
  - Programmer's give these files meaningful extensions:
    - .TXT, .C, .JAVA, .ASM, etc.
- You cannot open an ASCII file in its true binary form with a text editor!
  - Text editors want to convert it to ASCII first
  - You can see binary form with handy tools:
    - hexdump, od, etc. (there are many)

ASCII Is One Way; Why Can't We Make Another?

- Programmers do it all the time!
  - .jpg, .doc, .ppt, .xls, .exe, .class, .obj, .o, and on and on!
- We refer to these as binary files
  - Because their binary representation cannot be mapped to ASCII
- Programmers simply make up a format…
  - …and then stick to it!
  - Example: when Microsoft Word encounters a .doc file
    - Only it understands the made-up binary encoding
    - Like an ASCII editor, Word doesn't show you the binary underneath; it maps each byte (or more) to its own meaning

AN EXAMPLE NON-ASCII FILE (THE LC4'S .OBJ) – THE FORMAT

A Wonderful Example Of A Non-ASCII File Format

PennSim takes in an ASCII file: (.ASM)

- Assembles it and produces a binary .OBJ file
  - That .OBJ file is not just the binary equivalent of the .ASM
  - It is a format that instructs PennSim how to fill up memory
  - We made it up!
- If we close PennSim and reopen it:
  - We may reload the .OBJ file
  - Memory will fill up the same way
  - We no longer need the original .ASM file

A Wonderful Example Of A Non-ASCII File Format

PennSim's .OBJ file format explained:

- Breaks down into 5 sections
  - We call the beginning of each section a header

1. **CODE** section
   - Maps to the .CODE directive (your code)
2. **DATA** section
   - Maps to the .DATA directive (initial data values)
3. **SYMBOL** section
   - Maps to the LABELS you create in your assembly code
4. **FILENAME** section
   - Maps to the name of the .C files the assembly came from
5. **LINE NUMBER** section
   - Tells you which assembly lines came from which .C file

  - These sections can repeat over and over

A Wonderful Example Of A Non-ASCII File Format

PennSim's .OBJ file format explained:

- The header tells us the length of each section
  - Each header has its own format:

1. **CODE** header (3 word header = 3 x 16 header)
   - Format: `xCADE, <address>, <n=#words> (16 bits each)`
2. **DATA** header (3 word header = 3 x 16 header)
   - Format: `xDADA, <address>, <n=#words> (16 bits each)`
3. **SYMBOL** header (3 word header = 3 x 16 header)
   - Format: `xC3B7, <address>, <n=#bytes> (16 bits each)`
4. **FILENAME** header (2 word header = 3 x 16 header)
   - Format: `xF17E, <n=#bytes>                (16 bits each)`
5. **LINE NUMBER** header (3 word header = 3 x 16 header) – *no body*
   - Format: `x715E, <address>, <line>, <file-index>`

- The <n> in each header tells us the length of the section

AN EXAMPLE NON-ASCII FILE (THE LC4'S .OBJ) – THE CONTENTS

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
CA DE 00 00 00 02 90 02
10 00 DA DA 40 00 00 01
00 00 C3 B7 40 00 00 05
4D 59 56 41 52 C3 B7 00
00 00 06 4C 41 42 45 4C
31
```

Appears to be a random dump of hex numbers

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
CA DE 00 00 00 02 90 02
10 00 DA DA 40 00 00 01
00 00 C3 B7 40 00 00 05
4D 59 56 41 52 C3 B7 00
00 00 06 4C 41 42 45 4C
31
```

Appears to be a random dump of hex numbers, until we assign meaning!

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
CADE 0000 0002
90 02 10 00
DADA 4000 0001
00 00
C3B7 4000 0005
4D 59 56 41 52
C3B7 0000 0006
4C 41 42 45 4C 31
```

First, look for the headers

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
CADE 0000 0002
90 02 10 00
DADA 4000 0001
00 00
C3B7 4000 0005
4D 59 56 41 52
C3B7 0000 0006
4C 41 42 45 4C 31
```

Next, interpret the headers!

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
CADE 0000 0002
90 02 10 00
```

xCADE, <address>, <n>

.CODE
x0000
n=2

Meaning that next 2 words are at ADDRESS 0000

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
CADE 0000 0002
90 02 10 00
```

```
xCADE, <address>, <n>
```

What are the next 2 words?

```
9002 = 1001 000 000000010
       CONST RD  IMM9

1000 = 0001 000 000 000 000
       ADD  Rd  Rs      Rt
```

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
C3B7 0000 0006
4C 41 42 45 4C 31
```

```
xC3B7, <address>, <n>
```

```
.C3B7
x0000
n=6
```

Meaning that the next 6 bytes are ASCII label for address x0000

Example: .ASM->.OBJ Format Explained

file_format_example.asm

```
.CODE
.ADDR x0000

LABEL1
  CONST R0, #2
  ADD R0, R0, R0

.DATA
.ADDR x4000
  MYVAR .BLKW x1
```

file_format_example.obj

```
C3B7 0000 0006
4C 41 42 45 4C 31
```

xC3B7, <address>, <n>

What are the next 6 bytes?

4C 41 42 45 4C 31

ASCII Equivalent:

L A B E L 1

Example: .ASM->.OBJ Format Explained

The breakdown of the rest of the .OBJ sections

- Is given in the homework
- This was just meant to give you a sample of how they work

OVERVIEW OF FILES IN C

An Important Metaphor

- Files were developed to model sequential access devices like magnetic tape drives
- The basic operations on files make sense in this context

Basic File Operations

Opening a file for reading or writing

- Read/Write head positioned at the start of the file

Read/Write an element to the file

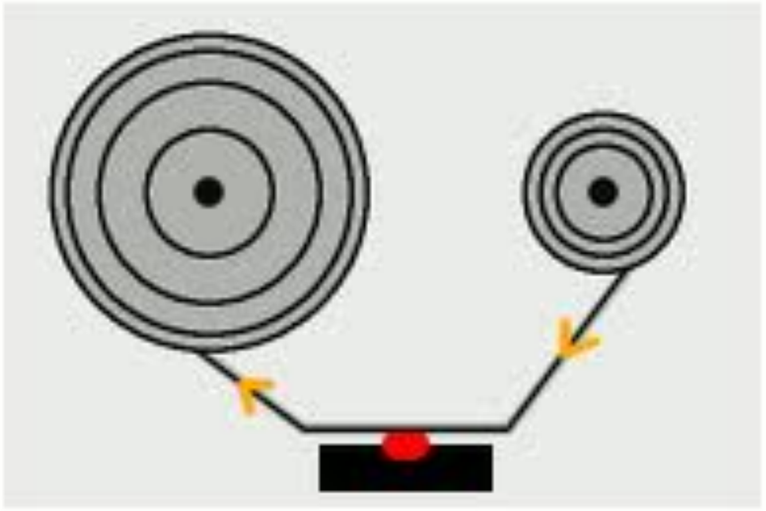
- Advancing the read/write head by the number of bytes read or written.

Rewind

- Rewind file to the start

Seek

- Seek to a specific position in the file



Tape drive - sequential access

Files

In C, files are very simple objects – they simply consist of a sequence of bytes. Any interpretation that we ascribe to those bytes is up to the programs we write.

Basic operations on files

- Open
- Close
- Read
- Write

Files usually exist within the context of a **file system** which provides an overall context for organizing and naming the files.

C treats files in two ways: text or binary

Two basic types

Text (aka ASCII) files: plain printable text

- Generally human readable and editable
- Readable/writeable a byte at a time (since ASCII needs 8-bits)

Binary files: everything else

- Kind of a misnomer (is ASCII not binary?)
- Generally not human readable and editable (looks like garbage)
- Readable/writeable in multiple bytes at a time

Modeling I/O as File Access

In the UNIX operating system, I/O devices were modeled as files: you could **open** them, **write** bytes to them and/or **read** bytes from them.

This made sense in the context of the I/O devices of the time which did in fact work that way – like tape drives, keyboards, and ASCII terminals.

USEFUL FUNCTIONS FOR WORKING WITH FILES IN C

C Functions For Operations On Files

- **fopen( )**
- **fclose( )**
    - To open/close a file

- **fgetc( )**
- **fputc( )**
    - To read/write 1 character (aka a byte) from/to a file

- **fgets( )**
- **fputs( )**
    - To read/write 1 line (as a string) from/to a file

- **fread( )**
- **fwrite( )**
    - To read/write multiple bytes from/to a file

READ/WRITE BYTE AT A TIME

Function: FOPEN( )

Purpose: helper function to open up a file

Function declaration:

FILE* fopen (const char *filename, const char *mode)

2 Arguments:

filename: a string containing name of the file in the file system

mode: a string containing the type of file access (read/write/etc.)

- "r" – open file for reading
- "w" – open file for writing
- "a" – append to file; if file exists, add stuff at the end
- "rb" – open binary file for reading
- "wb" – open file for binary output

Return:

- If file does not exist or cannot be created, NULL is returned
- Otherwise, a pointer to the open FILE is returned

Type: FILE

Purpose:

- A data type that holds information about an open file
- Information like place in the file system, our position in the file, etc.

Details:

- Operating system dependent!
- Not a structure we ever actually probe; we use file helper functions to interact with it

Example structure declaration:

```
typedef struct {
    short int level ;
    short int token ;
    short int bsize ;
    char fd ;
    unsigned int flags ;
    unsigned char hold ;
    unsigned char *buffer ;
    unsigned char *curp ;
    unsigned int istemp;
} FILE ;
```

Function: FGETC( )

Purpose: reads character from a file and advances "position indicator"

Function declaration:

int **fgetc** (FILE* ***stream***)

1 Argument:

stream: the pointer to an open file one wishes to read a character from

Return:

- Returns a byte (1 character) read from the file as an integer
- If the file is at its end, it returns EOF
  - EOF is typically -1 and indicates it has reached the end of the file

Example:

- Calling it once gets you the first byte, and calling it again gets you the next byte!

Function: FPUTC( )

Purpose: writes character to a file and advances "position indicator"

Function declaration:

int **fputc** (int character, FILE* ***stream***)

2 Arguments:

character: the character (byte) to be written
stream: pointer to an open file one wishes to read a character from

Return:

- If no error occurs, returns the same character that has been written
- If an error occurs, returns EOF

What is EOF?

- #define EOF -1 /* could be different # on different systems*/
- A constant used to indicate the end of a file

Examples Of fgetc And fputc

```
#include <stdio.h>

/* this code will copy a file byte by byte */

int main () {
  FILE *src_file, *des_file ;
  int byte_read ;

  src_file = fopen (“file_format_ex.obj”, “rb”);
  if (src_file == NULL) { return 1 ; }

  des_file = fopen (“file_format_ex_cp.obj”, “wb”);
  if (des_file == NULL) { fclose (src_file) ; return 2 ; }
```

Examples (Continued) Of fgetc And fputc

```
  do {
        byte_read = fgetc (src_file) ;
        if (byte_read == EOF) break ;

        fputc (byte_read, des_file) ;
  }  while (1) ;

  fclose (src_file) ; /* flcose() returns 0 if file closes */
  fclose (des_file) ; /* otherwise returns EOF on failure  */

  return 0 ;
}
```

What does this program do?
Makes a copy of a file one byte at a time!

READ/WRITE TO AN ARRAY

Function: FREAD( )

Purpose: reads data from a file into an array

Function declaration:

size_t fread (void **ptr*, size_t *size*, size_t *nmemb*, FILE **stream*)

note: size_t is a typically a typedef'ed "unsigned int"

4 Arguments:

ptr*:* pointer (of any type) to an array you want to read data into

size*:* size (in bytes) of a single element of your array

nmemb: the total number of elements in your array

stream: pointer to an open file to read from

Return:

- The total number of elements successfully read
- If this number is different from ***nmemb*** parameter, you've hit EOF *or an error occurred*

Function: FWRITE( )

Purpose: writes data from an array into a file

Function declaration:

size_t fwrite (const void **ptr*, size_t *size*, size_t *nmemb*, FILE **stream*)

note: size_t is a typically a typedef'ed "unsigned int"

4 Arguments:

ptr*:* pointer (of any type) to an array you want to write data from

size*:* size (in bytes) of a single element of your array

nmemb: total number of elements in your array

stream: pointer to an open file to write to

Return:

- The total number of elements successfully written
- If this number is different from ***nmemb*** parameter, an error has occurred

Example Of fwrite To Write Data From An Array

```
#include <stdio.h>
#define ARRAY_SIZE 4
int main ()
{
  int num1 = 0xFEDC ;      // A single HEX integer
  int array[ARRAY_SIZE]={0, 1, 2, 3} ;    // A integer array
  FILE *theFile = fopen ("output_file", "wb");      // test me for NULL!

   // write out the whole array of ints
  fwrite (array, sizeof(int), ARRAY_SIZE, theFile);

  // write out just a single int!
  fwrite (&num1, sizeof(int), 1, theFile);

  fclose (theFile);
  return 0;
}
```

`sizeof(int)` tells you size, in bytes, required by a type

Why Won’t This Example Work?

```
#include <stdio.h>
int main ()
{

  int* array ;
  FILE *theFile = fopen (“input_file", “rb"); // test me for NULL!

   // read in an array of ints
  fread (array, sizeof(int), 10, theFile);

  fclose (theFile);
  return 0;
}
```

It will compile but it will crash on fread( )…why?

- “array” is a pointer that points to nothing

How do we fix it?

- Allocate memory to the pointer!

One Way To Fix It: Using Memory On The Stack

```
#include <stdio.h>
int main ()
{

  int array[10] ; // allocate memory on the stack first!
  FILE *theFile = fopen (“input_file", “rb"); // test me for NULL!

   // read in an array of ints
  fread (array, sizeof(int), 10, theFile);

  fclose (theFile);
  return 0;
}
```

READ/WRITE STRINGS

Function: FGETS( )

Purpose: reads a string from a file and stores it in string. Stops reading when n-1 characters are read, when the newline character is read, or EOF

Function declaration:

char* fgets (char* *str*, int *n*, FILE* *stream*)

3 Arguments:

str: pointer to an array of ***chars*** to read string into

n: maximum number of characters to be read from file (including NULL)

stream: pointer to an open file to read from

Return:

- On success, function returns pointer to *str*
- On failure, NULL pointer is returned

Function: FPUTS( )

Purpose: writes a string to a file

Function declaration:

char* fputs (const char* *str*, FILE* *stream*)

3 Arguments:

str: pointer to array containing NULL terminated string

stream: pointer to an open file to write to

Return:

- On success, returns a non-negative value
- Otherwise, returns EOF

Example of fputs to Write Data from a String

```
#include <stdio.h>
#define ARRAY_SIZE 4
int main ()
{
   char array[ARRAY_SIZE]={‘T’,‘o’, ‘m’, ‘\0’} ;
   char* array2 = “Tom” ;
   FILE *theFile = fopen ("output_file.txt", "w"); // test me for NULL!

   fputs (array,  theFile) ;      // why don’t I need array size?
                                  // fputs writes until it hits NULL
   fputs (array2, theFile) ;      // write out char*
   fputs (“Tom”, theFile) ;       // write out string literal

   fclose (theFile);
   return 0;
}
```

This Code Compiles But Crashes: Why?

```
#include <stdio.h>
int main ()
{
   char* array1 ;
   char* array2 = “Tom” ;
   char  array3 [3] ;
   FILE *theFile = fopen (“input_file.txt", “r");       // test me for NULL!

   fgets (array1, 4, theFile) ;          // this will fail!
   fgets (array2, 4, theFile) ;          // this too will fail!
   fgets (array3, 4, theFile) ;          // and this one too…why?

   fclose (theFile);
   return 0;
}
```

Array1 points to nothing and must allocate memory
Array2 is considered a constant and cannot change literals!
Array3 does not have enough length

WRITE FORMATTED STRINGS

Function: FPRINTF( )

Purpose: writes formatted string to a file

Function declaration:

int fprintf (FILE* *stream*, const char* *format*, ...)

2 Arguments:

stream: pointer to an open file to read from

Format: “formatted” string to be written to the file

**optionally, the formatted string can embed format tags, replaced with these extra arguments*

Return:

- On success, total number of characters written
- On failure, a negative number is returned

Example Of fprintf To Write Out A Formatted String

```
#include <stdio.h>
int main ()
{
   int a = 5 ;
   char* string = “World” ;
   FILE *theFile = fopen ("output_file.txt", "w"); // test me!

   fprintf (theFile, “Hello World\n”) ;
   fprintf (theFile, “Hello %s\n”, string) ;
   fprintf (theFile, “a = %d\n”, a) ; // prints in decimal
   fprintf (theFile, “a = %x\n”, a) ; // prints in hex
   fprintf (theFIle, “a’s address = %p\n”, &a) ;
                                            // prints out a’s add.

   fclose (theFile);
   return 0;
}
```

stdin; stdout; stderr

File Handles in the C Library

- Constant: **always open**, cannot close
  - `stdin`: standard input (console)
  - `stdout`: standard output (console, for output)
  - `stderr`: standard error (console, for error message)
  - `printf("hi\n");` equivalent to `fprintf(stdout,"hi\n");`

Printing Formatted Strings to the Screen

```
#include <stdio.h>
int main ()
{
   int a = 5 ;
   char* string = “World” ;

   fprintf (stdout, “Hello World\n”) ;            // prints to screen!
   fprintf (stderr, “Error has occurred!\n”) ;    // prints to admin’s console

   return 1;   // indicates error has occurred
}
```

READING FORMATTED STRINGS

Function: FSCANF( )

Purpose: reads formatted string from a file

Function declaration:

int fscanf (FILE* *stream*, const char* *format*, ...)

2... Arguments:

stream: pointer to an open file to read from

format: the "formatted" string to be read to with formatting tags

Return:

- On success, returns number of items successfully matched and assigned
- On failure, less than you expected indicates error has occurred

Reading Formatted Strings From a File

```
#include <stdio.h>
int main ()
{
  char str1[10], str2[10];
  int year, match ;
  FILE *theFile = fopen (“input_file.txt", “r");     // test me!

  match = fscanf (theFile, "%s %s %d", str1, str2, &year);
  printf (“Read in %d items from file”, match) ;     // prints 3

  fclose (theFile) ;
}
```

If text file contained: “I love 1980”

str1=“I”
str2=“love”
year=1980

A MORE COMPLEX EXAMPLE!

Function: SSCANF( )

Purpose: reads formatted data from a STRING!

Function declaration:

int sscanf (const char* *str*, const char* *format*, ...)

2... Arguments:

str: a string sscanf() will parse – basically, the input to your function

format: the "formatted" of the string to be parsed (the formatting tags)

Return:

- On success, returns number of items successfully matched and assigned
- On failure, less than you expected indicates error has occurred, EOF is returned

Using fgets And fscanf To Read And Parse Strings

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

#define MAX_LINE_LENGTH 80

int main ()
{
  int i, num1, num2;
  char input[MAX_LINE_LENGTH];
  char fname[MAX_LINE_LENGTH];

  printf ("Enter command: SCRIPT <filename>,
           SET R<number> <value>, EXIT\n") ;
```

Using fgets And fscanf To Read And Parse Strings

```
 /* now we drop into a loop and read string from keyboard */
while (fgets (input, MAX_LINE_LENGTH, stdin)) {
    printf ("\nString Read In: %s\n", input);

    if (sscanf (input, "SCRIPT %s", fname) == 1)
      printf ("It's a SCRIPT command - fname = %s\n", fname);

    if (sscanf (input, "SET R%d %d", &num1, &num2) == 2)
      printf ("It's a SET command :
               Set register %d to 0x%x\n", num1, num2);

    if (strcmp (input, "EXIT\n") == 0 ) {
        printf ("Exiting loop") ;
        break ;
    }
    printf ("##############################\n");
  } }
```