Extras/Reference/Review

C Operators

C has the following operators:

`*`	Multiplication
`/`	Division
`%`	Modulus
`+`	Addition
`-`	Subtraction
`++`	Increment
`--`	Decrement
`<<`	Shift left
`>>`	Shift right
`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
`~`	Bitwise NOT,

the following relationship operators:

`>`	Greater than
`<=`	Greater that or equal to
`<`	Less than
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to

and the following logical operators:

`&&`	Logical AND
`||`	Logical OR.

If Statement

The if statement in C has the following format:

```
if (condition) {
        statements;
} else                  {
        statements;
}
```

Loops

The while loop has the following format:

```
while (condition) {
        statements;
}
```

The do..while loop that this format (always executes at least once):

```
do {
        statements;
} while (condition);
```

The for loop is as follows:

```
for (initial; test; loop end) {
        statements;
}
```

Functions

In C the programmer must declare and define a function. Declare a function after include external header file or create your own include file and include it in the C file that uses the function.

File I/O

File can be created, write to and read from in two distinct ways in C. The first uses high level routines such as `fscanf()`, `fprintf()`. The other method uses low level I/O routes such as `read()` and `write()`. Each has its place. The high level routes are typically used for file I/O that is formatted where as the low level routines are used for unformatted input or output.

Basic high level I/O functions are `fopen()`, `fclose()`, `fprintf()`, and `fscanf()`. `fopen()` opens a file for reading, write or appending. `fclose()` closes a file. `fprintf()` uses the same formatting as `printf()` but prints to files. `fscanf()` reads formatted information from a file using format specifies like `printf()`. All high level file I/O requires a file descriptor (`FILE *`) to be declared. Suppose a file contained a set of points and an associated number for the set of points:

```
0,0       2.3
0.1,-.1   5.467
0.2,3.11  -34
```

The following reads this file using high level file I/O:

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

// A program to read 3D points from a file
int main(int argc, char * argv[])
{
        int temp;
        FILE * filedesc;        // File handle
        struct data {           // Struct to contain data
                float x;
                float y;
                float value;
        };
        struct data my_data[3];  // the data file has 3 points (array of structs

        // open 'testfile' and open it readonly and in text mode
        filedesc=fopen("testfile","rt");
        if (filedesc==NULL) {  // make sure we did really open it
                perror("Cannot open the file");
                exit(0);
        }
        for (temp=0;temp<3;temp++) {
                // read file with this format:
                // float<comma>float<tab>float<newline>*/
                // an error will result if the file is not in this format */
                fscanf(filedesc,"%f,%f        %f\n",&(my_data[temp].x),
                        &(my_data[temp].y),&(my_data[temp].value));
        }
        fclose(filedesc);       // Close file
        return 0;
}
```

Basic low level file I/O functions are `open()`, `close()`, `read()` and `write()`. They all use file descriptors. These functions do not read formatted information. Thus all I/O is done with buffers. For example to read 64k of a binary file:

```
char *buffer;
int fdes;

// Allocate buffer, open file, and read data.  No error checking occurs!  (bad)
buffer = (char *)malloc(64*1024*sizeof(char));
fdes=open("binary_file",O_RDWR);
read(fdes,(void *)buffer,64*1024*sizeof(char));
```

and to write the first 1k of `buffer`:

```
write(fdes,(void *)buffer,1024);
close(fdes);
```

All C programs have three file handles and file descriptors. The first descriptor (`0`) is standard in (`stdin`), open for reading only. The second (`1`) is standard out (`stdout`), open for output. The last (`2`) is standard error (`stderr`) open for output only. Thus these calls do the same thing:

```
fprintf(stdout,"Hello World\n");

printf("Hello World\n");

char * string[]="Hello World";
write(1,(void *)string,strlen(string));
```

In Unix, files descriptors are not limited to just files. All network communication, serial port I/O and pipe among several other things act the same as reading and writing to a file on disk. This makes networking very easy. Next time we'll look at network I/O.

Write a program that will open a file and print this contents in ASCII and hex. Suppose a file contained this:

```
0123456776543210
```

and in HEX it would be

```
Address                         Data
0x00           0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37
0x08           0x37 0x36 0x35 0x34 0x33 0x32 0x31 0x30
0x10           0x0a
```

The last character in the file is a new line. Output should be like this:

```
# hexdump testfile
Address        Data
0x00000000     0x30313233     0x34353637     0x37363534     0x33323130
                    '0123'         '4567'          '7654'         '3210'
0x00000010     0x0a
                      '.'
```

Print any unprintable characters as a period (.). Your program should take the filename as a command line argument and issue and error if more that one file or no files are on the command line. The error message should be informative and print the correct usage for the program.

Pass the filename on the command line. Do this by declaring main() as follows:

```
int main(int argc, char * argv[])
```

where `argc` is the number of arguments in `argv[]`, `argv[]` is an array of pointers to command line argument strings. `argv[0]` is always the name of the program running. Thus the first argument for the program is `argv[1]` and will contain the passed filename of type `char *` (string).