To:
Tenouk

MODULE 9b_1 - FINAL PART OF C FILE INPUT/OUTPUT 5

MODULE 19 - C++ FILE I/O

create this, delete that, write this, read that, close this, open that

My Training Period: xx hours

- Continue...The following is another program example for non-current directory files location. Our program under /home/bodo/ directory but we try to create **teseight.bin** under /testo1/testo2/ directory. You must have root privilege to create files in this case.

```
////////// rwbinary.c //////////
///// FEDORA 3, gcc x.x.x /////
// reading, writing, rewind and binary data
#include <stdio.h>

enum   {SUCCESS, FAIL, MAX_NUM = 5};

// function prototypes...
void DataWrite(FILE  *fout);
void DataRead(FILE  *fin);
int   ErrorMsg(char  *str);

int main(void)
{
      FILE *fptr;
      // binary type files...
      char filename[] = "/testo1/testo2/teseight.bin";
      int reval = SUCCESS;

      // test for creating, opening binary file for writing...
      if((fptr = fopen(filename, "wb+")) == NULL)
      {
            reval   =  ErrorMsg(filename);
      }
      else
      {
            // write data into file teseight.bin
            DataWrite(fptr);
            // reset the file position indicator...
            rewind(fptr);
            // read data...
            DataRead(fptr);
            // close the file stream...
            if(fclose(fptr) == 0)
              printf("%s successfully closed\n", filename);
      }
      return reval;
      }

// DataWrite() function definition
void DataWrite(FILE  *fout)
{
      int   i;
      double   buff[MAX_NUM] = {145.23, 589.69, 122.12, 253.21, 987.234};

      printf("The size of buff: %d-byte\n", sizeof(buff));
      for(i=0; i<MAX_NUM; i++)
      {
            printf("%5.2f\n", buff[i]);
            fwrite(&buff[i], sizeof(double), 1, fout);
      }
}

// DataRead() function definition
void DataRead(FILE  *fin)
{
      int   i;
      double x;

      printf("\nReread from the binary file:\n");
      for(i=0; i<MAX_NUM; i++)
      {
            fread(&x, sizeof(double), (size_t)1, fin);
            printf("%5.2f\n", x);
      }
```

```
}

// ErrorMsg() function definition
int ErrorMsg(char *str)
{
        printf("Cannot open %s.\n", str);
        return FAIL;
}
```

```
[root@bakawali bodo]# gcc rwbinary.c -o rwbinary
[root@bakawali bodo]# ./rwbinary

The size of buff: 40-byte
145.23
589.69
122.12
253.21
987.23

Reread from the binary file:
145.23
589.69
122.12
253.21
987.23
/testo1/testo2/teseight.bin successfully closed
```

Further C file i/o information

- The following sections compiled from GNU glibc library documentation, provide a summary and other collections that you may interested related to file I/O. Sockets will be discussed in another Module. It looks that the file attributes also not discussed here.

A. Simple Output by Characters or Lines

- The following Table describes functions for performing character and line-oriented output.
- These narrow streams functions are declared in the header file stdio.h and the wide stream functions in wchar.h.

int fputc(*int c, FILE *stream*)
The fputc() function converts the character *c* to type unsigned char, and writes it to the stream *stream*. EOF is returned if a write error occurs; otherwise the character *c* is returned.
wint_t fputwc(*wchar_t wc, FILE *stream*)
The fputwc() function writes the wide character *wc* to the stream *stream*. WEOF is returned if a write error occurs; otherwise the character *wc* is returned.
int fputc_unlocked(*int c, FILE *stream*)
The fputc_unlocked() function is equivalent to the fputc() function except that it does not implicitly lock the stream.
int putc(*int c, FILE *stream*)
This is just like fputc(), except that most systems implement it as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. putc() is usually the best function to use for writing a single character.
wint_t putwc(*wchar_t wc, FILE *stream*)
This is just like fputwc(), except that it can be implement as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. putwc() is usually the best function to use for writing a single wide character.
int putc_unlocked(*int c, FILE *stream*)
The putc_unlocked() function is equivalent to the putc() function except that it does not implicitly lock the stream.
int putchar(*int c*)
The putchar() function is equivalent to putc() with stdout as the value of the *stream* argument.
wint_t putwchar(*wchar_t wc*)
The putwchar() function is equivalent to putwc() with stdout as the value of the *stream* argument.
int putchar_unlocked(*int c*)
The putchar_unlocked() function is equivalent to the putchar() function except that it does not implicitly lock the stream.
int fputs(*const char *s, FILE *stream*)

The function fputs() writes the string *s* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string. This function returns EOF if a write error occurs, and otherwise a non-negative value. For example:
 	
`fputs ("Are ", stdout);`
`fputs ("you ", stdout);`
`fputs ("hungry?\n", stdout);`
 	
Outputs the text Are you hungry? followed by a newline.

int fputws(*const wchar_t *ws, FILE *stream*)

The function fputws() writes the wide character string *ws* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string. This function returns WEOF if a write error occurs, and otherwise a non-negative value.
int puts(*const char *s*)
The puts() function writes the string *s* to the stream stdout followed by a newline. The terminating null character of the string is not written. Note that fputs() does *not* write a newline as this function does. puts() is the most convenient function for printing simple messages. For example: `puts("This is a message.");` Outputs the text This is a message. followed by a newline.

Table 9.11: Output by characters or lines functions

B. Character Input

- This section describes functions for performing character-oriented input. These narrow streams functions are declared in the header file stdio.h and the wide character functions are declared in wchar.h.
- These functions return an int or wint_t value (for narrow and wide stream functions respectively) that is either a character of input, or the special value EOF/WEOF (usually -1). For the narrow stream functions it is important to store the result of these functions in a variable of type int instead of char, even when you plan to use it only as a character.
- Storing EOF in a char variable truncates its value to the size of a character, so that it is no longer distinguishable from the valid character (char) -1.
- So always use an int for the result of getc() and friends, and check for EOF after the call; once you've verified that the result is not EOF, you can be sure that it will fit in a char variable without loss of information.

int fgetc(*FILE *stream*)
This function reads the next character as an unsigned char from the stream *stream* and returns its value, converted to an int. If an end-of-file condition or read error occurs, EOF is returned instead.
wint_t fgetwc(*FILE *stream*)
This function reads the next wide character from the stream *stream* and returns its value. If an end-of-file condition or read error occurs, WEOF is returned instead.
int fgetc_unlocked(*FILE *stream*)
The fgetc_unlocked() function is equivalent to the fgetc() function except that it does not implicitly lock the stream.
int getc(*FILE *stream*)
This is just like fgetc(), except that it is permissible (and typical) for it to be implemented as a macro that evaluates the *stream* argument more than once. getc() is often highly optimized, so it is usually the best function to use to read a single character.
wint_t getwc(*FILE *stream*)
This is just like fgetwc(), except that it is permissible for it to be implemented as a macro that evaluates the *stream* argument more than once. getwc() can be highly optimized, so it is usually the best function to use to read a single wide character.
int getc_unlocked(*FILE *stream*)
The getc_unlocked() function is equivalent to the getc function except that it does not implicitly lock the stream.
int getchar(*void*)
The getchar() function is equivalent to getc() with stdin as the value of the *stream* argument.
wint_t getwchar(*void*)
The getwchar() function is equivalent to getwc() with stdin as the value of the *stream* argument.
int getchar_unlocked(*void*)
The getchar_unlocked() function is equivalent to the getchar() function except that it does not implicitly lock the stream.

Table 9.12: Character oriented input functions

- An example of a function that does input using fgetc(), it would normally work just as well using getc() instead, or using getchar() instead of fgetc(stdin). The code would also work for the wide character stream functions as well.

C. Line-Oriented Input

- Since many programs interpret input on the basis of lines, it is convenient to have functions to read a line of text from a stream. Standard C functions for these tasks aren't very safe: null characters and even (for gets()) long lines can confuse them.
- This vulnerability creates exploits through buffer overflows. That is why you see warning everywhere; you may check your implementation documentation for safer version of those functions. All these functions are declared in stdio.h.

char * fgets(*char *s, int count, FILE *stream*)

The fgets() function reads characters from the stream *stream* up to and including a newline character and stores them in the string *s*, adding a null character to mark the end of the string. You must supply *count* characters worth of space in *s*, but the number of characters read is at most *count* - 1. The extra character space is used to hold the null character at the end of the string. If the system is already at end of file when you call fgets(), then the contents of the array *s* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *s*. **Warning:** If the input data has a null character, you can't tell. So don't use fgets() unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message. We recommend using getline() instead of fgets().
wchar_t * fgetws(*wchar_t *ws, int count, FILE *stream*)
The fgetws() function reads wide characters from the stream *stream* up to and including a newline character and stores them in the string *ws*, adding a null wide character to mark the end of the string. You must supply *count* wide characters worth of space in *ws*, but the number of characters read is at most *count* - 1. The extra character space is used to hold the null wide character at the end of the string. If the system is already at end of file when you call fgetws(), then the contents of the array *ws* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *ws*. **Warning:** If the input data has a null wide character (which are null bytes in the input stream), you can't tell. So don't use fgetws() unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message.
char * gets(*char *s*)
The function gets() reads characters from the stream stdin up to the next newline character, and stores them in the string *s*. The newline character is discarded (note that this differs from the behavior of fgets(), which copies the newline character into the string). If gets() encounters a read error or end-of-file, it returns a null pointer; otherwise it returns *s*. **Warning:** The gets() function is very dangerous because it provides no protection against overflowing the string *s*. The GNU library includes it for compatibility only. You should always use fgets() or getline() instead. To remind you of this, the linker (if using GNU ld) will issue a warning whenever you use gets().

Table 9.13: Line oriented input functions

D. Block Input/Output

- This section describes how to do the input and output operations on blocks of data. You can use these functions to read and write binary data, as well as to read and write text in fixed size blocks instead of by characters or lines.
- Binary files are typically used to read and write blocks of data in the same format as is used to represent the data in a running program.
- In other words, arbitrary blocks of memory, not just character or string objects, can be written to a binary file, and meaningfully read in again by the same program.
- Storing data in binary form is often considerably more efficient than using the formatted I/O functions.
- Also, for floating-point numbers, the binary form avoids possible loss of precision in the conversion process. On the other hand, binary files can't be examined or modified easily using many standard file utilities (such as text editors), and are not portable between different implementations of the language, or different kinds of computers.
- These functions are declared in stdio.h.

size_t fread(*void *data, size_t size, size_t count, FILE *stream*)
This function reads up to *count* objects of size *size* into the array *data*, from the stream *stream*. It returns the number of objects actually read which might be less than *count* if a read error occurs or the end of the file is reached. This function returns a value of zero (and doesn't read anything) if either *size* or *count* is zero. If fread() encounters end of file in the middle of an object, it returns the number of complete objects read, and discards the partial object. Therefore, the stream remains at the actual end of the file.
size_t fwrite(*const void *data, size_t size, size_t count, FILE *stream*)
This function writes up to *count* objects of size *size* from the array *data*, to the stream *stream*. The return value is normally *count*, if the call succeeds. Any other value indicates some sort of error, such as running out of space.

Table 9.14: Block oriented I/O functions

E. Some File System Interfaces

E.1 Deleting Files

- You can delete a file with unlink() or remove().
- Deletion actually deletes a file name. If this is the file's only name, then the file is deleted as well. If the file has other remaining names, it remains accessible under those names.

int rmdir(*const char *filename*)
The rmdir() function deletes a directory. The directory must be empty before it can be removed; in other words, it can only contain entries for . and .. In most other respects, rmdir() behaves like unlink().
int remove(*const char *filename*)

This is the ISO C function to remove a file. It works like unlink() for files and like rmdir() for directories. remove() is declared in stdio.h.

Table 9.15: Remove directory and file functions

- The rename() function is used to change a file's name.

int rename(*const char *oldname, const char *newname*)
The rename() function renames the file *oldname* to *newname*. The file formerly accessible under the name *oldname* is afterwards accessible as *newname* instead. (If the file had any other names aside from *oldname*, it continues to have those names.) The directory containing the name *newname* must be on the same file system as the directory containing the name *oldname*. One special case for rename() is when *oldname* and *newname* are two names for the same file. The consistent way to handle this case is to delete *oldname*. However, in this case POSIX requires that rename() do nothing and report success, which is inconsistent. We don't know what your operating system will do. If *oldname* is not a directory, then any existing file named *newname* is removed during the renaming operation. However, if *newname* is the name of a directory, rename() fails in this case. If *oldname* is a directory, then either *newname* must not exist or it must name a directory that is empty. In the latter case, the existing directory named *newname* is deleted first. The name *newname* must not specify a subdirectory of the directory oldname which is being renamed. One useful feature of rename() is that the meaning of *newname* changes "atomically" from any previously existing file by that name to its new meaning (i.e. the file that was called *oldname*). There is no instant at which *newname* is non-existent "in between" the old meaning and the new meaning. If there is a system crash during the operation, it is possible for both names to still exist; but *newname* will always be intact if it exists at all.

Table 9.16: Rename function

E.2 Creating Directories

- Directories are created with the mkdir() function. There is also a shell command mkdir() which does the same thing.

int mkdir(*const char *filename, mode_t mode*)
The mkdir() function creates a new, empty directory with name *filename*. The argument *mode* specifies the file permissions for the new directory file.

Table 9.17: Create directory function

F. Pipes and FIFOs

- A pipe is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.
- A FIFO special file is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.
- A pipe or FIFO has to be open at both ends simultaneously. If you read from a pipe or FIFO file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file. Writing to a pipe or FIFO that doesn't have a reading process is treated as an error condition; it generates a SIGPIPE signal, and fails with error code EPIPE if the signal is handled or blocked.
- Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

C & C++ programming tutorials

Related C file i/o reading and digging:

1. For C++ and MFC (Windows GUI programming) it is called Serialization and the topics are in Single Document Interface (SDI) and Multiple Document Interface (MDI).
2. Check the best selling C / C++ books at Amazon.com.
3. The source code for this Module is: C file input/output program source codes.
4. Wide character/Unicode is discussed Character Sets, Unicode & Locale and the implementation using Microsoft C is discussed Windows Users & Groups C programming.
5. Implementation specific information for Microsoft can be found Microsoft C Run-Time Tutorials and More Win32 Windows C Run-Time programming Tutorials.