RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Module 05 – File Processing

Files

- Streams vs Files
- An “external” array of bytes
- Type FILE (and FILE *)
- stdin, stdout, and stderr
- fopen() and fclose()
- Character I/O
- Text (string) I/O
- Formatted I/O
- Random I/O
- Binary I/O

"Streams" vs "Files" in C (1)

- C file system is designed to work with a wide variety of devices.
- Devices are different →C file system transforms each device into a logical device
- Think "streams" and I/O system converts it to any device
- Streams
  - Logical device
  - Consistent interface for the programmer
  - Independent of the actual device being used
  - A level of abstraction between the programmer and the device
  - Can write to disk file or another type of device (e.g. console)
  - Has two types: (1) text streams; (2) binary streams

"Streams" vs "Files" in C (2)

- Text Streams
  - A sequence of characters
  - Can be organised into lines terminated by a newline character
    - Optional for the last line
  - Character translation may occur as required by the host environment; For example:
    - newline → carriage return / linefeed *[when writing]*
    - Carriage return / linefeed → newline *[when reading]*
  - Not necessarily a 1-to-1 relationship between characters of the stream and the actual device

"Streams" vs "Files" in C (3)

- Binary Streams
  - A sequence of bytes
  - No character translation occurs
  - There is a 1-to-1correspondence between bytes of the stream and the actual device
  - May contain a certain number of null bytes at the end
    - For example, for padding so the file fills a sector on a disk

"Streams" vs "Files" in C (4)

- Files
  - Can refer to anything from a disk file to terminal or printer
  - Should be associated to a stream by an **open operation**
  - May support different capabilities, for example:
    - Disk files support random access
    - Some printers do not support random access
    - Files may or may not support *position requests* and *position indicator*
  - Disassociate a file and a stream by a **close operation**

"Streams" vs "Files" in C (5)

- Closing Files
  - Disassociates a file and a stream
  - If opened for *writing*, it will be flushed
    - all the content in the stream will be written to the file
  - Files are closed automatically when
    - main() returns
    - exit() is called
    - But not when abort() is called

An “external” array of bytes

- A file in C is simply an external array of bytes
- No other structure – records, indexes etc is imposed by C
- We can access files via the operating system using:
  - C I/O libraries
  - operating system (e.g. UNIX) functions
- We can overlay our own structure on files, by how we store/retrieve our data

Commonly Used Functions

fopen()	Opens a file	**fclose()**	Closes a file
putc() **fputc()**	Writes a character to a file	**getc()** **fgetc()**	Read a character from a file
fgets()	Reads a string from a file	**fputs()**	Write a string to a file
fseek()	Seeks to a specified byte in a file	**ftell()**	Returns the current file position
fprinf()	Similar to printf() but for a file	**fscanf()**	Similar to scanf() but for a file
feof()	Returns true if end-of-file is reached	**ferror()**	Returnes true if an error has occurred
rewind()	Reset the file position indicator	**remove()**	Erases a file
fflush()	Flushes a file		

Type FILE (and FILE *)

- The type FILE is a type defined in <stdio.h> which, along with the I/O functions, form the interface to the operating system management of files.
  - internal structure is machine dependent
  - we don't need to know the details
    - the whole design of the I/O system in C is to insulate us from these implementation specific dependencies
    - portability
- One of these structs is created each time we successfully open a file.
- Access is via a pointer (FILE *).

Standard Streams: stdin, stdout, and stderr (1)

- All C programs automatically have 3 pre-defined files (in <stdio.h>) open and available for use:

  FILE *stdin, *stdout, *stderr;

- These are the defaults used by I/O functions where we don't explicitly use our own file variables:

  scanf(), getchar(), printf(), putchar()

- *stdin* typically defaults to **keyboard**, and *stdout* and stderr typically default to **monitor**

Standard Streams: stdin, stdout, and stderr (2)

- *stdin* is an input stream, and as the name tells, is intended for input.
- *stdout* and *stderr* are both output streams.
- We typically use *stdout* for the intended program output, whereas we typically use *stderr* for diagnostic messages.
- Standard streams can be redirected by the OS:
  - `./myprogram > outputfile.txt`
  - `./myprogram < inputfile.txt > outputfile.txt`
- Standard streams can be redirected via code:
  - `FILE *freopen(const char *filename, const char *mode, FILE *stream);`

fopen() and fclose()

- Prototype is:

  FILE *fopen(char *fname, char *accessMode);

- Usage:

```
char fname[] = "myfile",
FILE *fp;

fp = fopen(fname, "r");
if (fp == NULL)
        fprintf(stderr, "Unable to open file %s\n", fname);
...
fclose(fp);
```

All I/O (including fopen) can always fail, so we must always test whether successful [file does not exist; not sufficient permissions; file may be locked by someone else etc]

- fclose () flushes any remaining data in output buffers to the device.
- Important: close files when finished with them
- No need to test the output of fclose (nothing we can do about it)

fopen() and fclose() (cont'd)

- Access modes can be:

r	read	error if file doesn't exist
w	write	creates file **or** truncates(!) existing file
a	append	creates file **or** appends to existing file

'w' can be dangerous as it will remove the content of the file before writing to it: you might lose data!

- Also:

r+, w+, a+	for update (read and write) access
rb, wb, ab	for binary access
r+b, w+b, a+b	for binary update access

Permissible Values for *Mode*

r	Open a text file for reading	**w**	Create a text file for writing
a	Append a text file		
rb	Open a binary file for reading	**wb**	Create a binary file for writing
ab	Append to a binary file		
r+	Open a text file for read / write	**w+**	Create a text file for read / write
a+	Append or create a text file for read / write		
r+b	Open a binary file for read / write	**w+b**	Create a binary file for read / write
A+b	Append or create a binary file for read / write		

Special Cases for fopen()

1. Open for read-only → file does not exist → fopen() fails!
2. Open for append → file does not exist → will be created
3. Open for append → file exists → new data appended to the end
4. Open for writing → file does not exist → will be created
5. Open for writing → file exists → existing content will be destroyed!
6. r+ and w+ differ in that r+ will not created a new file if it does not exist
7. Open with w+ → file exists → existing content will be destroyed!

Character I/O

C Library Prototypes:

```
int fgetc(FILE *fp);
int getc(FILE *fp);          – macro version of fgetc()
int getchar();               – get a char from stdin
```

ch = fgetc(stdin) *is the same as* ch = getchar()

```
int fputc(int c, FILE *fp);
int putc(int c, FILE *fp);   – macro version of fputc()
int putchar(int c);          – equivalent to fputc(stdout)
```

Character I/O (cont'd)

- Note that character I/O has return type int not char
- If some **error** occurs during I/O (including end-of-file) then these functions return a **value of (-1)**
- <stdio.h> has it defined as:

```
#define EOF (-1)
```

- So we can test for success with our I/O by:

```
int ch;
ch = getchar();
if (ch == EOF)  /* no more data read*/
```

Character I/O (cont'd)

- Unfortunately (-1) is a negative integer value:
  - Probably 16 or 32 bits – certainly more than char's 8 bits
  - "leftmost" (most significant) bit is probably set
  - Assigning 16 (or 32) bits to 8 bits means that some data will be lost
  - Compilers can attempt to preserve either the **sign** or the **value** in such assignments

- So we must declare ch to be int not char … or we may get unpredictable results!!!

Example: Count $ Signs in a Text File

```
pFile=fopen ("myfile.txt","r");
if (pFile==NULL)
   printf ("Error opening file");
else
{
   do {
      c = getc (pFile);
      if (c == '$') n++;
    } while (c != EOF);
fclose (pFile);
printf ("File contains %d$.\n",n);
}
```

- Open file for reading "r"
- Check for fopen errors
- If there is no errors
- Read one character from the file
- Count the $ signs
- Close the file as soon as possible

Borrowed from: http://www.cplusplus.com/reference/cstdio/getc/

Using feof()

- getc() returns EOF (i.e. -1) when end-of-file is reached
- What if the byte read from file is actually equal to -1 ?
  - Can happen for binary streams
  - Will cause incorrect end-of-file detection

- getc() returns EOF when if fails too!
- How to tell the difference of EOF and error?
  - Impossible by checking the result of getc()

- Use feof() instead and it will solve the above issues
  - `while(!feof(fp)) ch = getc(fp);`

Text (string) I/O

C Library Prototypes:

```
char *gets(char *str);
char *fgets(char *str, int n, FILE *fp);

char *puts(char *str);
char *fputs(char *str, FILE *fp);
```

Text (string) I/O (cont'd)

Note: important difference between gets and fgets:

char *gets(char *str);

- reads (from stdin) until new-line or end-of-file (or error)
- Possible to overflow destination array (str)
- New-line char('\n') is replaced with ASCII-NUL ('\0')

char *fgets(char *str, int n, FILE *fp);

- reads (from fp) until new-line or end-of-file (or error) or until (n -1) characters have been read
- protects against overflow of destination array (str)
- new-line is not replaced, ASCII-NUL is appended

Text (string) I/O (cont'd)

- The return type of these functions is char *
- If some error occurs during I/O (including end-of-file) then these functions will return NULL
- Typical usage would be:

```
char str[SIZE + 1];
while (fgets(str, sizeof str, fp) != NULL)
{
    /*  got some input; process it …  */
}
```

- Strongly recommend **never using gets()**; instead use fgets().

rewind()

- Resets the file position indicator to the beginning of the file

```
void rewind(FILE *fp);
```

- Example: Create a file and write some data into it and, without closing it, attempt to read what is currently stored in the file
  - This example needs “w+” as the mode → read/write

ferror()

- Determines if a file operation has produced an error

```
int ferror(FILE *fp);
```

- It works only for the last operation
- Call immediately after each file operation
- Otherwise the error may be lost

remove()

- Erases the specified file

```
int remove(const char *filename);
```

- Returns zero if successful
- Non-zero if unsuccessful

fflush()

- Flushes the content of an output stream

```
int fflush(FILE *fp);
```

- Writes the contents of any buffered data to the file
- If *fp* is null, all files opened for output are flushed
- Returns zero if successful; otherwise, it returns EOF

Formatted I/O

C Library Prototypes:

```
int printf(char *controlString, … );
int fprintf(FILE *fp, char *controlString, … );
int sprintf(char *destString, char *controlString, … );

int scanf(char *controlString, … );
int fscanf(FILE *fp, char *controlString, … );
int sscanf(char *str, char *controlString, … );
```

Formatted I/O (cont'd)

- We start to see now just how complex scanf() and printf() actually are …
- Note that sscanf() and sprintf() are not strictly I/O functions – their target/source are strings in memory
- Also … return-type (int) is the number of successful conversions that were performed
- Return-type can be EOF if the end of input is reached or a read error occurs

Random I/O

C Library Prototypes:

```
int fseek(FILE *fp, long offset, int from);
```

- position into file fp, offset bytes from position from
- from can be:
  - SEEK_SET (=0) start of file
  - SEEK_CUR (=1) current position in file
  - SEEK_END (=2) end of file
- EOF is returned if unsuccessful

fseek() example

```
#include <stdio.h>
int main ()
{
  FILE * pFile;
  pFile = fopen ( "example.txt" , "wb" );
  fputs ( "This is an apple." , pFile );
  fseek ( pFile , 9 , SEEK_SET );
  fputs ( " sam" , pFile );
 fclose ( pFile );
 return 0;
}
```

File now contains "This is an apple"

File now contains "This is a sample"

Borrowed from: http://www.cplusplus.com/reference/cstdio/fseek/

Random I/O (cont'd)

C Library Prototypes:

long ftell(FILE *fp);

- returns the current position in the file (bytes from the start of file)

… also ...

fgetpos(…);

fsetpos(…);

rewind(…); … details can be found in your reference manual ...

Example: ftell(), getting size of a file

```
#include <stdio.h>
int main ()
{
  FILE * pFile;
  long size;
  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) printf ("Error opening file");
  else
  {
    fseek (pFile, 0, SEEK_END);
    size=ftell (pFile);
    fclose (pFile);
    printf ("Size of myfile.txt: %ld bytes.\n", size);
  }
  return 0;
}
```

Move to the end of file

Bytes from the start of file?

Borrowed from: http://www.cplusplus.com/reference/cstdio/ftell/

Binary I/O

C Library Prototypes:

```
size_t fread(void *buffer, size_t num_bytes,
             size_t count, FILE *fp);
```

- read at most (num_bytes * count) bytes from fp into the array pointed to by buffer
- return value is the number of elements actually read or EOF
- typically used to read structs containing non-ASCII (binary) data

Example: Reading the Entire File

```
rm = fopen("ReadMe.txt", "r");
if (rm != NULL) {
    while (!feof(rm)) {
        res = fread(buf, (sizeof buf)-1,1, rm);
        buf[res] = 0;
        printf("%s", buf);
    }
    fclose(rm);
}
```

Read into *buf*; each read is *(sizeof buf) – 1 bytes* long; repeat reading for *1* time only;

Borrowed from: https://blog.udemy.com/fread-c/

Example: Reading the Entire File

```
int main( int argc, const char* argv[] ) {
    FILE *tf;

    tf = fopen("Test.txt", "r");
    fread(&myTest, sizeof myTest, 1, tf);
    fclose(tf);
    printf("First: %c\n", myTest.first);
    printf("Second: %u\n", myTest.second);
    printf("Third: %c\n", myTest.third);
}
```

```
struct TEST {
    char first;
    short second;
    char third;
} myTest;
```

Read the structure myTest from the file **once**

Borrowed from: https://blog.udemy.com/fread-c/

Binary I/O (cont'd)

C Library Prototypes:

```
size_t fwrite(void *buffer, size_t num_bytes,
              size_t count, FILE *fp);
```

- write (num_bytes * count) bytes from the array pointed to by buffer to file fp
- return value is the number of elements successfully written – a value less than *count* indicates an error
- typically used to write structs containing non-ASCII (binary) data

Binary I/O (cont'd)

Example:

```
/* Make a copy of a file */
#include <stdio.h>
#define MAXSTRING 100

int main(void)
{
        char fname[MAXSTRING];
        int c;
        FILE *fp1, *fp2 ;

        fprintf(stdout, "\nInput the source filename: ");
        scanf("%s", fname);
        fp1 = fopen(fname, "rb");          // read in binary mode
        fprintf(stderr, "\nInput the destination filename: ");
        scanf("%s", fname);
        fp2 = fopen(fname, "wb");          // write in binary mode
```

Binary I/O (cont'd)

```
    while (fread(&c, sizeof(c), 1, fp1) > 0)        // read until EOF
    {
            fwrite(&c, sizeof(c), 1, fp2);          // write to the new file
    }

    fclose(fp1);
    fclose(fp2);

    return 0;
}
```