CS240: Programming in C

Lecture 13: File I/O

How should C programs interact with the outside?

- Communicate via “standard” input and “standard” output
  - Typically bound to the display
- Use redirection to read or write to a file
  - a.out < inputfile
  - a.out > outputfile
    - when we use printf(), the results are written to outputfile rather than displayed on the screen.
  - a.out < inputfile > outputfile

More general approach ...

- Redirection is really part of the operating system, not part of the C language
- But, C provides a set of library functions for performing I/O
- We've used one such library extensively:
    - stdio.h
        - provides operations to read (getchar) and write (putchar) characters, print formatted strings (printf), read formatted strings (scanf), etc.

Stdio.h

- Also provides more general operations on files.
- A file is an abstraction of a non-volatile memory region:
  - its contents remain even after the program exits
  - C exposes the file abstraction using the FILE type:
    - FILE *fp   // *fp is a pointer to a file
  - Can only access the file using the interfaces provided by the language

File Systems

File system: specifies how the information is organized on the disk and can be accessed

- Directories
- Files

In UNIX the following are files

- Peripheral devices (keyboard, screen, etc.)
- Pipes (inter process communication)
- Sockets (communication via computer networks)

Files representation

- Text files (human readable format)
- Binaries (for example executables files)

System Calls

- System calls: services provided by the operating system.
- C Library provides support such that a user can invoke system calls through C functions.
- Example:
  - I/O operations (I/O access is slower than memory access)
  - Memory allocation

File manipulation

- Three basic actions:
  - “open” the file: make the file available for manipulation
  - read and write its contents
    - No guarantee that these operations actually propagate effects to the underlying file system
  - “close” the file: enforce that all the effects to the file are “committed”

File Descriptors

Any opened file has associated a non-negative integer called file descriptor.

For each program the operating system opens implicitly three files: standard input, standard output and standard error, that have associated the file descriptors 0, 1, 2

File descriptors

- Primitive, low-level interface to input and output operations.
- Must be used for control operations that are specific to a particular kind of device.

Streams

- Higher-level interface, layered on top of the primitive file descriptor facilities.
- More powerful set of functions for performing actual input and output operations than the corresponding facilities for file descriptors.
- It is implemented in terms of file descriptors
  - the file descriptor can be extracted from a stream and then perform low-level operations directly on the file descriptor
  - a file can be open as a file descriptor and then make a stream associated with that file descriptor.

Opening a file

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

- mode can be "r" (read), "w" (write), "a" (append)
- returns NULL on error (e.g., improper permissions)
- filename is a string that holds the name of the file on disk

int fileno(FILE *stream)

- returns the file descriptor associated with stream

Example

```
FILE *ifp, *ofp;
char *mode = "r";
char outputFilename[] = "out.list";

ifp = fopen("in.list", mode);

if (ifp == NULL) {
  fprintf(stderr, "Can't open input file in.list!\n");
  exit(1);
}

ofp = fopen(outputFilename, "w");

if (ofp == NULL) {
  fprintf(stderr, "Can't open output file %s!\n",
          outputFilename);
  exit(1);
}
```

Reading a file

- Can use fscanf
  - Just like scanf, but requires an extra first parameter, a FILE *, for the file to be read or written

    fscanf(ifp, “<format string>”, inputs)
  - Returns the special value EOF when it encounters the end of file
  - Returns in the normal case the number of values it could read

Example

- Suppose in.list contains

```
foo  70
bar  50
```

- To read elements from this file, we might write

```
fscanf(ifp, “%s  %d”, name, count)
```

- Can check against EOF:

```
while (fscanf(ifp, “%s  %d”, name, count) != EOF)
```

Testing against EOF

- Ill-formed input might not cause comparison with EOF to succeed
  - fscanf returns the number of successful matched items

```
while (fscanf(ifp, “%s  %d”, name, count) == 2)
```

- Can also use feof:

```
while (!feof(ifp)  {
  if (fscanf(ifp, “%s  %d”, name, count) !=2)
     break;
  fprintf(ofp, <format string>, <control arguments>)
}
```

Closing a file

- fclose(ifp);   fclose(ofp);
- Why do we need to close a file?
  - File systems typically buffer output
    - fprintf(ofp, “Some text”)
  - There is no guarantee that the string has actually been written out to disk
  - Could be stored in a file buffer (or cache) maintained in memory
- The buffer is flushed when the file is closed, or when it becomes full.

File pointers

- Three special file pointers:
  - stdin (standard input)
  - stdout (standard output)
  - stderr (standard error)
- Typically stdin is associated with the keyboard device
- stdout and stderr are associated with the display
  - redirecting stdout doesn’t redirect stderr
  - a.out > outfile
- Can be used wherever a regular FILE * is expected

Other file operations

- Remove file from the file system:

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

- Rename file

```
int rename (const char * oldname,
            const char * newname)
```

- Create temporary file (removed when program terminates)

```
FILE * tmpfile (void)
```

Raw I/O

- Read at most **nobj** items of size **size** from **stream** into **ptr**
  - feof and ferror used to test end of file

```
size_t fread(void* ptr, size_t size, size_t nobj, FILE * stream)
```

- Write at most **nobj** items of size **size** from **ptr** onto **stream**

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

File Position

- Set file position in the stream. Subsequent reads and writes begin at this location
- Origin can be SEEK_SET, SEEK_CUR ,SEEK_END for binary files
- For text streams, offset must be zero (or a value returned by ftell -- next slide)

```
int fseek(FILE * stream, long offset, int origin)
```

File Position

- Return the current position within the stream

```
long ftell(FILE * stream)
```

- Sets the file to the beginning of the file

```
void rewind(FILE * stream)
```

- see page 247-248 in the text

Example

```
#include <stdio.h>
int main() {
  long fsize;
  FILE *f;

  f = fopen(“log”, “r”);

  /* compute the size of the file */
  fseek(f, 0, SEEK_END) ;
  fsize =  ftell(f) ;
  fprintf(stderr, “file size is: %d\n”, fsize);

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

Text Stream I/O Read

- Read the next character from the stream and return it as an unsigned char cast to an int, or EOF

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

- Reads in at most one less than size characters from the stream and stores them into the buffer pointed to by s; the buffer is null-terminated. Stop on EOF or error

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

Text Stream I/O write

- Writes the character **c** cast to an unsigned char to **stream** and return the unsigned char cast to int.

```
int fputc (int c, FILE * stream)
```

- Writes the string s to the stream without null terminating; returns a non-negative number (typically 0) on success, or EOF on error

```
int fputs(const char *s, FILE *stream)
```

File Descriptors

- A handle to access a file (or I/O device), like the file pointer in streams
- It is a small non-negative integer used in same open / read-write / close paradigm
- Returned by the open system call; all active opens have distinct file descriptors
- Once a file is closes, fd can be reused
- Same file can be opened several times, and be associated with multiple fd’s

Management functions

#include <unistd.h>

int open(const char *pathname, int flags);

int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

Flags: O_RDONLY, O_WRONLY or O_RDWR bitwise OR with O_CREAT, O_EXCL, O_TRUNC, O_APPEND, O_NONBLOCK O_NDELAY

int close(int fd);

FD IS an INT (file descriptor) not a FILE* !!!!!

Read/Write

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, const void *buf, size_t count);

fd is a descriptor, _not_ FILE pointer

Returns number of bytes transferred, or -1 on error

Normally waits until operation is enabled (e.g., there are bytes to read), except under O_NONBLOCK and O_NDELAY (in which case, returns immediately with “try again” error condition)

Example

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

int main() {
  int  f1, f2;
  int n;
  char buf[100];

  f1 = open("log1", O_RDONLY);
  f2 = open("log2", O_RDONLY);
  fprintf(stderr, "Log1 file descriptor is: %d\n", f1);
  fprintf(stderr, "Log2 file descriptor is: %d\n", f2);
  close(f1);
  close(f2);

  f2 = open("log2", O_RDONLY);
  fprintf(stderr, "Opening again log2, notice the new file descriptor: %d\n", f2);
  close(f2);

  return 0;
}
```