Programming and Algorithms

C: Input/output. Files. Using argc, argv.

Jean-Bernard Hayet

CIMAT, A.C.

Files

Handling files

The C language allows to open files/read files/write into files.

- Uses a series of specific functions: fopen, fclose, fread and makes use of a structure **FILE**.
- All are declared in <stdio.h> (Caution: <conio.h> is not part of the standard).
- The FILE structure keeps information about the point of access, the buffer, ...

Handling files

I/O functions use a **buffer mechanism**:

- Writing physically in a file (hard drive) is **slow** (system calls, works in blocks), in particular for small amounts of data.
- Writing in memory is **much faster**.
- Idea is to delay the writing on disk to avoid losing time.

To handle the I/O, the address of this memory buffer should be known, altogether with the read head: This is what the FILE structure contains. A pointer to a FILE is a **file stream**.

Handling files: fopen

FILE *fopen(fileNam e , accessMode) ;

- The fopen function initializes a file stream from a **file name** (file on the disk, interface port, . . .).
- Needs two strings as arguments: filename/access mode.
- Returns the **open file stream if OK, NULL otherwise**.

Handling files: fopen

"r"	File open as a text, in reading mode.
"w"	File open as a text, in writing mode (new file).
"a"	File open as a text, in append mode. Output at end.
"rb"	File open as binary, in reading mode.
"wb"	File open as binary, in writing mode.
"ab"	File open as binary, in append mode.
"r+"	File open as a text, in read/update mode.
"w+"	File open as a text, in write/update mode.
"a+"	File open as a text, in append/update mode at end.
"r+b"	File open as binary, in read/update mode.
"w+b"	File open as binary, in write/update mode.
"a+b"	File open as binary, in append/update mode at end.

Handling files: fopen

- **Binary**: no notion of line, not for formatted data, raw data (fwrite/fread).
- **Text**: all data interpreted as characters (and '$\backslash n$' as a line separator).
- In **update** mode, input and output may be both performed but writes that follow reads imply a call to fflush or to a file positioning function (fseek, fsetpos, or rewind).

```
> man fopen
```

Handling files: fopen

```
#include <stdio.h>
FILE *fp;
 ...
 if  ((fp=fopen("data.txt","r"))==NULL) {
          fprintf (stderr ,"Cannot open  file\n");
          exit ( 1 )  ;
}
```

Handling files

When running a program, three streams are open by default:

- **stdout** (standard output) : terminal, by default,
- **stdin** (standard input) : keyboard, by default,
- **stderr** (standard output error) : terminal, by default.

By the pipe mechanism, streams can be redirected from one to another.

Handling files: fclose

Closes an open stream:

```
fclose(fp);
```

Return 0 if everything OK.

Formatted output

Formatted writing into a file:

```
float  a,b;
int  i;
fprintf (fp,"%f %f %d",a,b,i) ;
```

Formatted reading from a file:

```
float  a,b;
int  i;
fscanf(fp,"%f %f %d",&a,&b,&i);
```

The **printf** command is equivalent to fprintf(stdout,...) and **scanf** to fscanf(stdin,..)

Handling files

The **fflush** command empties the buffer to target file.

```
fflush (fp) ;
```

Maybe useful when we want to ensure that data are written to the target file while the program runs (and if we do not care about writing times).

Handling files

If the given stream was open for writing and the last i/o operation was an output operation, any unwritten data in the output buffer is written to the file.
If the stream was open for reading, the behavior depends on the specific implementation. In some implementations this causes the input buffer to be cleared.
If the argument is a null pointer, all open files are flushed. The stream remains open after this call.
When a file is closed, either because of a call to fclose or because the program terminates, all the buffers associated with it are automatically flushed.

Handling files: Classic bug

```
#include <stdio.h>
int  main() {
        char  x,y;
         printf ("First :  ");
        scanf("%c",&x);
         printf ("Second :  ");
        scanf("%c",&y);
         printf ("\n\n%c,%c",x,y);
        return  0;
}
```

Handling files: Classic bug

```
#include <stdio.h>
int main() {
        char x,y;
         printf ("First : ");
        scanf("%c",&x);
        while ((c = fgetc(stdin))!= '\n') {}
         printf ("Second : ");
        scanf("%c",&y);
         printf ("\n\n%c,%c",x,y);
        return 0;
}
```

Handling files

```
char  buffer[BUFSIZ];
setbuf(fp1, buffer );
setbuf(fp2,NULL);
```

To have more control on the buffer memory or choose not to have (NULL).

Handling files: I/O for single characters

```
// Get a character
int fgetc(FILE* stream);
// Put a character
int fputc(int character,FILE* stream);
// Push back a character in the stream
int ungetc(int character,FILE* stream);
```

Why int?

`fgetc() reads the next character from stream and returns it unsigned char cast to an int, or EOF on end of file or error.`

Handling files: I/O for single characters

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
        FILE* fin;
        if ((fin = fopen("data.txt","rb")) == NULL) {
                fprintf (stderr,"\n Error while trying to open file");
                return (1);
        }
        int c;
        while ((c=fgetc(fin))!=EOF)
                fputc(c,stdout);
        fclose (fin);
        return (0);
}
```

Handling files: I/O for single characters

```
#include <stdio.h>
int main() {
        FILE* pFile;
        pFile = fopen("data.txt","rb");
        if (pFile==NULL) perror("Error opening file");
        else {
                char buffer[256];
                while (!feof(pFile)) {
                        int c=getc(pFile);
                        if (c=='#') ungetc('@',pFile);
                        else ungetc(c,pFile);
                        fgets(buffer,255,pFile);
                        fputs(buffer,stdout);
                }
        }
}
```

Handling files: fread/fwrite

Read/write without representations (raw)

```
size_t fread(void* ptr,size_t s,size_t num,FILE* stream);
size_t fwrite(void* ptr,size_t s,size_t num,FILE* stream);
```

Arguments: Pointer to the data; Number of bytes of each data; Number of data; File stream.

Handling files: fread/fwrite

Reading the header of a MIDI file.

```
char  firstchain [4];
// Read first 4 bytes
fread(& firstchain [0],1,4, f) ;
if ( firstchain [0]!='M'|| firstchain [1]!='T' ||
          firstchain [2]!='h' || firstchain [3]!='d') {
          fprintf (stderr ,"This is not a MIDI header") ;
          return 1 ;
}
```

Handling files: Moving within a file stream

```
int fseek(FILE* stream, long displacement, int ref);
```

To move of a given number of bytes in the stream starting from a reference.
Possible references:

- SEEK_SET (start),
- SEEK_CUR (current)
- SEEK_END (end)

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

To start again, nearly equivalent to:

```
fseek(stream,0L,SEEK_SET);
```

Handling files

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

Returns the value of the position: number of bytes in case of binary files, nothing guaranteed in case of text files.

Handling files

```
/* fread example: read a complete file */
#include <stdio.h>
#include <stdlib.h>
int main() {
        FILE* pFile = fopen("myfile.bin", "rb");
        if (pFile==NULL) { fputs("File error", stderr); exit(1);}
        // obtain file size:
        fseek(pFile,0,SEEK_END);
        long lSize = ftell(pFile);
        rewind(pFile);
```

Handling files

```
    // allocate memory to contain the whole file :
    char *buffer = (char *)malloc( sizeof(char)*lSize );
    if ( buffer == NULL) { fputs("Memory error",stderr); exit(2);}
    // copy the file into the buffer :
    size_t result = fread( buffer ,1, lSize , pFile );
    if ( result !=lSize) { fputs("Reading error ", stderr ); exit (3);
    // the whole file is now loaded in the memory buffer
    // terminate
    fclose ( pFile ) ;
    free ( buffer ) ;
    return 0 ;
}
```

Redirection and pipes

Redirection

A feature in C which is inherited from the development of Unix is the **redirection** of file streams.

Redirecting a file stream means associating a currently open file stream (which is then closed) to another file.

```
FILE *freopen(const char *fname, const char *mode, FILE *stream);
```

mode is the sane as with fopen function.

Redirection

```
int main() {
        FILE *fp;
        printf("This is displayed at scree\n");

        if ((fp=freopen("out.txt", "w", stdout))==NULL) {
                printf("Error in opening the file \n");
                exit(1);
        }
        printf("This is written to the file out.txt\n");
        fclose(fp);
        return 0;
}
```

Usage is mainly to re-direct the standard input/outputs.

Pipes

Another feature which comes from the development of Unix is the **pipe** mechanism.

As its name says, it is a channel of communication which can be seen as a **pair of file streams**: o**ne input and one output**.

It works as a **queue**.

It may not be very useful in a simple program (write in one side, read on the other, what for?)

Pipes

```
int pipe(int fds[2]);
```

The two elements of the array are **file descriptors** (much lower level than FILE).

Pipes

```
char* msg1 = "Hey Jude";
char* msg2 = "Don't make it bad";
char rcv[100];
int main()
{
        int p[2];
        if (pipe(p) < 0) exit(1);
        write(p[1], msg1, strlen(msg1));
        write(p[1], msg2, strlen(msg2));
        read(p[0], rcv, strlen(msg1));
        printf("%s\n", rcv);
        read(p[0], rcv, strlen(msg2));
        printf("%s\n", rcv);
        return 0;
}
```

Pipes

- **Pipes** are more useful in the case of multiple processes program: they allow a communication between processes!
- In Unix: fork() system calls makes a child process from a parent process.

```
int main()
{
        // Fork call : first block is what the parent process does
        if ((pid = fork()) > 0) {
                ...
                // Waits for the child to end
                wait(NULL);
        }
        // The child process
        else {
                ...
        }
        return 0;
}
```

Pipes

```
int main()
{
        int p[2], pid, nbytes;
        if (pipe(p) < 0)
        exit (1);
        if ((pid = fork()) > 0) {
                write(p[1], msg1, strlen(msg1));
                write(p[1], msg2, strlen(msg2));
                close(p[1]);
                wait(NULL);
        }
        else {
                close(p[1]);
                while ((nbytes = read(p[0], rcv, 10)) > 0)
                printf("%s\n", rcv);
                if (nbytes != 0)
                exit (2);
        }
```

arc/argv

argc/argv

The main() function may have several prototypes:

- Returning void/int (for notifying errors to the system).
- With/without arguments.

In the case of handling arguments: those are an int and an array of strings.

argc/argv

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

The idea is that **argv contains all the command line that lanch the program**, and argc is the number of strings that make this command line.

The command line is separated into strings: This allows to **pass options or arguments to the program (file names...) interactively**.

argc/argv

```
./myProgram toto.txt 1 120
```

In that case, argc is **four** and argv is made of 4 strings. Useful:

```
int  atoi (const char * str );
double atof (const char* str );
```