File Handling

CS10003 PROGRAMMING AND DATA STRUCTURES

What is a file?

A named collection of data, stored in secondary storage (typically).

Typical operations on files:

- Open
- Read
- Write
- Close

How is a file stored?

- Stored as sequence of bytes, logically contiguous (need not be physically contiguous on disk).
- C gives us a simplified view of a file stored on the disk.

File Types

Two kinds of files

- **Text :: Human-readable files**
  - **A text editor can show the contents of a text file.**
- **Binary :: Not meant for direct human reading**
  - **Image, audio, video, executable, etc.**
  - **You need special programs (like an image viewer or a multimedia player or a program runner to read (and process) binary files meaningfully.**

File handling in C

In C, we use `FILE*` to represent a pointer to a file.

`fopen()` is used to open a file. It returns the special value `NULL` to indicate that it is unable to open the file.

```
FILE *fptr;
char filename[ ]= "file2.dat";

fptr = fopen (filename,"w");

if (fptr == NULL) {
   printf ("ERROR IN FILE CREATION");
   /* do something */
}
```

Modes for opening files

The second argument of `fopen` is the *mode* in which we open the file. There are three basic modes.

"r" opens a file for reading.

- **"r+" allows write**

"w" creates a file for writing and writes over all previous contents (deletes any previous file of the same name, so be careful!).

- **"w+" allows read**

"a" opens a file for appending – writing at the end of the file (previous contents are kept intact).

- **"a+" allows read**

The exit( ) function

Sometimes error checking means we want an *immediate exit* from a program.

In `main()`, we can use `return` to stop the execution of the program.

In any function, we can use `exit()` to do this.

`exit()` is declared in the header file `stdlib.h`.

```
exit(0);    // exit the program successfully
exit(1);
exit(2);
exit(3);
...                                  // type of
unsuccessful termination
```

"*All happy families are alike; each unhappy family is unhappy in its own way.*"

– Leo Tolstoy, Anna Karenina

Usage of exit( )

```
FILE *fptr;
char filename[ ]= "file2.dat";
fptr = fopen (filename,"w");

if (fptr == NULL) {
  printf ("ERROR IN FILE CREATION\n");
  exit(0);
}
```

Writing to a file using fprintf( )

`fprintf()` works just like `printf()`

except that its first argument is a file pointer.

```
int a = 10, b = 5;
FILE *fptr;
fptr = fopen ( "file.dat", "w" );

fprintf (fptr, "Hello World!\n");
fprintf (fptr, "%d %d", a, b);
```

Reading data from a file using fscanf( )

```
int x, y;
FILE *fptr;
fptr = fopen ("input.dat", "r");

fscanf (fptr, "%d%d", &x, &y);
```

The C library maintains a 'file pointer' to remember the position up to which a file has been read so far. The file pointer moves forward with each read operation.

Next read operation (e.g., `fscanf, fgets, fgetc`) will give the contents of the file after this position.

Each function for reading from a file has a way to inform that the end of file has been reached (usually by returning a special value like `NULL` or `EOF`)

EOF (End of File)

`EOF` is a special value that signifies that the file pointer has reached the end of the file stream.

`EOF` is returned by `fgetc()` and `fscanf()` if the end of file has been reached.

Another way to detect the end of a file is to use the call `feof(fp)`.

`feof()` returns true only after a read operation from the file fails.

`feof()` cannot probe and notify that the next read operation will fail. This is oftentimes not possible because whether the end-of-file is reached or not depends on what you plan to read next (like an int or a char).

Example of “end of file”

The task is to read the integers stored in `input.txt` one by one, add the integers read, and print the sum.

After the last integer 5 is read:

The file contains no more integers.

The file contains a few more characters.

So a %d reading at this stage will fail, but a %c reading will not.

You need to attempt a reading, and then check for `EOF` or `feof()`.

Input file `input.txt`

```
45 2 33
9
11 24

17
5
```

There are a few blank lines at the end.

Checking end-of-file using EOF

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  int n, sum, x;
  fp = (FILE *)fopen("input.txt", "r");
  if (fp == NULL) exit(1);
  n = sum = 0;
  while (1) {
     if (fscanf(fp, "%d", &x) == EOF) break;
     ++n; sum += x;
     printf("%d-th integer: %d\n", n, x);
  }
  fclose(fp);
  printf("Sum of the integers read is %d\n", sum);
  exit(0);
}
```

Output

```
1-th integer: 45
2-th integer: 2
3-th integer: 33
4-th integer: 9
5-th integer: 11
6-th integer: 24
7-th integer: 17
8-th integer: 5
Sum of the integers read is 146
```

Checking end-of-file using feof( )

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  int n, sum, x;
  fp = (FILE *)fopen("input.txt", "r");
  if (fp == NULL) exit(1);
  n = sum = 0;
  while (1) {
     fscanf(fp, "%d", &x);
     if (feof(fp)) break;
     ++n; sum += x;
     printf("%d-th integer: %d\n", n, x);
  }
  fclose(fp);
  printf("Sum of the integers read is %d\n", sum);
  exit(0);
}
```

Output

```
1-th integer: 45
2-th integer: 2
3-th integer: 33
4-th integer: 9
5-th integer: 11
6-th integer: 24
7-th integer: 17
8-th integer: 5
Sum of the integers read is 146
```

Checking end-of-file using feof( ): Wrong program

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  int n, sum, x;
  fp = (FILE *)fopen("input.txt", "r");
  if (fp == NULL) exit(1);
  n = sum = 0;
  while (!feof(fp)) {
     fscanf(fp, "%d", &x);
     ++n; sum += x;
     printf("%d-th integer: %d\n", n, x);
  }
  fclose(fp);
  printf("Sum of the integers read is %d\n", sum);
  exit(0);
}
```

Output

```
1-th integer: 45
2-th integer: 2
3-th integer: 33
4-th integer: 9
5-th integer: 11
6-th integer: 24
7-th integer: 17
8-th integer: 5
9-th integer: 5
Sum of the integers read is 151
```

Reading data from a file using fgets( )

We can read a string from a file using `fgets()`.

`fgets()` takes 3 arguments – a string, maximum number of characters to store in the string, and a file pointer. It returns `NULL` if there is an error (or end of file is reached).

```
FILE *fptr;
char str [1000];
...                  /* Open file and check it is open */
...
while ( fgets(str, 1000, fptr) != NULL )
/* Read 999 chars at most at a time */
{
   printf ("We have read the string: %s\n", str);
}
```

Reading data from a file using fgets( )

A maximum of size – 1 bytes will be read from the input file stream.

The reading includes the new line character if it appears in these many bytes.

`fgets()` null-terminates the string by putting the null character '\0'.

With appropriate size, `fgets()` never leads to buffer overflow.

In the example:

- If the line contains at most 998 characters, the entire line and the new-line character will be read and stored in the string.
- If the line contains 999 or more characters, only the first 999 characters will be read and stored in the string.

Closing a file

We can close a file simply using `fclose()` and the file pointer.

```
FILE *fptr;
char filename[ ]= "myfile.dat";

fptr = fopen (filename,"w");

if (fptr == NULL) {
    printf ("Cannot open file to write!\n");
    exit(0);
}

fprintf (fptr,"Hello World of filing!\n");
fclose (fptr);
```

Three special file streams

Three special file streams are defined in the header file `<stdio.h>`. These FILE pointers are automatically opened in every program.

- `stdin` reads input from the keyboard
- `stdout` send output to the screen
- `stderr` prints errors to an error device (usually also the screen)

`scanf(...)` is the same as `fscanf(stdin, ...)`

`printf(...)` is the same as `fprintf(stdout, ...)`

An example program

```
#include <stdio.h>
main( )
{
      int i;

      fprintf(stdout,"Give value of i \n");
      fscanf(stdin,"%d",&i);
      fprintf(stdout,"Value of i=%d \n",i);
      fprintf(stderr,"No error: But an example to show error message.\n");
 }
```

Output

Give value of i
15
Value of i=15
No error: But an example to show error message.

Reading and Writing a character

A character reading/writing is equivalent to reading/writing a byte.

```
int getchar( );
int putchar(int  c);
```

} stdin, stdout

```
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
```

} file

Example:

```
char c;
c = getchar();
putchar(c);
```

```
char c; FILE *fp;
c = fgetc(fp);
fputc(c,fp);
```

Random access using fseek( )

`ftell()` returns the present position of the file pointer

```
long ftell(FILE *fp)
```

`fseek()` can be used to set the position of a file pointer (say, fp).

```
int fseek(FILE *fp, long offset, int from_where)
```

New position of file pointer specified by 2 more arguments – `offset` (specified in bytes) and `whence`. The field `from_where` can take one of 3 values:

- `SEEK_END` end of the file
- `SEEK_SET` beginning of the file
- `SEEK_CUR` current position of the file pointer

Example – fseek( ) and ftell( )

```
int main(){
  char c; FILE *fp;
  fp=fopen("file.txt", "r+");
  printf("%ld \n", ftell(fp));
  c = fgetc(fp); c = fgetc(fp);
  printf("%ld \n", ftell(fp));
  fseek(fp, 2, SEEK_CUR);
  printf("%ld \n", ftell(fp));
  fputs("fast purple",fp);
  printf("%ld \n", ftell(fp));
  fclose(fp);
  return 0;
}
```

Output:

0
2
4
15

Contents of file.txt

Before: the quick brown fox jumped over the lazy dogs

After: the fast purple fox jumped over the lazy dogs

Another example of fseek( )

- The file `primes.txt` stores the first 1000 primes, one in each line with no extra spaces.
- We open the file in the read mode.
- We do not read the file from beginning to end.
- When the user specifies some n in the range [1,1000], we go to the location where the n-th prime is stored, and read that prime.
- 2 is the first prime (not the zero-th prime).
- We need to know exactly where the n-th prime is stored.
- The following statistics help us do that.
  - There are exactly four 1-digit primes.
  - There are exactly 21 2-digit primes.
  - There are exactly 143 3-digit primes.
  - The 1000-th prime is a 4-digit prime.
- We must not forget the new line character at the end of each line.

The file primes.txt

```
2
3
5
7
11
13
...
97
101
103
107
...
997
1009
...
7919
```

The program for seeking primes

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
   FILE *fp;
   int n, p;
   fp = (FILE *)fopen("primes.txt", "r");
   while (1) {
      printf("Which prime? "); scanf("%d", &n);
      if ((n < 1) || (n > 1000)) break;
      if (n <= 4) fseek(fp, (n - 1) * 2, SEEK_SET);
      else if (n <= 25) fseek(fp, 4 * 2 + (n - 5) * 3, SEEK_SET);
      else if (n <= 168) fseek(fp, 4 * 2 + 21 * 3 + (n - 26) * 4, SEEK_SET);
      else fseek(fp, 4 * 2 + 21 * 3 + 143 * 4 + (n - 169) * 5, SEEK_SET);
      fscanf(fp, "%d", &p);
      printf("%d-th prime is %d\n", n, p);
   }
   fclose(fp);
   exit(0);
}
```

Sample Output

```
Which prime? 25
25-th prime is 97
Which prime? 26
26-th prime is 101
Which prime? 168
168-th prime is 997
Which prime? 169
169-th prime is 1009
Which prime? 1000
1000-th prime is 7919
Which prime? -1
```

Example: Program for Copying a File

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

int main( )
{
      FILE *ifp, *ofp;
      int i, c;
      char src_file[100], dst_file[100];

      strcpy (src_file, "source.txt");
      strcpy (dst_file, "copy.txt");

     /* continued in the next slide … */
```

Example: contd.

```
  if ( (ifp = fopen(src_file,"r")) == NULL )   {
     printf ("Input File does not exist.\n");    exit(0);
  }
  if ( (ofp = fopen(dst_file,"w")) == NULL )   {
     printf ("Output File not created.\n");    exit(0);
  }

  while ( (c = fgetc(ifp)) != EOF )
          fputc (c,ofp); // This is where the copying is done

  fclose(ifp);
  fclose(ofp);
}
```

Example of creating and reading binary files

We want to store the Fibonacci numbers F(0), F(1), F(2), . . . , F(40) in a file.

Text mode

We store 0, 1, 1, 2, 3, . . . , 102334155, one in a single line.

Some separators are needed between to consecutive Fibonacci numbers (here we use '\n').

fseek() to locate F(i) will be difficult.

Binary mode

Assume that int is of size 32 bits (4 bytes).

We store the raw 32 bits of each F(i) one after another.

No separators are needed because each F(i) occupies exactly 4 bytes.

fseek() to locate F(i) will be easy: just go to the 4i-th byte from the beginning.

Storing in the text (human-readable) format

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  int n = 40, i, F[41];
  F[0] = 0; F[1] = 1;
  for (i=2; i<=n; ++i) F[i] = F[i-1] + F[i-2];
  fp = (FILE *)fopen("Fib.txt", "w");
  if (fp == NULL) exit(1);
  for (i=0; i<=n; ++i) fprintf(fp, "%d\n", F[i]);
  fclose(fp);
  exit(0);
}
```

Output file Fib.txt

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
. . .
39088169
63245986
102334155
```

Storing in the binary format

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  int n = 40, i, j, F[41];
  char *p;
  F[0] = 0; F[1] = 1;
  for (i=2; i<=n; ++i) F[i] = F[i-1] + F[i-2];
  fp = (FILE *)fopen("Fib.dat", "w");
  if (fp == NULL) exit(1);
  for (i=0; i<=n; ++i) {
     p = (char *)(F + i);
     for (j=0; j<4; ++j) { fprintf(fp, "%c", *p); ++p; }
  }
  fclose(fp);
  exit(0);
}
```

Try to open Fib.dat in a text editor

Fib.txt takes 220 bytes
Fib.dat takes 164 bytes

The binary mode often reduces storage space.

Reading in the binary format

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  int n = 40, F[41], i, j;
  FILE *fp;
  char *p;
  fp = (FILE *)fopen("Fib.dat", "r");
  for (i=0; i<=n; ++i) {
     p = (char *)(F+i);
     for (j=0; j<4; ++j) { fscanf(fp, "%c", p); ++p; }
  }
  fclose(fp);
  for (i=0; i<=n; ++i) printf("F(%d) = %d\n", i, F[i]);
  exit(0);
}
```

Output

```
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
F(11) = 89
F(12) = 144
. . .
F(38) = 39088169
F(39) = 63245986
F(40) = 102334155
```

Practice Problems

1. Write a program that reads a file, converts all lower-case letters to the upper case, and keeps the other characters intact, and stores the output in another file.
2. Write a program that reads a 2-d array of integers from a file and replaces the contents of the file with the transpose of the matrix represented by the 2-d array.
3. Write a program that reads student records containing name (string), roll_number (int), CGPA (float) from the user and writes them in a file, one record per line.
4. Write a program that reads the records written by the above program into an array of structures. The structure should contain name, roll_number and CGPA as members.
5. Write a program that takes as a command line argument a C program filename and outputs the number of occurrences of the keywords int, float, double, long, short in the file.

Advanced topics

Input and Output redirection to files

Input File and Output File redirection

One may redirect the standard input and standard output to other files (other than `stdin` and `stdout`).

Usage: Suppose the executable file is `a.out`:

```
$ ./a.out < in.dat  > out.dat
```

`scanf()` will read data inputs from the file "`in.dat`", and
`printf()` will output results on the file "`out.dat`".

A Variation

```
$ ./a.out < in.dat  >> out.dat
```

`scanf()` will read data inputs from the file "`in.dat`", and `printf()` will append results at the end of the file "`out.dat`".

Command Line Arguments

What are they?

A program can be executed by directly typing a command at the shell prompt.

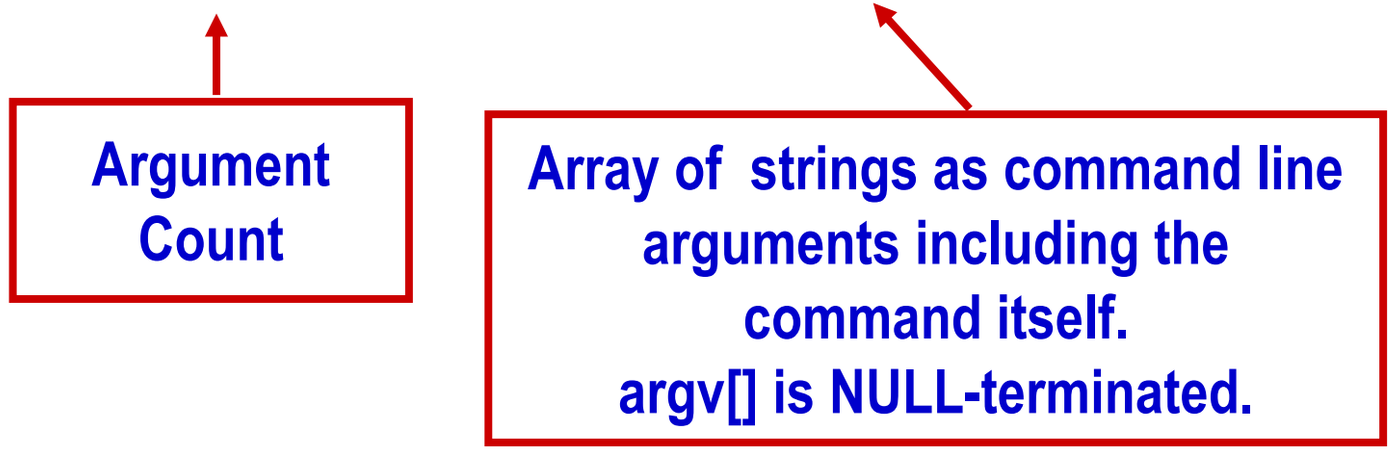
```
$ gcc test.c
$ ./a.out in.dat out.dat
$ prog_name param_1 param_2 param_3 ..
```

- The individual items specified are separated from one another by spaces. Use quotes to enter arguments with spaces.
  - First item is the program name.
- Variables ***argc*** and ***argv*** keep track of the items specified in the command line.

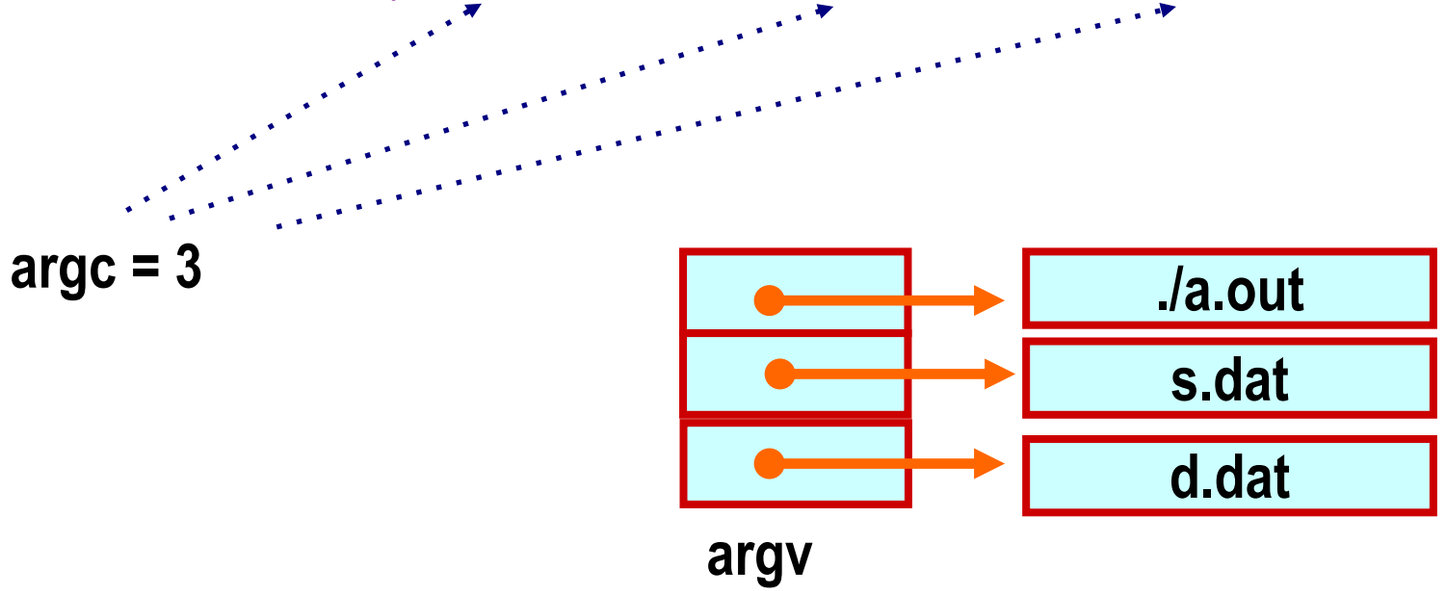
How to access them?

Command line arguments may be passed by specifying them under `main()`.

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



```
$  ./a.out  s.dat  d.dat
```



argv[0] = "./a.out" argv[1] = "s.dat" argv[2] = "d.dat" argv[3] = NULL

Example: Copying a file with command-line arguments

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

int main( int argc, char *argv[ ] )  {
      FILE *ifp, *ofp;
      int i, c;
      char src_file[100], dst_file[100];
      if (argc!=3) {
            printf ("Usage: ./a.out <src_file> <dst_file> \n");
                  exit(0);
      }
      else {
            strcpy (src_file, argv[1]);
                  strcpy (dst_file, argv[2]); // using cmd line args
      }

      /* continued to the next slide … */
```

Example: contd.

```
   if ((ifp = fopen(src_file,"r")) == NULL)   {
      printf ("Input File does not exist.\n");   exit(0);
   }

   if ((ofp = fopen(dst_file,"w")) == NULL)   {
      printf ("Output File not created.\n");   exit(0);
   }

   while ((c = fgetc(ifp)) != EOF)   fputc (c,ofp);
      // This is where the copying is done

   fclose(ifp);  fclose(ofp);
}
```

Converting arguments to other data types

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

Use `atoi()`, `atol()`, `atof()` to convert an argument to `int`, `long`, or `double`.

Example:

You run

```
./a.out "Foolan Barik" 22FB14641 9.31 178
```

Inside the code:

```
strcpy(name, argv[1]);                  /* Copying to string */
strcpy(roll, argv[2]);                  /* Copying to string */
CGPA = atof(argv[3]);                   /* Converting to double */
gradepointer = argv[3][0];              /* Taking a single character */
height = atoi(argv[4]);                 /* Converting to int */
```