CMSC 104 - Lecture 26
Park , adapted by C Grasso

File I/O: Part 1 of 2

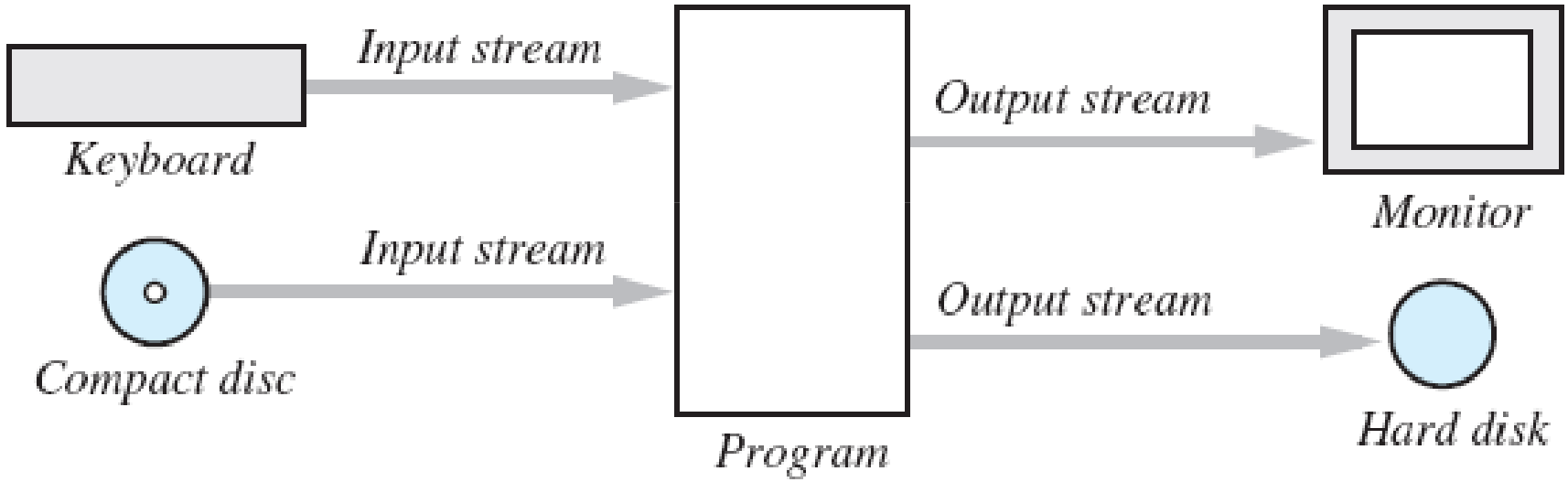
Overview

- Files & Streams
- Basic File I/O Algorithm
- FILE modes
- Three Special Streams
- Input/Output Operations on Files
- I/O Errors
- Command line arguments

The Concept of a Stream

- Use of files
  - Store programs
  - Store pictures, music, videos
  - Can also use files to store program I/O
- A *stream* is a flow of input or output data
  - Characters
  - Numbers
  - Bytes

The Concept of a Stream



Why Use Files for I/O

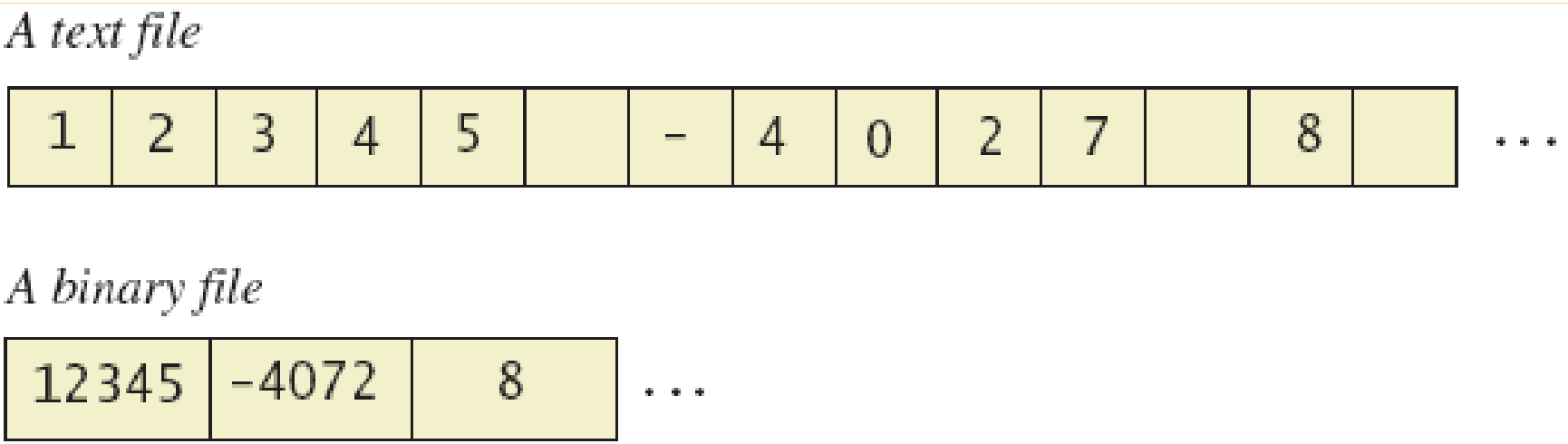
- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

Text Files and Binary Files

- All data in files is stored in binary
  - Long series of zeros and ones
- Files that are a sequence of characters are called *text files*
  - Program source code
  - Can be viewed, edited with text editor
- All other files are called *binary files*
  - Movie, music files
  - Executables
  - Access requires specialized program

Text Files and Binary Files

- Figure 10.2 A text file and a binary file containing the same values



Algorithm for Using a File

- Declare a file variable
- Associate the variable with a file on the disk
- Open the file
- Use the file
- Close the file

FILE structure

- Everything that the C programming environment needs to know about a file is contained in a data structure called FILE.
  - It is the link between your program and the OS
- FILE * represents a pointer to a file.
- fopen() is used to open a file.
  - It returns a **FILE *** if the call was successful.
  - It returns **NULL** to indicate that it couldn't open the file.

File handling in C

```
// Specify the name & path of the file
char filename[]= "file2.dat";

// Open the file for output
// Return the ptr to the file structure
FILE *fptr = fopen (filename,"w");

// If the file wasn't opened, quit
if (fptr == NULL) {
     fprintf (stderr, "ERROR");
     exit(-1);
}
```

Modes for opening files

- The second argument of `fopen` is the *mode* in which we open the file.
  - "r" - reading
  - "w" - writing
    - overwrites all previous contents
    - creates a new file if it does not already exist
  - "a" - appending
    - adds on to the end of the file

The exit() function

- Sometimes error checking means we want an "emergency exit" from a program. We want it to stop dead.
  - In main we can use **`return`** to stop.
  - In functions we can use **`exit()`** to do this.
    - Exit is part of the `stdlib.h` library

Getting out Quickly

From main() routine

```
return -1;
```

From a function

```
exit(-1);
```

They are exactly the same.

Closing a File

- File must be closed as soon as all operations on it completed

```
fclose (fptr );
```

- Ensures
  - All outstanding information associated with file flushed out from buffers
  - All links to file broken
  - Accidental misuse of file prevented
- If want to change mode of file, then first close and open again

Writing to a File using fprintf()

- fprintf() works just like printf() and sprintf() except that its first argument is a file pointer.

```
FILE *fptr;
fptr= fopen ("file.dat","w");
if (fptr != NULL)
      fprintf (fptr,"Hello World!\n");
```

- We could also read numbers from a file using fscanf() – but there is a better way.

Reading from a file using fgets

- fgets() is a better way to read from a file
  - We can read into a string using fgets()
- fgets() takes 3 arguments
  - a string
  - a maximum number of characters to read
  - a file pointer
- It returns NULL if there is an error (such as EOF)

Reading from a file using fgets

```
FILE *fptr = fopen (filename,"w");

#define SIZE 1000

char line [SIZE];

char *line = fgets(line,SIZE,fptr);

fclose(fptr);
```

Reading loops

- It is quite common to want to read every line in a file.
- The best way to do this is a while loop using fgets()

Reading loops

```
#define MAXLEN 1000
char tline[MAXLEN];  /* A line of text */

FILE *fptr = fopen ("sillyfile.txt","r");

/* check if it's open or at EOF */
while (fgets (tline, MAXLEN, fptr) != NULL)
{
    printf ("%s",tline); // Print it
}

fclose (fptr);
```

Three special streams

- Three special file streams are defined in the `stdio.h` header
  - **`stdin`** reads input from the keyboard
  - **`stdout`** send output to the screen
  - **`stderr`** prints errors to an error device

- What might this do ?:

```
fprintf (stdout,"Hello World!\n");
```

Using fgets to read from the keyboard

- fgets() and stdin can be combined to get a safe way to get a line of input from the user

```
#include <stdio.h>
int main()
{
    const int MAXLEN=1000;
    char readline[MAXLEN];
    fgets (readline,MAXLEN,stdin);
    printf ("You typed %s",readline);
    return 0;
}
```

Input/Output operations on files

- C provides several different functions for reading/writing
  - getc() – read a character
  - putc() – write a character
  - fprintf() – write set of data values
  - fscanf() – read set of data values
  - getw() – read integer
  - putw() – write integer

getc() and putc()

- Handle one character at a time.
- c = getc(fp2); putc(c,fp1);
  - c : a character variable
  - fp2 : pointer to open file
- File pointer moves by one character position after every getc() and putc()
- getc() returns end-of-file marker EOF when file end reached

Program to read on char at a time

```
#include <stdio.h>
main()
{
    char c;
     FILE *f1 =fopen(“INPUT.txt”, “r”);  /* open file */

    while((c=getc(f1))!=EOF)     /*read char from file INPUT*/
            printf(“%c”, c);          /* print character to screen */

    fclose(f1);

} /*end main */
```

Getting numbers from strings

- Once we've got a string with a number in it (either from a file or from the user typing) we can use the C library to convert it to a number
  - `atoi()`
  - `atof()`
- The functions are part of `stdlib.h`
- Both of these functions return 0 if they have a problem

Getting numbers from strings

```
char *numberstring = "3.14";

double pi;
pi= atof (numberstring);

int i;
i= atoi ("12");
```

Errors that occur during I/O

- Typical errors that occur
  - trying to read beyond end-of-file
  - trying to use a file that has not been opened
  - perform operation on file not permitted by 'fopen' mode
  - open file with invalid filename
  - write to write-protected file

Error handling

- int feof (FILE *fptr)
  - check if EOF reached
  - takes file pointer as input
  - returns zero if there is more data to read in the file

```
if  (  feof(fp) != 0 )
   printf("End of data\n");
```

Error handling

- int ferror(FILE *fptr)
  - check if an error has occurred
  - takes file-pointer as input
  - returns nonzero integer if an error was detected else returns zero

```
if( ferror(fp) !=0 )
   printf("An error has occurred\n");
```

Error while opening file

- If the file cannot be opened then fopen() returns a NULL pointer
  - Good practice to check if pointer is NULL before proceeding

```
fp = fopen(“input.dat”, “r”);

if (fp == NULL)
  printf(“File could not be opened”);
```

Command line arguments

- Can give input to C program from command line

```
$>    */prog  10   name1    name2 ....
```

- main (int argc, char *argv[])
  - **int argc** – gives a count of number of arguments
  - **char *argv[]** - defines an array of strings
  - argv[0] – program name
  - argv[1] to argv[argc -1] give the other arguments as strings

Example args.c

```
#include <stdio.h>

main(int argc,char *argv[])
{
  int i;
  for(i=0; i < argc; i++) /*print out args*/

      printf("%s\n", argv[i]);
}
```

Example args.c

```
~/C/ > gcc -o args args.c
~/C/ > ./args these are my arguments

./args
these
are
my
arguments
```