CS1010 http://www.comp.nus.edu.sg/~cs1010/

Programming Methodology

UNIT 19

File Processing

Unit 19: File Processing

Objectives:

- Understand the concepts of file I/O
- Learn about functions to read and write text files

Reference:

- Section 6.1 (Pointers to Files)
- Chapter 11 Text and Binary File Pointers

Unit 19: File Processing

1. Introduction (1/4)

- Problems on arrays usually involve a lot of data, so it is impractical to enter the data through the keyboard.
- We have been using the UNIX input file redirection < to redirect data from a text file. Eg: a.out < data1
- However, that is not a C mechanism. C provides functions to handle file input/output (I/O).
- We will focus on these basic file I/O functions on text files:

```
fopen()
fclose()
fscanf()
fprintf()
```

1. Introduction (2/4)

- In C, input/output is done based on the concept of a stream
- A stream can be a file or a consumer/producer of data

Monitor

Keyboard

Network port

Hard disk

Printer

1. Introduction (3/4)

- A stream is accessed using file pointer variable of type **`FILE`** *
- The I/O functions/macros are defined in stdio.h
- Two types of streams: text and binary
- We will focus on text stream:
  - Consists of a sequence of characters organized into lines
  - Each line contains 0 or more characters followed by a newline character '\n'
  - Text streams stored in files can be viewed/edited easily using a text editor like vim

1. Introduction (4/4)

- 3 standard streams are predefined:
  - stdin points to a default input stream (keyboard)
  - stdout points to a default output stream (screen)
  - stderr points to a default output stream for error messages (screen)
- `printf()` writes output to stdout
- `scanf()` reads input from stdin
- The 3 standard streams do not need to be declared, opened, and closed
- There are 2 useful constants in file processing
  - **NULL**: null pointer constant
  - **EOF**: used to represent end of file or error condition

Note that null pointer **NULL** is **not** the null character **'\0'** !

2. Demo: Sum Array (1/6)

Unit19_SumArray.c

```
#include <stdio.h>
#define MAX 10   // maximum number of elements

int scanPrices(float []);
float sumPrices(float [], int);
void printResult(float);

int main(void) {
   float prices[MAX];
   int size = scanPrices(prices);
   printResult(sumPrices(prices, size));
   return 0;
}

// Compute sum of elements in arr
float sumPrices(float arr[], int size) {
   float sum = 0.0;
   int i;
   for (i=0; i<size; i++)
      sum += arr[i];
   return sum;
}
```

2. Demo: Sum Array (2/6)

Unit19_SumArray.c

```
// Read number of prices and prices into array arr.
// Return number of prices read.
int scanPrices(float arr[]) {
   int size, i;

   printf("Enter number of prices: ");
   scanf("%d", &size);

   printf("Enter prices:\n");
   for (i=0; i<size; i++)
      scanf("%f", &arr[i]);

   return size;
}

// Print the total price
void printResult(float total_price) {
   printf("Total price = $%.2f\n", total_price);
}
```

2. Demo: Sum Array (3/6)

Unit19_SumArray_with_Files.c

```c
#include <stdio.h>
#define MAX 10   // maximum number of elements

int scanPrices(float []);
float sumPrices(float [], int);
void printResult(float);

int main(void) {
   float prices[MAX];
   int size = scanPrices(prices);
   printResult(sumPrices(prices, size));
   return 0;
}

// Compute sum of elements in arr
float sumPrices(float arr[], int size) {
   float sum = 0.0;
   int i;
   for (i=0; i<size; i++)
      sum += arr[i];
   return sum;
}
```

No difference from Unit19_SumArray.c !

2. Demo: Sum Array (4/6)

Unit19_SumArray_with_Files.c

```
// Read number of prices and prices into array arr.
// Return number of prices read.
int scanPrices(float arr[]) {
   FILE *infile;
   int size, i;

   infile = fopen("prices.in", "r"); // open file for reading
   fscanf(infile, "%d", &size);

   for (i=0; i<size; i++) fscanf(infile, "%f", &arr[i]);

   fclose(infile);
   return size;
}

// Print the total price
void printResult(float total_price) {
   FILE *outfile;
   outfile = fopen("prices.out", "w"); // open file for writing
   fprintf(outfile, "Total price = $%.2f\n", total_price);
   fclose(outfile);
}
```

2. Demo: Compare Input Functions (5/6)

Unit19_SumArray.c

```
int scanPrices(float arr[]) {
   int size, i;

   printf("Enter number of prices: ");
   scanf("%d", &size);

   printf("Enter prices:\n");
   for (i=0; i<size; i++)
      scanf("%f", &arr[i]);

   return size;
}
```

Note that when we use an input file, prompts for interactive input become unnecessary.

```
int scanPrices(float arr[]) {
   FILE *infile;
   int size, i;

   infile = fopen("prices.in", "r");
   fscanf(infile, "%d", &size);

   for (i=0; i<size; i++)
      fscanf(infile, "%f", &arr[i]);

   fclose(infile);
   return size;
}
```

Unit19_SumArray_with_Files.c

2. Demo: Compare Output Functions (6/6)

Unit19_SumArray.c

```
void printResult(float total_price) {
   printf("Total price = $%.2f\n", total_price);
}
```

```
void printResult(float total_price) {
   FILE *outfile;

   outfile = fopen("prices.out", "w");
   fprintf(outfile, "Total price = $%.2f\n", total_price);

   fclose(outfile);
}
```

Unit19_SumArray_with_Files.c

3. Opening File and File Modes (1/2)

- Prototype:

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

- Returns **`NULL`** if error; otherwise, returns a pointer of **`FILE`** type
- Possible errors: non-existent file (for input), or no permission to open the file
- File mode for text files (we will focus only on "r" and "w"):

Mode	Meaning
"r"	Open for reading (file must already exist)
"w"	Open for writing (file needs not exist; if exists, old data are overwritten)
"a"	Open for appending (file needs not exist)
"r+"	Open for reading and writing, starting at beginning
"w+"	Open for reading and writing (truncate if file exists)
"a+"	Open for reading and writing (append if file exists)

3. Opening File and File Modes (2/2)

- To ensure a file is opened properly, we may add a check. Example:

```
int scanPrices(float arr[]) {
   FILE *infile;
   int size, i;
   if ((infile = fopen("prices.in", "r")) == NULL) {
      printf("Cannot open file \"prices.in\"\n");
      exit(1);
   }
   . . .
}
```

- Function exit(*n*) terminates the program immediately, passing the value *n* to the operating system. Putting different values for *n* at different exit() statements allows us to trace where the program terminates. *n* is typically a positive integer (as 0 means good run)
- To use the exit() function, need to include <stdlib.h>.

4. Closing File

- Prototype:

```
int *fclose(FILE *fp)
```

- Allows a file that is no longer used to be closed
- Returns **`EOF`** if error is detected; otherwise, returns 0
- It is good practice to close a file after use

5. I/O Functions to Read and Write

- Formatted I/O: **`fprintf`**, **`fscanf`**
  - Uses format strings to control conversion between character and numeric data
- Character I/O: **`fputc`**, **`putc`**, **`putchar`**, **`fgetc`**, **`getc`**, **`getchar`**, **`ungetc`**
  - Reads and writes single characters
- Line I/O: **`fputs`**, **`puts`**, **`fgets`**, **`gets`**
  - Reads and writes lines
  - Used mostly for text streams
- Block I/O: **`fread`**, **`fwrite`**
  - Used mostly for binary streams ← we won't cover this

5.1 Formatted I/O (1/4)

- Uses format strings to control conversion between character and numeric data
  - **`fprintf`**: converts numeric data to character form and writes to an output stream
  - **`fscanf`**: reads and converts character data from an input stream to numeric form
- Both **`fprintf`** and **`fscanf`** functions can have variable numbers of arguments
- Example:

```
float weight, height;
FILE  *fp1, *fp2;
. . .
fscanf(fp1, "%f %f", &weight, &height);
fprintf(fp2, "Wt: %f, Ht: %f\n", weight, height);
```

5.1 Formatted I/O (2/4)

- **`fprintf`** returns a negative value if an error occurs; otherwise, returns the number of characters written
- **`fscanf`** returns **`EOF`** if an input failure occurs before any data items can be read; otherwise, returns the number of data items that were read and stored

```
printf(" … ");   =   fprintf(stdout, " … ");
```

```
scanf(" … ");   =   fscanf(stdin, " … ");
```

5.1 Formatted I/O (3/4)

Unit19_Formatted_IO.c

```
#include <stdio.h>
int main(void) {
   FILE *infile, *outfile;
   char x;
   int y;
   float z;

   infile = fopen("formatted.in", "r");
   outfile = fopen("formatted.out", "w");

   fscanf(infile, "%c %d %f", &x, &y, &z);
   fprintf(outfile, "Data read: %c %d %.2f\n", x, y, z);

   fclose(infile);
   fclose(outfile);
   return 0;
}
```

File “formatted.in”:

```
10 20 30
```

What’s the output in “formatted.out”?

```
Data read: 1 0 20.00
```

5.1 Formatted I/O (4/4)

Unit19_Formatted_IO_v2.c

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

int main(void) {
   . . .

   if ((infile = fopen("formatted.in", "r")) == NULL) {
      printf("Cannot open file \"formatted.in\"\n");
      exit(1);
   }
   if ((outfile = fopen("formatted.out", "w")) == NULL) {
      printf("Cannot open file \"formatted.out\"\n");
      exit(2);
   }

   . . .
}
```

To use exit()

Check if file can be opened.

Use different exit values for debugging purpose.

It is better to check that the files can be opened.

5.2 Detecting End of File & Errors (1/2)

- Each stream is associated with two indicators: error indicator & end-of-file (EOF) indicator
  - Both indicators are cleared when the stream is opened
  - Encountering end-of-file sets end-of-file indicator
  - Encountering read/write error sets error indicator
  - An indicator once set remains set until it is explicitly cleared by calling **`clearerr`** or some other library function
- **`feof()`** returns a non-zero value if the end-of-file indicator is set; otherwise returns 0
- **`ferror()`** returns a non-zero value if the error indicator is set; otherwise returns 0
- Need to include <stdio.h>

5.2 Detecting End of File & Errors (2/2)

- Caution on using **`feof()`**

Unit19_feof.c

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

int main(void) {
   . . .
   while (!feof(infile)) {
      fscanf(infile, "%d", &num);
      printf("Value read: %d\n", num);
   }
   . . .
}
```

Input file "feof.in"

```
10 20 30
```

Output:

```
Value read: 10
Value read: 20
Value read: 30
Value read: 30
```

Why does the last line appear twice?
To be discussed in discussion session.
(Hint: http://www.gidnetwork.com/b-58.html)

5.3 Character I/O: Output (1/4)

- Functions: `fputc()`, `putchar()`

```
int ch = 'A';
FILE *fp;

putchar(ch); // writes ch to stdout

fp = fopen( ... );
fputc(ch, fp); // writes ch to fp
```

- `fputc()` and `putchar()` return EOF if a write error occurs; otherwise, they return character written

5.3 Character I/O: Input (2/4)

- Functions: `fgetc()`, `getchar()`, `ungetc()`

```
int ch;
FILE *fp;

ch = getchar() // reads a char from stdin

fp = fopen( ... );
ch = fgetc(fp); // reads a char from fp
```

- `fgetc()` and `getchar()` return EOF if a read error occurs or end of file is reached; otherwise, they return character read
  - Need to call either `feof()` or `ferror()` to distinguish the 2 cases

5.3 Character I/O: ungetc (3/4)

- **`ungetc()`** pushes back a character read from a stream and returns the character it pushes back
- Example: Read a sequence of digits and stop at the first non-digit

```
int ch;
FILE *fp = fopen( ... );

while (isdigit(ch = getc(fp))) {
   // process digit read
   . . .
}
ungetc(ch, fp); // pushes back last char read
```

isdigit(ch) is a function to check whether ch contains a digit character; it returns 1 if so, or 0 otherwise.

5.3 Character I/O: Demo Copy File (4/4)

Unit19_CopyFile.c

```
int copyFile(char sourcefile[], char destfile[]) {
   FILE *sfp, *dfp;
   int ch;

   if ((sfp = fopen(sourcefile, "r")) == NULL)
      exit(1);  // error - can't open source file
   if ((dfp = fopen(destfile, "w")) == NULL) {
      fclose(sfp); // close source file
      exit(2);  // error - can't open destination file
   }
   while ((ch = fgetc(sfp)) != EOF) {
      if (fputc(ch, dfp) == EOF) {
         fclose(sfp); fclose(dfp);
         exit(3);  // error - can't write to file
      }
   }
   fclose(sfp); fclose(dfp);
   return 0;
}
```

5.4 Line I/O: Output (1/6)

- Functions: `fputs()`, `puts()`

```
FILE *fp;

// writes to stdout with newline character appended
puts("Hello world!");

fp = fopen( ... );
// writes to fp without newline character appended
fputs("Hello world!", fp);
```

- **`fputs()`** and **`puts()`** return **`EOF`** if a write error occurs; otherwise, they return a non-negative number

5.4 Line I/O: Input (2/6)

- Functions: `fgets()`, `gets()`

```
char s[100];
FILE *fp;

gets(s);  // reads a line from stdin

fp = fopen( ... );
fgets(s, 100, fp); // reads a line from fp
```

- **`fgets()`** and **`gets()`** store a null character at the end of the string
- **`fgets()`** and **`gets()`** return a null pointer if a read error occurs or end-of-file is encountered before storing any character; otherwise, return first argument
- Avoid using **`gets()`** due to security issue

5.4 Line I/O: fgets() (3/6)

- Prototype:

```
char *fgets(char *s, int n, FILE *fp)
```

  - *s* is a pointer to the beginning of a character array
  - *n* is a count
  - *fp* is an input stream

- Characters are read from the input stream *fp* into *s* until
  - a newline character is seen,
  - end-of-file is reached, or
  - *n* – 1 characters have been read without encountering newline character or end-of-file

- If the input was terminated because of a newline character, the newline character will be stored in the array before the terminating null character (‘\0’)

5.4 Line I/O: fgets() (4/6)

- If end-of-file is encountered before any characters have been read from the stream,
  - `fgets()` returns a null pointer
  - The contents of the array *s* are unchanged
- If a read error is encountered,
  - `fgets()` returns a null pointer
  - The contents of the array *s* are indeterminate
- Whenever `NULL` is returned, `feof` or `ferror` should be used to determine the status

5.4 Line I/O: Demo Counting Lines (5/6)

- Write a function that takes as input the name of a text file and returns the number of lines in the input file.
- If an error occurs, the function should return a negative number.
- Assume that the length of each line in the file is at most 80 characters.

5.4 Line I/O: Demo Counting Lines (6/6)

Unit19_CountLines.c

```
#define MAX_LINE_LENGTH 80
int countLines(char filename[]) {
   FILE *fp;
   int  count = 0;
   char s[MAX_LINE_LENGTH+1];

   if ((fp = fopen(filename, "r")) == NULL)
      return -1;  // error

   while (fgets(s, MAX_LINE_LENGTH+1, fp) != NULL)
      count++;

   if (!feof(fp))    // read error encountered
      count = -1;

   fclose(fp);
   return count;
}
```

6. Exercise #1: Reverse Array (1/3)

- You are given the program Unit19_ReverseArray.c to read values into an integer array, reverse the array, and print the array after reversal.
- Modify the program such that it reads from a text file "array.in" and writes to a text file "array.out"

6. Exercise #1: Reverse Array (2/3)

Unit19_ReverseArray.c

```c
#include <stdio.h>
#define MAX_SIZE 10

int scanArray(int []);
void printArray(int [], int);
void reverseArray(int [], int);

int main(void) {
   int array[MAX_SIZE], size;

   size = scanArray(array);

   reverseArray(array, size);

   printf("After reversing: ");
   printArray(array, size);
   return 0;
}
```

```c
// Reverse the array
void reverseArray(int arr[],
                  int size) {
   int i, temp;

   for (i=0; i<size/2; i++) {
      temp = arr[i];
      arr[i] = arr[size-i-1];
      arr[size-i-1] = temp;
   }
}
```

6. Exercise #1: Reverse Array (3/3)

Unit19_ReverseArray.c

```
// Read elements into array and
// return number of elements read.
int scanArray(int arr[]) {
   int size, i;

   printf("Enter size of array (<=%d): ",
          MAX_SIZE);
   scanf("%d", &size);

   for (i=0; i<size; i++) {
      scanf("%d", &arr[i]);
   }

   return size;
}
```

```
// Print array
void printArray(int arr[],
                int size) {
   int i;

   for (i=0; i<size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

6. Exercise #2: Trimming Blanks

- Write a program TrimBlanks.c that contains a function
  `int trimBlanks(char infile[], char outfile[])`
  that takes an input text file and produces a new text file that is a duplicate copy of the input file except that each sequence of consecutive blank characters is replaced by a single blank character.
- The function returns `-1` if there is an error; otherwise, it returns the number of blank characters trimmed.
- An incomplete program Unit19_TrimBlanks.c is given. A test input file trimblanks.in is also given.

Summary

- In this unit, you have learned about
  - How to open text files for reading or writing
  - How to read input from text files
  - How to write output to text files

End of File