**Lecture 20 - Chapter 11: C File Processing – Mon Oct 30 or Tues Oct 31**

**Announcements**

Reading:

* Chapter 11

Assignments:

* Due: Assignment #8 - due on **Nov 1** (MW class) or **Nov 2** (TR class) (no late assignments accepted)

**Today’s Goals**

1. Exams #2 Feedback
2. Files and Streams
3. Creating a Sequential Access File
4. Reading Data from a Sequential Access File
5. Secure C Programming

**Exam #2**

Key Points:

* On functions invocations writing code that looks like:

**double** result = distance (~~double~~ x1, ~~double~~ y1, ~~double~~ x2, ~~double~~ y2);

Should be

**double** result = distance (x1, y1, x2, y2);

* Understanding the difference between the address operators and dereferencing operator

**Exam #2 Question 4b:**

Draw a picture to help you understand what is going on.

**int** a[] = {100, 200, 300, 400};

**int** \*aPtr = a;

**int** i = 42, k = 10;

**int** \*iPtr = &i,

**int** \*kPtr = &k;

|  |  |
| --- | --- |
| a | 100 |
|  | 200 |
|  | 300 |
|  | 400 |

**\*aPtr**

……..

……..

|  |  |
| --- | --- |
| aPtr |  |
| i | 42 |
| K | 10 |
|  |  |

|  |  |
| --- | --- |
| iPtr |  |
| kPtr |  |

\*kPtr = (\*kPtr) \* (\*kPtr);

\*iPtr = \*kPtr;

**printf** ("\*iPtr = %d\n", \*iPtr); \_\_\_\_\_\_100\_\_\_\_\_\_\_\_\_

**printf** ("\*kPtr = %d\n", \*kPtr); \_\_\_\_\_\_100\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| aPtr |  |
| i | ~~42~~ 100 |
| K | ~~10~~ 100 |
|  |  |

|  |  |
| --- | --- |
| iPtr |  |
| kPtr |  |

**if** (\*iPtr == \*kPtr) {

**printf** ("same\n"); \_\_\_\_\_ same\_\_\_\_\_\_\_\_

}

**else** {

**printf** ("different\n"); \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

**if** (iPtr == kPtr) {

**printf** ("same\n"); \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

**else** {

**printf** ("different\n"); \_\_\_\_\_different\_\_\_\_\_\_

}

**if** (\*kPtr == \*aPtr) { \_\_\_\_\_\_same\_\_\_\_\_\_\_\_

**printf** ("same\n");

}

**else** {

**printf** ("different\n"); \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

**Exam #2 Question 5:**

// Do two pointers point to same value.

// This function expects a pointer to an int (i.e. memory address)

bool **sameValue** (**int** \*ptr1, **int** \*ptr2) {

**if** (ptr1 == ptr2) { // I also accepted if (\*ptr1 == \*ptr2)

**return** true;

}

**else** {

**return** false;

}

//return (ptr1 == ptr2); // You could reduce all this to one line of code

}

**int** value = 10;

**int** \*ptr1 = &value;

**int** \*ptr2 = &value;

bool same = sameValue(ptr1, ptr2);

**if** (same) {

**printf** ("Pointers point to same value\n");

}

**else** {

**printf** ("Pointers point to different value\n");

}

**Note: These invocations are incorrect:**

bool same = sameValue(\*ptr1, \*ptr2); <- integer value not a memory address

bool same = sameValue(&ptr1, &ptr2); <- memory address of each pointer not the

memory address stored in the pointers!

**Exam #2 Question 6:**

Assume you have a 10-element integer array named ***sales*** with the values shown below. Write code for the following array/pointer related tasks.

--------------------------------------------------------------------------

| 15 | 12 | 27 | 55 | 20 | 24 | 10 | 68 | 19 | 90 |

--------------------------------------------------------------------------

Declare a pointer variable named ***salesPtr***. Assign starting address of array ***sales*** to variable ***salesPtr*.** Print the value of **last element** in ***sales*** using **array index notation,** then using the **pointer variable**. **(3 pts)**

**int** \*salesPtr = sales;

**printf** ("Sales[9] = %d\n", sales[9]);

**printf** ("salesPtr[9] = %d\n", salesPtr[9]);

**printf** ("\*(salesPtr + 9) = %d\n", \*(salesPtr + 9)); // Also works

Declare a pointer variable a***notherSalesPtr*** and assign it the address of the 3rd element in array. **(2 pts)**

**int** \*anotherSalesPtr = &sales[2]; // People forgot to include the &

**// Without the address operator you are**

**// referencing the value in sales[2] not**

**// the address of that location**

**int** \*anotherPtr = sales + 2; // Also works

Draw a picture of the memory for ***sales***, ***salesPtr, and anotherSalesPtr*** variables. Show the values stored in these 3 variables. Assume the sizeof an integer is 4 bytes and the starting address of the ***sales*** array is 0028FE60. **(3 pts)**

**Memory Address Value**

sales 0028FE60 15

0028FE64 12

0028FE68 27

… …

salesPtr some address 0028FE60

anotherSalesPtr some address 0028FE68

For each expression, what value would be displayed if these were placed in a printf statement? **(2 pts)**

\*(salesPtr + 2) \_\_\_\_\_\_27\_\_\_\_\_\_\_\_

\*salesPtr + 2 \_\_\_\_\_\_17\_\_\_\_\_\_\_\_

**Today’s Terminology**

**Terminology**

* Stream
  + Communication channel between programs and files.
  + Data stream is a stream of data that goes into or come out of a process.
* End of File Marker
  + Represented by **EOF** – symbol that stands for “end of file”
  + Normally EOF has the value -1 - it is system dependent so test for EOF instead of -1
  + Symbolic integer constant defined in <stdio.h>
* Standard Input
  + Stream that receives input from the keyboard
* Standard Output
  + Stream that sends output to the screen
* Standard Error
  + Stream that sends error messages to the screen

**Files and Streams**

**Purpose**

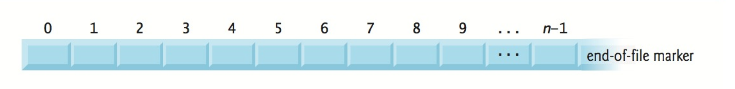
* Data stored in variables is temporary – goes away when program ends
* Files give a way to preserve values after program exists

**Streams**

* Communication channel between programs and files.
* The operating system controls how data gets into and out of the standard streams
* When your program begins execution, 3 data streams are opened automatically:
* Standard input
  + Received input from keyboard
* Standard output
  + Displays output on screen
* Standard error
  + Displays errors messages on screen

**Files in C**

* C views a file as a sequential stream of bytes
* Ends with an end-of-file marker



* Opening a file returns a pointer to
  + **FILE** structure (defined in <stdio.h>)
    - FILE \*filePtr;
  + This structure contains information to process file
* **stdin**, **stdout, stderr**
  + File pointers for standard input, standard output, standard error

**File Functions**

* <stdio.h> contains functions to process files
* **Reading File Data**
* **fgetc**
  + Reads **one character** from a file
  + Takes a FILE pointer of the file to be read
  + **fgetc (stdin)** – reads one character from stdin
  + **fgetc (stdin)** is equivalent to **getchar()** (chapter 8)
* **fgets**
  + Reads **one line** from a file
  + Assignment #6 we used the following to read one line and place it into the line array:

**fgets**(line, MAX\_LINE, filePtr);

* **fscanf**
  + Reads data from a data stream
  + Allows you to choose where you want to read text from
  + fscanf is a general version of scanf
  + Behind the scenes:

**scanf (“%d”, &value);**

When you call scanf() These calls are equivalent

A call is made to fscanf()

**fscanf (stdin, “%d”, &value)**;

Reads data from stdin is the Standard Data read from stream

A data stream input data stream

* **Writing File Data**
* **fputc**
  + Writes **one character** to a file
  + Takes a character to be written to a file and a FILE pointer of the file to be written to
  + **fputc (‘a’, stdout)** – writes the character ‘a’ to stdout
  + **fputc (‘a’, stdout)** is equivalent to **putchar(‘a)**
* **fputs**
  + Writes **one line** to a file
* **fprintf**
  + Writes data to a data stream
  + Allows you to choose where you want to send text to
  + fprintf is a general version of printf
  + Behind the scenes:

**printf (“I’m so glad exams are over\n”);**

When you call printf() These calls are equivalent

A call is made to fprintf()

**fprintf (stdout, “I’m so glad exams are over\n”)**;

Sends data to stdout is the Standard Data sent to stream

A data stream output data stream

**fprintf (stderr, “Invalid value %d was entered\n”, 20);**

stderr is the Standard error data stream

By default, set to display on the screen

**Creating a Sequential-Access File**

**Creating Your Data Stream**

* Operating system gives you 3 file data structures: stdin, stdout, stderr
* You can create your own data stream

**Pointer to a File**

* Each data stream (i.e. file) is represented by a **pointer to a file**
* Each file needs its own file pointer

FILE \*filePtr; // Create a pointer to a file

**Functions**

* **fopen** 
  + Function used to create a new data stream

FILE \*filePtr = **fopen** ("C:/writeFile.txt", "r");

* + Establishes the connection between the file and the file pointer
    - not established – returns NULL
    - established - returns pointer to file structure
  + Two parameters:
    - **name** – name of file to open, can include the path
    - **mode** – mode to open file in (see figure 11.5 for file modes)
      * “w” write mode
      * “r” read mode
      * “a” append mode

**if** ((filePtr = **fopen**("C:/concert.txt", "r")) != NULL) {

// If we get here the file was opened successfully!

}

* + What happens when
    - NULL is returned
      * File could not be opened maybe because it doesn’t exist or don’t have privileges
    - Open a file for writing but doesn’t exist?
      * **fopen** - creates the file as long as it has permission to do so in directory
    - Open an existing file for writing?
      * ***File contents are discarded with no warning!***
      * Starts writing at the beginning
* **feof**
  + Checks for end of file
  + EOF indicator tells the program there is no more data in the file to be processed

**int** number;

**while** (!feof(filePtr)) {

**fscanf**(filePtr, "%d", &number);

**printf** ("Value read from file = %d\n", number);

}

* **fscanf**
  + Reads from the file stream
    - scanf –used to read data from the keyboard
    - fscanf - used to read data from a file
* **fprintf**
  + Print to the file stream
    - printf –used to write data to the screen
    - fprintf - used to write data to a file

**if** ((filePtr = **fopen**("C:/writeFile.txt", "w")) != NULL) {

**fprintf** (filePtr, "%d\n", 222);

**fprintf** (filePtr, "%d\n", 333);

**fprintf** (filePtr, "%d\n", 444);

**fprintf** (filePtr, "%d\n", 555);

**fclose**(filePtr);

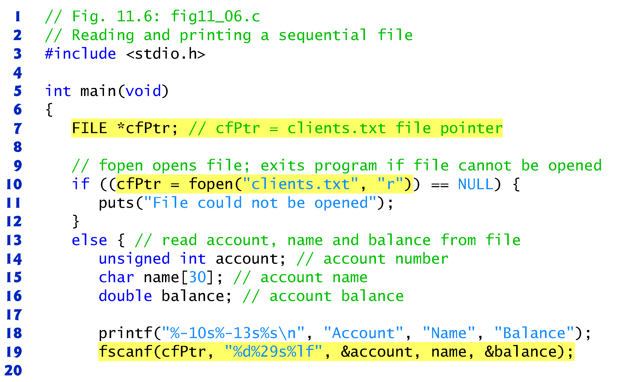
}

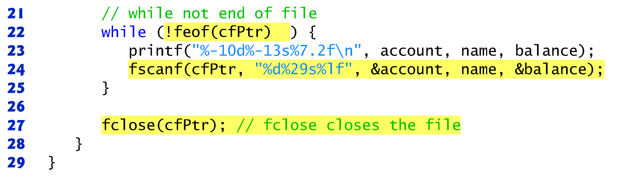
* **fclose**
  + Close the file when done using file
  + All streams are closed when program ends but good practice to close stream when done

**Reading Data from a Sequential-Access File**

**Reading Data from a File**

* Use fscanf
* Open file and fscanf read sequentially from the beginning
* There is a file position pointer – an offset value – that keeps track of where the next read will occur
* Figure 11.6 – shows how you can read several different values in one fscanf statement



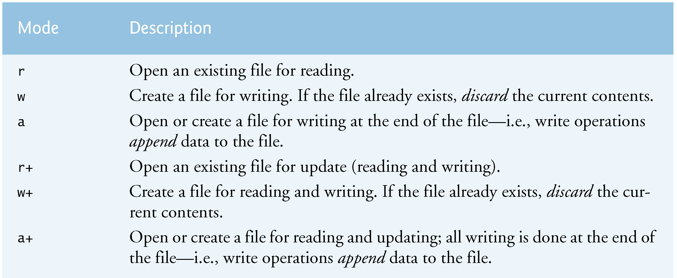


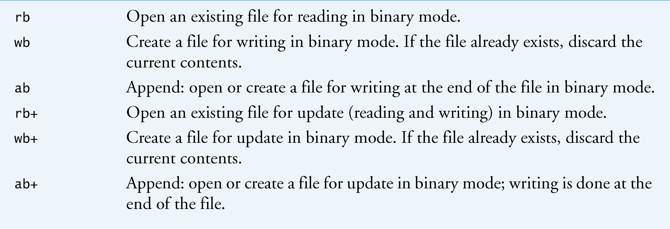
**Resetting the File Position Pointer**

* rewind (filePtr);
  + Resets file position pointer to beginning of file

**Notes:**

* Sequential access to a file is done with fscanf and fprintf
* If you need to update a file, using fscanf and fprintf doesn’t generally work because the size of the data records in the file vary
* The solution to this issue is to rewrite the entire file.
* Modes when dealing with files – Figure 11.5





**Secure C Programming**

**Secure Programming**

* To write code that uses techniques that can stand up to attacks
* This topic is an entire class so we won’t be focusing on this topic
* We will discuss some of the techniques

**CERT C Secure Coding Standard**

* CERT – Computer Emergency Response Team - [www.cert.org](http://www.cert.org)
* Publishes and promotes secure coding standards
* Standard for C
  + <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
* Standard for other lanaguages:
  + <https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

**Secure Version of fscanf and fprintf**

* fscanf\_s
* fprintf\_s

Remember not all C compilers include these functions.

**Always Check Return Value of File Functions**

* Check return value to ensure call was success and function performed task