**I/O Part 2 Standard and Binary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Since you should already know the Standard I/O library functions, this will be a quick review. It is followed by the binary I/O functions. In the previous section we discussed that stream I/O helps by doing data conversions, buffering data, and efficiently reading/writing data.  The standard library uses file pointers. When we discuss low level I/O, we will provide more details about file descriptors and file pointers.  **scanf**, **gets**, and **getc** read from **stdin**, which is a file pointer set up by C. The corresponding functions **fscanf**, **fgets**, and **fgetc** are passed a file pointer which is returned by the **fopen** function. Each of these functions read buffered text data. (In I/O Part 3, we will give an example of buffered input.)  **printf** and **putc** write to **stdout**, which is also a file pointer set up by C. The corresponding functions **fprintf** and **fputc** are passed a file pointer.  **fread** and **fwrite** read/write a specified number of bytes. The data can contain binary information. These functions are also passed a **FILE pointer**. | |  |  |  | | --- | --- | --- | | **Function** | **Category** | **Purpose** | | **scanf**  **fscanf** | std i/o (stream) | stream input using format codes | | **gets**  **fgets** | std i/o (stream) | stream input of text lines | | **printf**  **fprintf** | std i/o (stream) | stream output using format codes | | **getc**  **fgetc** | std i/o (stream) | get next char from a stream | | **putc**  **fputc** | std i/o (stream) | put a char to a stream | | **fopen** | std i/o | open a file for buffered i/o | | **fclose** | std i/o | close a file opened by fopen | | **fread** | binary read | binary input of one or more logical records | | **fwrite** | binary write | binary output of one or more logical records | | **fseek** | binary position | changes file position to a location relative to a number of bytes from the beginning of the file | |
| **Standard Stream I/O**  **Stream Output:**  **printf**(*formatSpecifier, value1, ...*);  Prints output to stdout based on the *formatSpecifier*. The number of values specified is dependent on the number of % format codes in the *formatSpecifier.*  **fprintf**(*file*, *formatSpecifier, value1, ...*)**;**  Similar to printf except it prints the output to the specified file (which is usually opened with an **fopen**).  **Stream Input:**  **char \* fgets**(*stringVariable, maxLength, file*);  This reads from the specified file until either *maxLength* - 1 characters are read or until a line feed character is encountered. The read text is placed in *stringVariable*. fgets returns the NULL for EOF or the address of *stringVariable*.  **int scanf**(*formatSpecifier, address1, address2, …);*  This reads from stdin based on the *formatSpecifier.* **fscanf** reads from a specified file. After the *formatSpecifier*, you must specify addresses since scanf stores its input into those addresses. scanf returns the number of successful conversions from its input that are stored into those addresses  **Input from Memory**  **int sscanf**(*stringVariable, formatSpecifier, address1, address2, …);*  We will usually read a line of text into a variable using **fgets** and then use **sscanf** to take the data from that variable to populate our target variables.  For more detail about these functions refer to <http://www.cs.utsa.edu/~clark/cs2123/quickCReviewAns.docx>. | **Example 2-1:** printf, fgets, and sscanf  // The stdin contains an ABC123 ID and two grades for many  // students.  // Read text lines (using fgets) until EOF.  #include <stdio.h>  void errExit(const char szFmt[], ... ); // prototype  int main()  {  char szInputBuffer[101]; // one text line of input  char szABC123[7]; // student ABC123 ID  double dGrade1; // student's first grade  double dGrade2; // student's second grade  int iScanfCount;  // Print a column heading  printf("%-6s %-6s\n", "ABC123", "Grade");  // Read a line of text until EOF.  // Note that fgets returns NULL when it reaches EOF.  while (fgets(szInputBuffer, 100, stdin) != NULL)  {  // copy the data to the target variables  iScanfCount = sscanf(szInputBuffer, "%s %lf %lf"  , szABC123  , &dGrade1  , &dGrade2);  if (iScanfCount < 3)  errExit("Only received %d valid values. Found : %s\n"  iScanfCount, szInputBuffer);  // Print the ABC123 ID and the higher grade  if (dGrade1 > dGrade2)  printf("%-6s %6.2lf\n", szABC123, dGrade1);  else  printf("%-6s %6.2lf\n", szABC123, dGrade2);  }  return 0;  }  why do we use fgets and then sscanf instead of scanf?  If there’s an error, scanf stops at that error and can not advance. sscanf has the ability to continue past the error and ignore it.  Note: you can find errExit.c in  /usr/local/courses/clark/cs3423/IO |
| **Buffered Output**  Buffered output to the terminal acts differently from output to a file. When writing to the terminal, the output is sent almost immediately instead of buffering it.  When directing output to a file, standard stream output will write to an internal buffer for efficiency. It won't actually write it to the file until there is enough data for a data block, it is asked to flush or the file is closed.  Note that the **tail** command in Linux prints the last portion of a file. Its most useful option is the **-f** switch which *follows* a stream output file. This is useful to monitor the output of a file while it is being produced (.e.g, watching a log file). tail will keep looking at the contents until you CTRL-C it.  If you want it to flush the buffer after the first printf, type this function call:  fflush(stdout); | **Example 2-2:** output to the terminal vs buffering to a file  # Let's see some buffering  $ vi well.c  #include <stdio.h>  int main()  {  printf("Step 1\n");  sleep(10); // wait 10 seconds  printf("Step 2\n");  return 0;  }  $ gcc -o well well.c  $ ./well  Step 1 (it then waits for at least 10 seconds)  Step 2  # now let's write it to a file and check its progress using tail -f  $ well > out.txt &  $ tail -f out.txt  (it waits for at least 10 seconds before you see both results  Step 1  Step 2  CTRL-C  What was the difference?  ?? |
| **binary i/o**  One of the benefits of C is that it provides the ability to read and write structures. Since the data in a structure isn't just text (it could be doubles and/or integers), the files contain **binary** (i.e., non-printable) data. You **cannot** easily edit binary data with a text editor.  Instead of using scanf and printf, we use **fread**() and **fwrite**(), which do not support format codes. | // We will use these typedefs and declaration in our examples  typedef struct  {  int iExemptionCnt; // Number of exemptions  char cFillingStatus; // M - Married, S - Single  // X - married but filling  // as single  double dWithholdExtra; // extra amount to withhold  } W4;  typedef struct  {  char szEmployeeId [10];  char szFullName[40];  double dHourlyRate;  W4 w4;  } Employee;  Employee employee; |
| **fopen**  FILE \***fopen** (const char \*pszFilename, const char \*szMode)   * opens the specified file and returns a FILE pointer or NULL if it couldn't be opened * The filename is the name of the file, possibly qualified with a path (e.g., "~/cs3423/employeeData.dat") * *mode* is a string which tells fopen how to open the specified file:   "r" open file as text stream input. If the file doesn't exist, fopen will return NULL.  "w" open file for text stream output. It will create the file if it doesn't already exist and overwrite the file it does already exist.  "a" open an existing file to append text to it.  "w+" open a file for read and write.  "rb" open file as binary input. If the file doesn't exist, fopen will return NULL.  "wb" open a file for binary output.  "ab" open an existing file to append binary data to it.  "wb+" open a file for binary read and write | **Example 2-3:** use fopen to open a file for binary input and another for binary output  FILE \*pfileEmployee;  #define EMPLOYEE\_FILE "employeeData.dat"  // Open binary file for read  pfileEmployee = fopen(EMPLOYEE\_FILE, "rb");  if (pfileEmployee == NULL)  errExit("could not find employee file: '%s'\n"  , EMPLOYEE\_FILE);  FILE \*pfileNewEmployee;  #define NEW\_EMPLOYEE\_FILE "employeeDataNew.dat"  // Open binary file for write  pfileNewEmployee = fopen(NEW\_EMPLOYEE\_FILE, "wb");  if (pfileNewEmployee == NULL)  errExit("could not open new employee file: '%s'\n"  , NEW\_EMPLOYEE\_FILE); |
| **fread**  long **fread** (void \*psbBuf, long lSizeOneRec  , long lNumberOfRec, FILE \*pFile)   * Beginning with the current file position, fread reads 1 or more binary records into the specified buffer which is typically a binary structure. * Returns the number of records successfully read as its functional value. If one record is read successfully, 1 is returned. Returns 0 if none are read. * lSizeOneRec is the size of one record or structure * lNumberOfRec is the number of records to be read. If only one, specify 1L. * psbBuf is a void pointer so that it can be any structure or a character buffer   A **void \* pointer** gets its data type from the invoking parameter, but it must be an address. | **Example 2-4:** use fread to read one employee record. The files would have been opened as done in example 2-3.  iNumRecRead = fread (&employee, // address to store record data  , sizeof(Employee) // size of one record  , 1L // number of records as a long  , pfileEmployee);  if (iNumRecRead != 1)  errExit("Expected to get an Employee record, %s"  , "but fread returned no records"); |
| **fwrite**  long **fwrite** (void \*psbBuf, long lSizeOneRec  , long lNumberOfRec, FILE \*pFile)   * Beginning with the current file position, fwrite writes 1 or more binary records to the file. * Returns the number of records successfully written as its functional value. * lSizeOneRec is the size of one record or structure * lNumberOfRec is the number of records to be write. If only one, specify 1L. * psbBuf is a void pointer so that it can be any structure or a character buffer * pFIle is a FILE pointer returned by fopen. | **Example 2-4:** read and write each binary employee record. The files would have been opened as done in example 2-3.  int iRecCount = 0;  // copy the employee file to create a new one  while (fread (&employee, // address to store record data  , sizeof(Employee) // size of one record  , 1L // number of records as a long  , pfileEmployee) == 1)  {  // process that employee record  iRecCount ++;  rc = fwrite(&employee, // address for the record data  , sizeof(Employee) // size of one record  , 1L // number of records is 1 as a long  , pfileNewEmployee);  if (rc != 1)  errExit("Error Writing record %d to the new employee file"  , iRecCount);  }  fclose (pfileEmployee);  fclose(pfileNewEmployee); |
| Warning!!! Warning !!! Warning !!!  Although fread is fairly safe, you can overwrite memory if you specify the wrong *sizeOfRecord* or *lNumber of Records.* |  |
| **fclose**  void **fclose** (FILE \*pFile)   * Closes the specified file, completing the I/O and freeing internal memory used for the file. | See the example above |
| **Sequential Files**  A sequential file contains multiple records which are arranged sequentially (i.e., one record after the other).   * No direct access * Very difficult to insert a new record anywhere other than at the end of the file. * Updates of a record are fairly easy if the new record is of the same size. * Many conventional application systems which use sequential files have a Master File and a Differential File (describes changes to the master). * A Master File contains the latest Master copy of the data sorted by a key. | For a complete example of merging sequential binary files, examine (on your own time) empMerge.c  The readme.txt file under the **/usr/local/courses/clark/cs3423/IO/** directory can be executed by bash. It will show how to create binary data files from stream input and how to merge them:  $ bash readme.txt |
| **Direct Access Files**  A direct access file is defined to allow access at particular positions directly (without having to read the file sequentially).   * **fseek** can be used to set the current position based on a relative byte offset. **fread** or **fwrite** can then be used to read or write the data at that position. * **Sequential access** is also provided by simply using fread or fwrite from the beginning of the file. | Assuming a file contains Employee records, we can use **fseek** to access the *ith* employee by determining a relativeByteAddress =  (i - 1) \* sizeof(Employee)  b-tree (balanced tree not binary tree) and hashed files are frequently stored/manipulated as direct access files. |
| **Direct Positioning in a File**  int **fseek** (FILE \*pFile, long lRelativeByteAddress  , int iSeekMode)   * Sets the position in the file based on an offset and iSeekMode. * Returns 0 if successful. * Values of iSeekMode:   SEEK\_SET set the position relative to the beginning of the file  SEEK\_CUR seek from the current position  SEEK\_END seek from the end of the file  When using SEEK\_CUR and SEEK\_END, negative values for lRelativeByteAddress are allowed.   * The lRelativeByteAddress is a byte offset (relative to zero). * pFIle is a FILE pointer returned by fopen.   If you **fseek** past the end of the file, it is not an error. If you subsequently:   * **fread** - it will not find a record * **fwrite** - it will write a record at that location. It also pads with records containing all zeroes up to that new record. | **Example 2-5:** output from direct.c using inputDirect.txt. This code either writes or reads a record at a particular RBA. This output shows the contents for various Writes and Reads. Notice in particular that we didn't write to these RBAs: 0, 6, and 7; however, we can read from those RBAs.  Output from cs3423/IO/direct.c when it is executed:  > W 1 11111 10.00 1 S 0.00 Highwater, Helen  > R 1  11111 10.00 1 S 0.00 Highwater, Helen  > W 2 22222 12.00 1 S 20.00 Flood, T. Rential  > W 3 33333 25.00 2 M 100.00 Tall, Jerry  > W 4 44444 40.00 1 S 200.00 Yuss, Jean E.  > W 5 55555 8.00 1 X 0.00 Absent, Marcus  > R 4  44444 40.00 1 S 200.00 Yuss, Jean E.  > W 8 88888 88.00 1 S 0.00 Moss, Pete  > R 8  88888 88.00 1 S 0.00 Moss, Pete  > R 7  0.00 0 0.00  > R 9  Record number 9 not found for RBA 648  > R 50  Record number 50 not found for RBA 3600  > R 0  0.00 0 0.00  Notice that record number 0 and 7 weren't written, but they are all zero. When reading record numbers 9 and 50, we get a not found error since those are past the last written record (record number 8). |
| We will quickly examine the code in **/usr/local/courses/clark/cs3423/IO/direct.c**  We opened the direct access file using:  pFileDirect = fopen(pszDirectFileName, "wb+");  what does that "wb+" mean?   * ?? * ?? | **Example 2-6:** Examine a subset of direct.c. This code either writes or reads records at particular RBAs. Earlier in the input loop, we received **cCommand** and **lRecNum.**  We process that commandusing the following code**:**  switch(cCommand)  {  case 'W': // write  iScanfCnt = sscanf(szRemaining, "%6s %lf %d %c %lf %40[^\n]\n"  , employee.szEmployeeId  , &employee.dHourlyRate  , &employee.w4.iExemptionCnt  , &employee.w4.cFillingStatus  , &employee.w4.dWithholdExtra  , employee.szFullName);  // Check for bad input.  if (iScanfCnt < 6)  errExit(ERR\_INVALID\_EMPLOYEE\_DATA, szInputBuffer);  // seek to the desired record  lRBA = lRecNum\*sizeof(Employee);  rcFseek = fseek(pFileDirect, lRBA, SEEK\_SET);  assert(rcFseek == 0);  // write it to the direct file  iWriteNew = fwrite(&employee  , sizeof(Employee)  , 1L  , pFileDirect);  assert(iWriteNew == 1);  break;  case 'R': // read  lRBA = lRecNum\*sizeof(Employee);  rcFseek = fseek(pFileDirect, lRBA, SEEK\_SET);  assert(rcFseek == 0);  // print the information at the RBA  rc = fread(&employee, sizeof(Employee), 1L, pFileDirect);  if (rc == 1)  printf("%-7s %8.2lf %5d %c %8.2lf %-40s\n"  , employee.szEmployeeId  , employee.dHourlyRate  , employee.w4.iExemptionCnt  , employee.w4.cFillingStatus  , employee.w4.dWithholdExtra  , employee.szFullName);  else  printf("Record number %ld not found for RBA %ld\n"  , lRecNum, lRBA);  break; |
| **Compiling and Executing direct.c** | $ gcc –o direct direct.c  $ ./direct –i inputDirect.txt –o direct.dat  (the generated output was shown earlier) |

**©2017 Larry W. Clark,** UTSA CS students may make copies for their personal use