**File Organization Part 2 – Sequential and Direct Files**

|  |  |
| --- | --- |
| **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 this structure 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 szSSN[10];  char szFullName[40];  double dHourlyRate;  W4 w4;  } 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 if it does already exist.  "a" open an existing file to append text to it.  "w+" open a file for read and write.  "rb" open an existing file as binary input. If the file doesn't exist, fopen will return NULL.  "wb" open a file for binary output. It will create the file if it doesn't already exist and overwrite the file if it does already exist.  "ab" open an existing file to append binary data to it.  "rb+" open an existing file for binary input and output  "wb+" open a file for binary read and write. It will create the file if it doesn't already exist and overwrite the file if it does already exist. | **Example 2-1:** use fopen to open a file for binary input and another for binary output  FILE \*pfileEmployee;  #define EMPLOYEE\_FILE "employeeData.dat"  // Open an existing binary file for read only  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 only  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 2-2:** use fread to read one employee record. The files would have been opened as done in example 2-1.  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 2-3:** read and write each binary employee record. The files would have been opened as done in example 2-1.  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. | Example 2-4: Sample Master Employee File (ordered by Employee Id)  11111 Sal Ed Barr …  22222 Polly Cracker …  33333 Ted D Barr …  44444 Anita Dayhoff … |
| **Merging Simple Files**  Both files must be sorted by a key.  General approach when both files have same structure and file2 is the updated data:  1. Open file1 and file2 for binary read.  Open newMaster for binary write.  2. Read file1 into rec1.  3. Read file2 into rec2.  4. while (file1 not at eof and file2 not at eof)  If rec1.key < rec2.key  write new master from rec1  read file1 into rec1  else if rec1.key > rec2.key  write new master from rec2  read file2 into rec2  else  write new master from rec2  read file1 into rec1  read file2 into rec2  end while  5. // finish off file1 if it isn't at EOF  while (file1 not at eof)  write new master from rec1  read file1 into rec1  end while  6. // finish off file2 if it isn't at EOF  while (file2 not at eof)  write new master from rec2  read file2 into rec2  end while  7. close all files | See **empMerge.c** for a full example in C. |
| **Differential File**  A Differential File contains the changes that have occurred since the last Master was written. Between intervals of Differential/Master merging, all operations are written to the Differential File. This is done since sequential files do not provide direct access capabilities. |  |
| **Differential File Record Structure**  The record structure includes the master's record structure plus:  - timestamp of the operation  - operations type (Insert, Delete, Update)  Depending on the sort utility, it may also be necessary to put the data key and the front of the differential file record.  The timestamps are often stored in Greenwich Mean Time instead of Central Standard Time. What are some reasons?  Timestamps may be in this format:  YYYY-MM-DD.hh.mm.ss.nnnnnn (26 characters not including null byte)  When processing data, the Differential File is written in timestamp order. Prior to file merge, the Differential File must be sorted in data key then timestamp order. | **Example 2-5**: typedefs used for Employee Differential  #include <sys/time.h>  // We will use this structure 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 szSSN[10];  char szFullName[40];  double dHourlyRate;  W4 w4;  } Employee;  Employee employee;  typedef struct  {  char szKey[10];  char szRecordTs[27];  char cOperation; // I-Input,D-Delete,U-Update  Employee employee;  } EmployeeDifferentialRecord; |
|  | |
|  | Sample Differential Data (in original order)  111111111 2014-03-21.08.30.22.139399 I111111111 IMA NOVICE … M  454545451 2014-03-21.08.30.22.279222 U454545451 COL. KORN … M  111111111 2014-03-21.08.30.23.736838 U111111111 IMA NOVICE … X  333333333 2014-03-21.08.30.23.989292 U333333333 PERRY SCOPE … M  222222222 2014-03-21.08.30.24.595030 U222222222 MICK ALOBE … S  111111111 2014-03-21.08.30.25.292919 U111111111 IMA NOVICE … X  212121212 2014-03-21.08.30.27.392947 D212121212 C U LATER … S  393939393 2014-03-21-08.30.29.557493 I393939393 HAMMOND ECKS … S  333333333 2014-03-21.08.30.35.767733 D333333333 PERRY SCOPE … M |
| Employee Master File Data (in order by szKey)  155155155 TOM E GUNN … M  191919191 HOLLY WOOD … S  212121212 C U LATER … S  222222222 MICK ALOBE … S  333333333 PERRY SCOPE … M  350033000 BUD WISER … S  454545451 COL. KORN … M | Sample Differential Data (in key and timestamp order)  111111111 2014-03-21.08.30.22.139399 I111111111 IMA NOVICE … M  111111111 2014-03-21.08.30.23.736838 U111111111 IMA NOVICE … X  111111111 2014-03-21.08.30.25.292919 U111111111 IMA NOVICE … X  212121212 2014-03-21.08.30.27.392947 D212121212 C U LATER … S  222222222 2014-03-21.08.30.24.595030 U222222222 MICK ALOBE … S  333333333 2014-03-21.08.30.23.989292 U333333333 PERRY SCOPE … M  333333333 2014-03-21.08.30.35.767733 D333333333 PERRY SCOPE … M  393939393 2014-03-21-08.30.29.557493 I393939393 HAMMOND ECKS … S  454545451 2014-03-21.08.30.22.279222 U454545451 COL. KORN … M |
|  | New Master:  111111111 IMA NOVICE … X (3rd one)  155155155 TOM E GUNN … M  191919191 HOLLY WOOD … S  222222222 MICK ALOBE … S (from Diff file)  350033000 BUD WISER … S  393939393 HAMMOND ECKS … S  454545451 COL. KORN … M |
|  | Algorithm (in pseudo code) to process sorted Differential against Master to create a new master:  1. Open Master and Diff for binary read;  Open NewMaster for binary write;  2. Read Master into masterRec;  3. Read Diff into diffRec;  4. while (Master not at eof and Diff not at eof)  if (masterRec.szSSN < diffRec.szkey)  write NewMaster from masterRec;  Read Master into masterRec;  else  {  if (masterRec.szSSN == diffRec.szKey)  // use the Diff. Don’t need the Master  Read Master into masterRec;  saveKey = diffRec.szkey;  while (saveKey == diffRec.szkey && Diff not at eof)  lastOperator = diffRec.operator;  saveEmployee = diffRec.Employee;  Read Diff into diffRec;  end while  if (lastOperator != 'D')  write NewMaster from saveEmployee;  }  end while  5. // finish off Master  while (Master not at eof)  write NewMaster from masterRec;  Read Master into masterRec;  end while  6. // finish off Diff file  while (Diff not at eof)  saveKey = diffRec.szkey;  while (saveKey == diffRec.szkey && Diff not at eof)  lastOperator = diffRec.operator;  saveEmployee = diffRec.Employee;  Read Diff into diffRec;  end while  if (lastOperator != 'D')  write NewMaster from saveEmployee;  end while  7. close all files |
| **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 2-6:** output from direct.c using inputDirect.txt. This code either writes or reads a record at a particular Relative Byte Address (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/cs3743/IO/direct.c**  We opened the direct access file using:  pFileDirect = fopen(pszDirectFileName, "wb+");  what does that "wb+" mean?   * if it exists, we overwrite it and open it for read/write. * if it doesn't exist, we create it and open it for read/write | **Example 2-7:** 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) |
| **Hashing**  We assume that you saw hash tables in data structures and have some familiarity with hashing algorithms. The goal of hashing algorithms is to take a business key and turn it into a number representing an RBA such that we have few collisions.  Hashing is used to map a set of possibly M keys to a smaller set of N numeric values. We can then use it as an integer to access an array quickly.  Assuming our hashing works well, the average number of reads to access a data row will be **one**. |  |
| **Keys that are Characters**  Very frequently in business, natural keys are character strings (e.g., email address, ABC123, City Name). Some keys that have digits are too long to be reasonable direct keys (SSN, Telephone Number). In some cases the values have a huge range of possibilities and we need to represent a very sparse subset (e.g., many 100 million SSNs mapped to just 100s of entries). How can we efficiently access data based on these natural keys?  Hashing is used to map a set of possibly M keys to a smaller set of N numeric values. We can then use it as an integer to access an an array quickly. | How can we map 999,999,999 SSNs to a smaller range of N values? |
| **Hash Function**  **Division Method -** map the key into a table of size N using the remainder of dividing the key by N.  iHash = iKey % N; | Example #7 (assuming N is 8):   |  |  | | --- | --- | | iKey | iHash (iKey % 8) | | 36 | 4 | | 18 | 2 | | 72 | 0 | | 6 | 6 | | 43 | 3 | | 85 | 5 |   Hash Table:   |  |  | | --- | --- | | 0 | 72 and other data | | 1 |  | | 2 | 18 and other data | | 3 | 43 and other data | | 4 | 36 and other data | | 5 | 85 and other data | | 6 | 6 and other data | | 7 |  | |
| **Hash Function**  **Midsquare Method -** return the middle portion of the square of the key | Example #2:  Hash Table has an iHash that is 4 digits. (subscripts from 0 to 9999 so N is 10,000).  Key Value is 6 digits.  Example key value: 113586  Squaring key value: 12901779396  Select positions 4 to 7 and return the Hash Result Value 1779. |
| **Hash Function**  **Folding Method -** return the sum of parts of the key. Partition the key into parts. The parts are added and the sum is used as the hash result. | Example #3:  Hash Table has an iHash that is 3 digits.  Key Value is 9 digits (e.g., SSN).  Example Key Value: 206 318 291  Hash Result Value is 206 + 318 + 291 = 815  If necessary, we can use the Division Method to take the remainder. |
| **What if the key isn't numeric?**  Unfortunately, many keys are character strings (e.g., ABC123, email).  C can treat any character value as a number since characters are ASCII numeric values. | int iHashValue = 0;  char szAbc123[7];  int i;  //Assume szAbc123 is populated.  for (i = 0; i < strlen(szAbc123); i++)  {  iHashValue += szAbc123[i];  }  iHash = abs(iHashValue) % N;  // Alternate solution  for (i = 0; i < 3; i++)  {  iHashValue += szAbc123[i] - 'A';  }  for (i = 3; i < 7; i++)  {  iHashValue += szAbc123[i] - '0';  }  iHash = iHashValue % N; |
| **What is the best hashing function for all cases?** |  |
| **Collisions**  Unfortunately, the mapping of M values into N values can cause collisions where more than one key value is mapped to the same hash result.  **Synonyms** are keys that hash to the same hash result.  There are many approaches to resolving collisions. | From Example #1 (assuming N is 8):   |  |  | | --- | --- | | iKey | iHash (iKey % 8) | | 36 | 4 | | 18 | 2 | | 72 | 0 | | 6 | 6 | | 43 | 3 | | 85 | 5 | | 38 | 6 | | 26 | 2 |   Hash Table:   |  |  | | --- | --- | | 0 | 72 and other data | | 1 |  | | 2 | 18 and other data | | 3 | 43 and other data | | 4 | 36 and other data | | 5 | 85 and other data | | 6 | 6 and other data | | 7 |  |   What do we do with 38 and 26? |
| **Collison Resolution**  **Open Address/Linear Probing -** look for the next free slot in the Hash Table and place the synonym there. Note that this can cause additional conflicts with keys that hash to that previously free slot.  To avoid a clustering of values, we can probe for an open slot based on some constant k.  Open Addressing impacts searching for a key since it isn't necessarily in the hash result slot. | Example 4: Open Addressing and Linear Probing  Suppose 55938 and 44238 both hash to 38.   * If k=7, we can place 44238 in slot 45. * Of course, values that hash directly to 45 are impacted. * Note that if 45 had been occupied, we would have placed 44238 in the first open slot found by adding k. Try slots 52, 59, …   Hash Table:   |  |  | | --- | --- | | 0 | … | |  | | 38 | 55938 | | 39 | 72139 | | 40 |  | | 41 | 12341 | | 42 |  | | 43 |  | | 44 | 45644 | | 45 | 44238 | | 46 |  | | 47 |  | | 48 | 33348 | |  |  | |
|  |  |
| **Collisions**  Collisions happen when two keys hash to the same integer. To handle collisions, databases typically use an overflow area with chaining. | **Example 2: Overflow Chaining -** only one of the synonyms is in the primary. All others are in the overflow area.  Suppose 55938 and 44238 both hash to 38.   * Assume slots 51 thru 70 are the overflow area. * We have a freeHead which points to the first free entry in the overflow area. * Slot 52 is open so we place 44238 there. * With chaining we link slot 38 to slot 52.   If we have another synonym at 38, we would continue a linked list in the overflow area.  Hash Table: freeHead was 52 and becomes 53   |  |  |  | | --- | --- | --- | | 0 | ... |  | |  |  | | 38 | 55938 | 52 | | 39 | 72139 | -1 | | 40 |  |  | | 41 | 12341 | -1 | | 42 |  |  | | 43 |  |  | | 44 | 45644 | 51 | | 45 | 77745 | -1 | | 46 |  |  | | 47 |  |  | | 48 | 33348 | -1 | | 49 |  |  | | 50 |  |  | | 51 | 44444 | -1 | | 52 | 44238 | -1 | | 53 |  | 54 | |  |  | 55 | |  | ... |  | |  |  | | 70 |  | -1 | |
|  |  |
|  |  |
|  |  |

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