Articles / Languages / C++

C++ Windows application

A Lightweight Indexed File for Storing Persistent Data and Objects

Richard Chambers

4.79/5 (6 votes)

30 Dec 2011 CPOL 14 min read 34.6K 924

A small function library for managing a file containing data records that have unique keys or indexes.

Download source code - 15.8 KB

Introduction

Sometimes there is a need for a way to store simple data objects into a file which can then be retrieved using a key or index. In many applications with such a need, an embedded database may be used. In some cases, using an embedded database is more than necessary and in addition, using the database increases the complexity of deployment. The embedded database becomes another component which has to be managed.

While working with the source code of an older point of sale application, I came across the implementation of a lightweight file storage mechanism for storing multiple instances of a given data structure using a simple key to identify the specific data instance to be accessed within the file. In the case of this point of sale application, the data stored in this fashion was a list of cashiers (each cashier's record providing data for controlling cashier operations), a list of employees (each employee's record providing data for employee time keeping), and other data. These data objects had the following characteristics in common:

1. the actual data stored was a simple C language binary data struct,
2. each object of a particular type had a unique key such as an employee identification number,

3. there were few changes to the elements of the data objects during operations once the objects were created and stored,
4. most of the time the files were created and a set of objects inserted into the file and the number of objects did not change often, and
5. there were usually only a few (less than 100) individual data objects of a particular type.

Background

At the time the original application was written, the developers did not consider a lightweight, embeddable database engine. The original application environment was a small microprocessor terminal with a small amount of memory using a simple, proprietary Operating System. The same pattern for file storage of objects was used in several places in the source code though the source text was cloned and modified rather than the developers creating a single, generalized function library to be used in multiple places. For one type of data object, the maximum number of which was changed by an order of magnitude, a B-tree style database function library was introduced, however the other files were not changed.

The basic pattern used in the data storage in the point of sale application was to have individual files for each data type. The file contained three data areas:

1. file description header containing file content management data,
2. an index section that had a finite size and which contained the index data used to lookup a particular data record, and
3. a data record section that contained the actual data being stored.

The number of indices available was fixed at the time an empty file was created and the sizing information was stored in the file description header. The fixed size would not ordinarily be a problem with the application as people setting up the point of sale terminal for an end customer would typically size the maximum number of data objects (cashiers, employees, etc.) much larger than the actual expected number of objects needed. For instance, the installation person configuring a setup for a small restaurant with 10 employees would size the employee file for 100 employees.

The index information is stored in the index area of the file as a sorted list. When a particular index is retrieved, a binary search of the sorted list of indices is used to locate the proper index, should it exist, and then use the index to locate the data record itself. When new records are stored, the insertion point for the new index is located and the index is inserted into the list at that point. When existing records are deleted, the existing index is located and the index is removed from the list and the list consolidated. This makes the insertion and the deletion of records expensive because the index list must be maintained in a sorted order without holes or empty index slots in the list. However if most operations used are read operations or record update operations, this structure is workable and simple for smaller data sets.

The implementation of these file access primitives within the point of sale application had a number of issues. First of all, instead of a generalized implementation, each of the data types had their own implementation of the generalized concept, resulting in source text being duplicated and increasing the size of the source text itself as well as the resulting size of the binary files. Secondly, using the file access primitives involved multiple steps during which the programmer needed to know a fair number of details about the file structure.

The source code provides a generalized function library with an interface to create a file and perform insertions, deletions, updates, and retrievals of the data in the file. The steps to use this library are:

1. determine the data object record and the data object index that provides a unique key for the data object record,
2. determine the maximum number of objects that will be stored, and
3. determine a comparison function that will compare two indices and indicate the collation order of the indices.

The comparison function will determine the sorted order of the indices.

The file management data is separated from the user's data in this implementation, and the file management data is encapsulated in the function library so that the user of the library does not know about the actual physical file implementation. The user is only concerned with the data that is stored and the semantics of the data access. There are two layers of accessing the data interfacing functions, a layer in which the file path name is specified and a lower level in which the file handle is specified. The layer that uses the file path name uses the lower level file handle layer by opening the specified file and then calling the appropriate file handle function to do the actual operation. For large amounts of multiple accessing, opening the file once and then doing multiple operations using the file handle layer is significantly less than using the file path name layer with the file opening overhead.

In order to separate out the file content management duties from the file content itself, file creation involves a step of creating the index section of the file before the file is first used. Each index entry contains two pieces of data, a record block number (a number from one to the maximum number of records the file will contain) and the user's index data. The initializing process involves writing the indices each with its record block number and a zeroed out section to be used for the user's index data. As new records are added or existing records are deleted, the indices with their assigned record block number are moved and the user data portion of the index is modified, however the assigned record block number assigned to a particular index is never changed. The indices are managed as a pool of handles to the actual record storage space.

Because the file and record management data is separated from the user's data, the user of the library does not know the physical location of the index or the record data to which the index references. The way that the index and the corresponding data record are linked means that the actual data record may or may not contain the index data. It also means that the user could use the library to create a file that is composed only of index data without a corresponding record.

Using the code

The function library provides basic file record manipulation:

1. create and initialize a file,
2. insert indices and records into the file with the indices ordered by a collation function,
3. delete an index and record from the file when a collation function indicates an index from the file matches the search index,
4. retrieve a record from the file by specifying an index with a collation function, and
5. update an existing record whose index matches a specified index.

Using the function library, the user will provide at least one data structure, the index, and possible two others that are optional, the header information and the record data structure used for data that is stored using the index as a key.

The file access operations (insertion, deletion, retrieval, and update) require two pieces of information that are used to determine the location of the index in the list of indices: the index data struct provided and a collation function that indicates whether the index data struct provided is above, below, or equal to an index data struct retrieved from the file. Using this method, the data type for the key can be whatever the user of the library desires so long as there is a collation sequence for the keys and there is a function that can compare two indices and determine whether one is higher or lower in the collation sequence than the other.

Some caveats and quid pro quos

The function library does have a couple of data structures and procedures that will constrain the size of the index struct. The size of the index struct should be small meaning less than 256 bytes because there are temporary buffer areas in the function library that are used for holding index data temporarily. In order to improve throughput, the index area of the file is shifted in multiples of the stored index entry size. The stored index size is the size of the user's index data struct along with an additional area that contains the record block number assigned to the index.

The actual record data associated with the index and which is stored in the record data section of the file is never moved the way the indices are moved. When a record is deleted, what is done is that the index associated with the record is copied to a temporary location, the index area is shifted or rippled down overwriting the index of the record that is being deleted, and the index in the temporary area is then put into the index area after the sorted list of indices that are active. The result is that indices are never destroyed, when needed they are pulled from the pool of available indices that are stored above the sorted list of in use indices, and when not needed they are put back into the pool of available indices. The result is that the records stored in the record area are not in a sorted list the way that the indices are kept. So after a number of insertions and deletions though, the indices will be in sorted order, the record data will not.

The current implementation of this function library does not overwrite the record data when its associated index is put into the free pool when the record is deleted. This could be a security concern for sensitive data.

The underlying file access primitives used in this function library are those from the Standard C Library as provided with the Microsoft Visual Studio 2005 Integrated Development Environment (`fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()`, and `fflush()`). Changing the file access primitives would require modifications to the function library internals. The external interface functions that use a file name expect the file name to be ANSI single byte zero terminated character strings. In applications requiring the use of UNICODE file names with a different set of file primitives, the function interface would change.

Creating and initializing the file

Creating and initializing the file requires that the user specifies the maximum number of records and provides the sizes for the index and for the record. A file content header that allows the user to store file content management information such as a version number or other data can also be specified. In some cases, the user of the library may want to query the file to determine the maximum number of records that can be put into the file as well as how many records are currently in the file. A secondary file information structure is provided in the header file to allow this inquiry to be made.

The intent for a person using the library is that they will have a C or C++ struct or class that will be used as a description of the actual objects stored, the record data, and a C or C++ struct or class that will be used as a description of the index object used to identify a specific record. So when the function is called to create and initialize the file, the easiest way of doing so is to use the `sizeof()` operator in the function call.

C#

```
// Create and initialize the data file. The name of the file to be created is "testname1"
// We are sizing this to hold 5,000 records. The index struct is of type IndexThing and
// the record struct is of type RecordThing. We are using a user defined header as well.
FileLibCreate ("testname1", 5000, sizeof(IndexThing), sizeof(RecordThing),
sizeof(HeaderThing));
```

Accessing and modifying the file contents

The function library provides a set of functions to insert a new record, delete an old record, or to update an existing record.

C#

```
// Create variables for specifying the index and the record
IndexThing MyIndex;
RecordThing MyRecord;

// Create a record with it's index.
MakeIndexAndRecord (iLoop, MyIndex, MyRecord);

// Now do the insertion of the record into the file. We specify the
// address of the variable containing the index and the address of the
// variable containing the record. In addition we must provide the address
// of a collation function whose prototype is int CollateFunc (void *pIndex1, void *pIndex2)
FileLibInsert ("testname1", &MyIndex, &MyRecord, CollateFunc);
```

All of these primitives use a collation function to locate either an existing record or the location within the sorted list that a new record should go. The prototype for the collation function is `int CollateFunc (void *pIndex1, void *pIndex2)` and this function will return one of three values. If the index `pIndex1` is lower in the collating sequence for the indices than the index `pIndex2`, then the collating function should return a value of negative one (-1). If the index of `pIndex1` is equal to the index `pIndex2`, then the collating function should return a value of zero (0). If the index of `pIndex1` is higher in the collating sequence than the index of `pIndex2`, the the collating function should return a value of positive one (+1). The values returned by the collating function should be similar to the values returned by comparison functions in the Standard C Library such as the string comparison function `strcmp(s1, s2)` or the memory comparison function `memcmp (b1, b2, n)`. In fact, if the key is a text string, the collation function may be nothing more than a use of the string comparison function `strcmp (s1, s2)` with the collation function returning the return code from `strcmp()`.

C#

```
// Collation function to determine the order of the two arguments.
//  - returns -1 if pIndex1 should go before pIndex2
//  - returns 0 if pIndex1 and pIndex2 have the same key
//  - returns +1 if Pindex1 should go after pIndex2
static int CollateFunc (void *pIndex1, void *pIndex2)
{
    IndexThing *pLookFor = (IndexThing *)pIndex1;
    IndexThing *pFileItem = (IndexThing *)pIndex2;

    // do a string compare on the two keys to determine if pLookFor is lower
    // in the collating order than pFileItem (strcmp returns -1), if the two
    // keys are equal (strcmp returns 0), or if pLookFor is higher in the
    // collating order than pFileItem.
    return (strcmp (pLookFor->aszKey, pFileItem->aszKey));
}
```

Points of interest

When testing, I tried using a file with five thousand indices and found that the open and close overhead was substantial when inserting five thousand records into a file. By separating the file open logic from the actual file manipulation function creating a layer that used a file handle to specify the file rather than the text name of the file, the time taken to generate a file with five thousand records on a laptop was significantly reduced.

The search procedure used to find a specified index in the sorted list of indices uses a binary search to probe the list of indices stored on disk to determine a small window of indices. At the point where the binary search probing narrows the list to a subsection, the subsection is read into memory and searched sequentially. The reason for this is that searching a sequential list in memory is quicker than searching a disk file that is stored on an electro-mechanical device with quite a few delays involved in locating and retrieving a particular area on the disk. An interesting experiment would be to change the procedure used so as to increase the size of the subsection and to use a binary search for the in memory search as well.

While working on the test harness and demonstration source code showing how the library could be used, I realized that the various files were analogous to tables in a database. This insight led to the adding of an additional function to the library to iterate over a file allowing a filter or selection function to be used to select indices that matched a criteria other than the collation function criteria. By adding this iterator functionality, we now are able to use foreign keys with a more database oriented approach when using this function library.

Another interesting piece of data that may be useful would be to add to the file management data that is stored in the file, using the struct `FileLibHeader`, information about file usage such as a record of the date and time of the last insert, the last delete, and the last update. Other information that may be useful would be to keep a count of the number of inserts, deletes, updates, retrieves, etc., in the header.

License