CDBFile v. 1.0, a C++ package for handling dBASE III files DOS/Windows version

Hervé GOURMELON, ENSSAT Lannion

March 25, 1997

Contents

1 Introduction

Last term, I had to begin a reengineering project for the ENSSAT, the French graduate school I am working at. The core of the project was a software that drives automatic doors with magnetic cards, which are referred to in data files. Well, I had long thought that the **dBASE III** file standard was no longer in use since **dBASE IV** and **dBASE V** had appeared. It turned out that the former software that was running those doors seemed to work very well with those DBF files, and yet their structure was quite simple. I could have chosen to use raw text files for that application, which woud have made it a great deal easier to program. But there are quite a few companies and administrations in France that used the same system, and I thought that it would be easier for them to migrate to a new software if they did not have to modify the structure of their own data files.

Browsing the Net for some documents or some code, I discovered that hardly anything had been released under the GPL[1] in that field. The only thing I found was Mark W. Schumann's `"ffld.c"`, but I needed more than that, and I wanted those tools to be object-oriented. So I decided to distribute the few tools I would build under the GPL (please read the `COPYING` text file included in the package). They are far from being comprehensive, but I hope that their modularity will enable other developpers to expand them and/or customize them as they like...

2 The dBASE III Database file structure

The structure of a dBASE III database file is composed of a header and data records. The layout is given below.

BYTE	CONTENTS	MEANING
0	1 byte	dBASE III version number (03H without a .DBT file) (83H with a .DBT file)
1-3	3 bytes	date of last update (YY MM DD) in binary format
4-7	32 bit number	number of records in data file
8-9	16 bit number	length of header structure
10-11	16 bit number	length of the record
12-31	20 bytes	reserved bytes (version 1.00)
32-n	32 bytes each	field descriptor array (see below)
n+1	1 byte	0DH as the field terminator

Figure 1: dBASE III database file header

BYTE	CONTENTS	MEANING
0-10	11 bytes	field name in ASCII zero-filled
11	1 byte	field type in ASCII (C N L D or M)
12-15	32 bit number	field data address (address is set in memory)
16	1 byte	field length in binary
17	1 byte	field decimal count in binary
18-31	14 bytes	reserved bytes (version 1.00)

Figure 2: A field descriptor

[1] Well, I discovered later that a dBASE file viewer also existed, written in C for UNIX/Linux...

The data records are layed out as follows:

1. Data records are preceeded by one byte that is a space `(20H)` if the record is not deleted and an asterisk `(2AH)` if it is deleted.

2. Data fields are packed into records with no field separators or record terminators.

3. Data types are stored in ASCII format as described in Figure 3.

DATA TYPE	DATA RECORD STORAGE
Character	(ASCII characters)
Numeric	- . 0 1 2 3 4 5 6 7 8 9
Logical	? Y y N n T t F f (? when not initialized)
Memo	(10 digits representing a .DBT block number)
Date	(8 digits in YYYYMMDD format, such as 19840704 for July 4, 1984)

Figure 3: Data type storage

3 Description of the object

3.1 Specification

For the project I was working on, I had to process various dBASE files that were very different in their structures. The easiest way to solve that problem was to create a set of tools that would be completely generic, smart enough to process various files with various kinds of data. I thought that there could be a specific object for the file, which would contain at least the whole description of the header.

The next problem was : how will I store the records? The first idea that springs to mind is :

> ”Okay, let’s create a generic structure containing a dynamic list of fields, each of which will contain the data. Every field in the list will have a type identifier, a buffer containing the data, and a pointer to the next field. Every record will be a list of those fields, and the records will also be stored in a dynamic list”.

This might be an elegant solution, but it is very heavy: suppose you have 256 fields in each record, each of which containing only one byte of data... With all the pointers involved in the dynamic structures, the data could take up to ten times as much space in memory as on the disk!

In order to save memory, I decided to keep a simple structure: every record is stored as a string of raw ASCII read directly from the disk, into a dynamic list of records. The description of the fields (with their types and lengths) is stored in a separate list. Values are read from and written to the records using functions which format the data according to the field types.

This solution consumes less memory and is easier to implement. There is a price to pay, though, for the genericity of the tools: using the functions to access the data (reading from / writing to a record) you have to pass or to receive a void pointer, so the programmer has to know exactly which is the type of the value that is passed or received.

3.2 Structure

The CDBFile object structure is layed out as described in Figure 4. It contains two dynamic lists: one list of field descriptors (pointers to CField objects) and one list of records (pointers to Record structures).

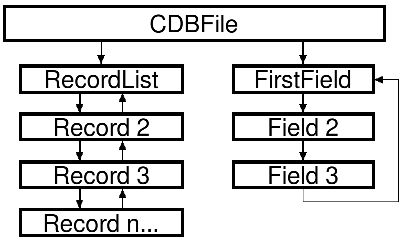

Figure 4: CDBFile internal structure

The field descriptors are organized in a ring: the last CField pointer points to the first CField* in the list. The ring is single-linked. I chose to implement it that way because it is easier to maintain than a double-linked dynamic list, and quite efficient, given the few number of fields that we have in general.

The records are stored in a double-linked list. I tried to use a single-linked list first, but it turned out that I needed a second link in order to apply the quick sort algorithm to the list. Pity.

Another thing that worries me about the quality of the tools is that I did not declare Record as an object, but only as a structure (I thought I didn't need to, given the little data contained in that structure). **R**eal **C++ P**rogrammers won't like it at all... Maybe in a later version I (or someone else) will fix that.

Nota: in the next subsections, I will describe the various components of the CDBFile object and their behavior. That description will only be a quick overview of what every member function does, so if you really need more details, have a look at the code in "cdbfile.cpp", which is thoroughly commented and should help you understand my somehow foggy and twisted ideas.

3.3 The CField object

It is described in the header file as follows :

```
class CField
{
private :
  char            Name[11];    // field name in ASCII zero-filled
  char            Type;        // field type in ASCII
  unsigned char   Length;      // field length in binary
  unsigned char   DecCount;    // field decimal count in binary
  unsigned char   FieldNumber; // field number within the record
  unsigned short  Offset;      // field offset within the character string
  CField*         Next;        // Next field in the list
  ...
};
```

You can see that all the data that can be found in the field descriptors is stored in those CField objects (apart from the field data address, which is useless when you are not using the first versions of **Ashton-Tate's dBASE**). An additional value has been appended: the Offset field, which indicates the beginning of the field within the raw character string. The CField* Next pointer points to the next field structure in the ring. Every member variable can be read or written to using the associated public member functions. Two special functions (in fact, two overloaded versions of the same member function) are provided to access the right CField object in the ring, using either its name or its number to identify it. These are recursive functions which need a `CField` pointer to return the position of a `CField` object within the ring — see the code for more details. So here are the headers for those member functions :

```
class CField
{
private :
...
public :
  CField();                     // default constructor
  CField(char* NName, char NType, unsigned char NLength,
    unsigned char NDecCount, unsigned char FieldNum);
                                // another constructor
  ~CField();                    // default destructor

  char* GetName()               { return Name; }
  void SetName(char* NewName)   { strcpy(Name, NewName); }
  // ...and so on for all the member variables

  CField* GetField(char* FieldName, CField* Start=NULL);
  CField* GetField(unsigned short Number, CField* Start=NULL);
};
```

That's all for the `CField` object, which you won't have to use directly — unless you want to add some more functions to CDBFile.

3.4 The CDBFile object

Now let's get down to some serious work. This is the main part of the toolbox, and only a few member functions of `CDBFile` are available for the user (i.e., the programmer who doesn't want to handle raw DBF files barehands!). This object manipulates four major entities :

- **The CField ring:** see the description of `CField` above.
- **The Record list:** A double-linked list of structures containing the data in a raw ASCII format, straight from the file. This is the core of the object, and most of the member functions will manipulate that list.
- **The contents of those records:** A character table containing raw data. Since we know the offset, the length and the type of each field, we can convert that data into numbers or character strings, or even booleans; we can also do the reverse operation in order to write or modify a value within a record.
- **The DBF file:** it will be accessed for reading or writing records. It is *absolutely necessary* to open such a file in order to initialize the field descriptor ring, since that toolbox does not provide yet (shame! shame!) enough functions to build a `CDBFile` object from scratch.

3.4.1 Handling the record list

Here is the structure of the records, as declared in the source:

```
struct rec
{
  char* Contents;              // raw character string
  BOOL ModifFlag;              // TRUE if the record has been modified
  unsigned long RecordNumber; // Position of the record within the file
  struct rec *Next;            // Points to the next record in the list
  struct rec *Previous;        // Points to the previous record in the list
};
typedef struct rec Record;
```

This is the minimum structure that we need to process any kind of DBF records, provided that we know the field descriptor array. Let's have a look at the private variables and function headers below :

```
class CDBFile
{
private :
  Record* RecordList;   // Head of the list of records
  Record* CurrentRec;   // Current record pointed in the list
...
  Record* ReadRecord(unsigned long RecNum);
  Record* CreateNewRecord();
  Record* GetRecord(unsigned long RecordNum);
  void Append(Record* Rec, Record* Tail=NULL);
  void DeleteRecord(Record *Rec);
  void SortAllRecords(Record *Head, Record *Tail, CField* Criter1);
};
```

`CDBFile` contains two pointers to records: the former, `RecordList`, is the at head of the double-linked list of records, whereas the latter, `CurrentRec`, points to the record that is being handled by the user. `CurrentRec` is mainly used by public functions that give the user some possibilities to access the records indirectly (for reading or writing, creating or deleting).

As for the private functions, they do the following :

- `ReadRecord()` reads the `RecNum`th record in the DBF file, and returns a pointer to a newly created record containig the accessed data.
- `CreateNewRecord()` returns a pointer to a newly created record.
- `GetRecord()` points to the record whose number matches `RecordNum`.
- `Append()` inserts `Rec` in the list after `Tail`, if `Tail` is not `NULL`. Otherwise, it will insert `Rec` in the list according to its record number.
- `DeleteRecord()` simply deletes the record and its contents.
- `SortAllRecords()` operates a quicksort algorithm on the record list. The sorting is done on the criterium `Criter1`, which points to a field descriptor: the list will be sorted in increasing order of the `Criter1` field of the records.

The public functions are in fact a convenient way to embed the private functions without requiring the user to manipulate those ugly record lists or structures (yuck!), let alone the field descriptors (ugh!). He just has to instantiate a `CDBFile` object in his program, and mind his own business. All he has to know about how that object works is that there is a special pointer `CurrentRec` (see above) which points at the record he wants to access or to modify, and that `CurrentRec` can be moved all along the list very easily, in an iterative way. Here are the public functions that will enable the programmer to do so :

```
class CDBFile
{
...
public:
  unsigned long LoadFileToMemory();
  unsigned long WriteModified();
  void SortOn(unsigned short Criterium1);
  void GetAtRecord(unsigned long RecordNum)
     { CurrentRec=GetRecord(RecordNum); }
  unsigned long GetRecordNum()
     { if (CurrentRec) return CurrentRec->RecordNumber;
       else return 0L;}
   BOOL GetNextRecord()
     { if (CurrentRec) { CurrentRec=CurrentRec->Next; return TRUE;}
                  else return FALSE; }
   BOOL GetPreviousRecord()
     { if (CurrentRec) { CurrentRec=CurrentRec->Previous; return TRUE;}
                  else return FALSE; }
  void LoadRecord(unsigned long RecordNum)
     { CurrentRec=ReadRecord(RecordNum); Append(CurrentRec); }
  void DeleteCurrentRec()
     { CurrentRec=CurrentRec->Previous; DeleteRecord(CurrentRec->Next);}
  void CreateAndAppend()
     { CurrentRec=CreateNewRecord(); Append(CurrentRec);}
  void ClearAllRecords();
...
};
```

For most of those functions, the souce code is self-explanatory. Some others are less obvious: `SortOn()`, for example, launches the recursive quicksort function `SortAllRecords()` (see above). `LoadFileToMemory()` reads all the records on the file using `ReadRecord()` and stores the resulting records in the list. `WriteModified()` checks all the records in the list and writes some of them to the file if they have been edited or modified. `ClearAllRecords()` empties the list and deletes all the records. Some more complex functions also involve manipulating the contents of those records (`SortAllRecords()` is one of those), or the DBF file. They will be explained in the next paragraphs.

3.4.2 Handling the contents of the records

We have seen the structure of the records in the previous section; let's have a look at the functions that can access the contents of those records.

```
class CDBFile
{
...
private:
  void* GetFieldValue(Record* Rec, CField* Field);
  void SetFieldValue(Record* Rec, CField* Field, void* Value);
  BOOL IsBigger(void *v1, void *v2, CField* Criterium);
  BOOL IsSmaller(void *v1, void *v2, CField* Criterium);
...
};
```

The first one is `GetFieldValue()`. That function takes two arguments: a pointer to the record that is being accessed and a pointer to the field descriptor (the field that contains the accessed value). Knowing the offset and the length of the field within the contents string, the function first extracts the corresponding substring. Then it checks the type of the field. If the field is of type `'N'` (numeric value), there are two cases: if its decimal count is null, then the substring will be converted into a `long` integer value; if not, it will be converted into a `double` floating-point value. If the field is of type `'L'` (logical value), the substring will be converted into a boolean value (defined as `BOOL` in that package, and in Micro$oft's Visual C++). In other cases, if the field is of type `'C'` (character string) or `'D'` (date), or even `'M'` (Memo number for .DBT blocks) , there will be no conversion, the character substring will be kept 'as is'. Maybe some specific conversions would be useful, particularly for dates or memo block numbers, but I did not need those functionalities for my own application. It's up to you to add more conversions, in respect of the GPL, of course.

In all cases, the result of the conversion will be put in a dynamically allocated pointer of the corresponding type, and that pointer will be returned as a `void` pointer. This is the inelegant aspect of the function: the programmer has to know exactly what kind of data he is receiving in that `void` pointer, and he also has to delete that pointer (it has to be deleted, since it has been allocated dynamically). If he is using a fixed type of records (for a specific application, an address book for example) this will not be a problem. But if the type of records can vary in that application (a generic spreadsheet application, for example), the programmer will probably have to verify the type of the data, using public functions such as `GetFieldType()` (see below).

The same canvas applies to `SetFieldValue()`, the other way round: when calling that function, a `void` pointer containing the value to be set is passed, along with a pointer to the field descriptor. The function then converts the contents of the `void` pointer to the right substring, and copies it to the contents of the record.

The other two functions, `IsBigger()` and `IsSmaller()`, use `GetFieldValue()` to compare two records on the basis of their respective values for a given field. They are called by `SortAllRecords()` only so far.

The public functions that can be used by the programmer are the following:

```
class CDBFile
{
...
public:
  void SetFieldValue(char* Field, void* Value);
  void SetFieldValue(unsigned short FieldNum, void* Value);
  void* GetFieldValue(char* Field);
  void* GetFieldValue(unsigned short FieldNum);
  char GetFieldType(unsigned short NumField)
     { return (FirstField->GetField(NumField))->GetType(); }
  void DeleteVoidPointer(void* Pointer, unsigned short Field);
  void DeleteVoidPointer(void* Pointer, char* Field);
...
};
```

The overloaded and public versions of `SetFieldValue()` and `GetFieldValue()` are the equivalent of their private versions, but they use either the name or the number of the field to refer to it. `GetFieldType()` enables the programmer to know the type of the field referenced by `NumField`. I also chose to create `DeleteVoidPointer()` so that the programmer can delete easily the `void` pointers he receives from calls to `GetFieldValue()`.

That's all about handling the contents of the records, let's have a look now at how the files are handled.

3.4.3 Handling the files

As I said earlier on, I did not implement the possibility to create a DBF file from scratch, adding fields in the field descriptor ring and the records or removing them. Therefore, one has to open an existing DBF file first, then one can only add, edit or remove records from that file, and save it or save it as a new file (with another pathname).

When referring to Section 2, the header of the file contains a few specific parameters that describe the characteristics of the file. They are implemented in CDBFile as private member variables:

```
class CDBFile
{
...
private:
  char PathName[256];           // Path name for the dBase III file
  FILE *FileHandle;             // Handler for the dBase III file
  unsigned short HeaderSize;    // Length of header structure
  unsigned short FieldCount;    // Number of fields in each record
  unsigned long RecordCount;    // Number of records in the file
  BOOL ModifiedFlag;            // TRUE if the data has been modified
  BOOL FullFileInMemory;        // TRUE if the entire file is loaded
  unsigned short RecordLength;  // Length of the record strings

  Record* RecordList; // Head of the list of records
  Record* CurrentRec; // Current record pointed in the list
  CField* FirstField; // Head of the list of fields
...
};
```

Some of them can be accessed and modified directly by the programmer, using dedicated public functions; most of them, however, are only accessed and processed internally by the member functions of the package, since the programmer should not need to read or modify them in any way. Here are the few functions that enable the programmer to do so:

```
class CDBFile
{
...
public:
  BOOL IsOpen()
     { return (BOOL)(FileHandle!=NULL); }
  unsigned long GetRecordCount()
     { return RecordCount; }
  unsigned short GetFieldCount()
     { return FieldCount; }
...
};
```

The other functions perform bigger tasks, of a higher level. Here they are:

```
class CDBFile
{
...
public:
  CDBFile();
  CDBFile(char* Path);
  ~CDBFile();
  BOOL Clean();
  BOOL OpenFile(char* Path);
  BOOL CloseFile();
  unsigned long WriteAllToFile(char *Path=NULL);

private:
  BOOL WriteHeader(char* Path=NULL);
...
};
```

The only private function here is used to rewrite the file header when some of its information has changed. It is also used to write the modified file under a new pathname.

The public function that uses `WriteHeader()` is `WriteAllToFile()`. That function is used only when the entire file has been loaded into memory and has to be rewritten or saved under a new pathname. Note that it contains non-portable time-encoding functions, which are specifically DOS-oriented, and will cause trouble when ported to UNIX / Linux. See the code in `"cdbfile.cpp"` for more details.

The other public functions perform the following tasks:

- `CDBFile()` is the default constructor, whereas `CDBFile(char* Path)` creates the object and opens the file known as `Path`;
- `~CDBFile()` is the default destructor;
- `Clean()` resets (or initializes) the contents of the `CDBFile` object, and is called both by the constructors and the destructor;
- `OpenFile(char* Path)` first resets the object and then opens a new file under `Path`; that function also sets most of the data members to the values mentioned in the file, and particularly, it builds the field descriptor ring.
- `CloseFile()` simply closes the file and resets all the data.

That's all for the description of the code. Again, if you need more details, RTFC[2].

4 An example: the TestDBF file editor

This is an elementary C-style console application, proposing a few tools to edit DBF files. It does not demonstrate *all* of the possibilities of `CDBFile`, but it shows you most of them. Also, note that not all the shortkeys are displayed on the top line of the console when you launch the program. This is a typically user-unfriendly (but hopefully hacker/programmer-friendly) application.

If you take a look at the source code (`"testdbf.cpp"`), you will have a better idea of how the `CDBFile` objects and its methods should be used within a program. The `"case 'e' :"` part of the `main()` function particularly deserves your attention: it shows how to implement a generic record editor using that package (which is non-trivial).

I will not comment more on that application: the best thing for you is definitely to have a look at the source code. *Hmmm... Didn't I say that earlier on? I'm not sure...* :-)

5 Conclusion

Well, that's all for the time being. I'm not quite sure that I will have time enough to support that package in the next few months, but I will do my best. However, if you have some questions (in spite of all the efforts I did to make that project understandable :-), comments or bug reports, just e-mail me at: `herve.gourmelon@enssat.fr` (until April 98).

As far as the portability is concerned, I only tried that package under Micro$oft's Visual C++. Porting it to Borland's or Watcom's should be no trouble, but beware of the definition of the `BOOL` type in `cdbfile.h`: you might have to modify it using pre-compilation directives. If you want ot use it under UNIX/Linux (using `cc` or `gcc`), you should be able to find a special GNU/Linux version that I released lately in the Linux part of the Sunsite archive (that version is easier to port, by the way),

[2] Read The F***ing Codefiles!

as follows:
ftp://sunsite.unc.edu/pub/Linux/devel/db/cdbfile.tar.gz

If you decide to improve significantly the code and to add new functionalities, please contact me: I will provide you with the LaTeX source of the postscript file you are reading, so that you can add the description of your new functions or objects and your name will be added to the contributor's file of the package, and to this page.

Have fun!

6 Contributors

Hervé GOURMELON
herve.gourmelon@enssat.fr

Erik Bachmann (for the GNU/Linux version)
ebp@dde.dk
http://www.geocities.com/SiliconValley/Pines/2563/

I hope that several other people will add their names to that list...