Garret Object Oriented Database Management System

Konstantin Knizhnik

March 16, 1999

Contents

1 Introduction **1**

2 Persistent object header **1**
2.1 Information about object type 2
2.2 Information about object size 2

3 Access to the objects **2**
3.1 Database as array of objects 2
3.2 Smart pointers 3

4 Database control methods **4**

5 File IO **4**

6 Fault protected storage **5**

7 System parameters **5**

8 Examples **6**

List of Tables

1 Introduction

GOODS is very simply single-user object oriented database management system with C++ language interface. GOODS provides work with arbitrary number of logical files, which can be splitted into several physical operating system files. Transactions and fault tolerance are also supported. To reach high performance GOODS uses object cache and pages pool. GOODS is very portable and works at many systems systems starting from MSDOS and up to Digital Unix.

Database is controlled by class `Object_Database`. Database consists of a set of storage (up to 32). Abstract class `Dbs_Store` provides interface to different storage implementations. Simple and fault tolerant storage are supported at this moment. Each storage should be connected with page pool. To eliminate operating system restriction for number of opened files, pool of opened channels is used. GOODS multiplex channels, providing access to arbitrary number of files.

2 Persistent object header

All persistent classes should be derived from class `File_Object`. This class provides methods common for all persistent objects:

- Change object status

```
void Modify(); // mark objects as been changed
void Save(); // write object to the storage
```

- Allocation of object in current storage

```
void* operator new(size_t Size);
void* operator new(size_t Size, o_size_t Ext);
// allocate object with varying length - add Ext bytes to object size
```

- Allocation of object in specified storage

```
void* operator new(size_t Size, Dbs_Store &Store);
void* operator new(size_t Size, Dbs_Store &Store, o_size_t Ext);
```

- Deleting of object

```
void operator delete(void* Addr);
```

2.1 Information about object type

Each persistent object can store unique object class identifier in object header. This information is needed when persistent classes with virtual functions are used in application. In this case programmer should assign unique identifier to each class and specify initialization table, which contains pointers to the functions-constructors performing object initialization (setting virtual table pointers). GOODS uses default constructor (constructor without parameters) for object initialization, so programmer should not define its own constructor without parameters.

But is not necessary to store type information in persistent objects. If you don;t define macro `TYPE_INFO`, then no type information will be stored in object headers and objects will become a little bit shorter. Also no initialization table should be specified in this case and default constructors can be used by programmer.

2.2 Information about object size

At the beginning of each persistent object there is a field containing length of object in bytes. By default two-bytes unsigned integer is used for storing object size (so maximal size of object is 65535). Programmer can redefine type `o_site_t` to support larger objects.

3 Access to the objects

GOODS can be used with standard C++ compilers and require no special preprocessor. All manipulation with persistent object is done through C++ classes. GOODS provides two approaches for accessing persistent objects: representing database as array of objects and using smart pointers.

3.1 Database as array of objects

Programmer work with database as with abstract indexed collection of objects of different classes using object identifiers as index in this collection. Object identifiers are stored in normal C++ pointers. This approach provides most flexible and most efficient way of working with GOODS database. It is possible to access object using two kinds of pointers: virtual pointers in database (containing object identifiers) and direct pointers (containing address of object in physical memory). GOODS provides macro `Odb_File_Object` to define methods for accessing objects for each persistent class:

`T& opeator()(T* ptr)`	access to object for reading
`T& opeator[](T* ptr)`	access to object for modification
`friend T* Fix(T* ptr)`	fix object in memory
`friend T* UnFix(T* ptr)`	unfix object
`friend T* Pos(T* ptr)`	getting object virtual address (OID)

When object is accessed through virtual address (OID), GOODS searches object cache for object with specified OID and if there is no such object in cache (cache miss), then object is loaded from the storage. It is possible to store physical address of fetched object in operating memory and access object directly through this pointer without any extra overhead. But object can be thrown away from object cache by LRU replacing discipline (least recent accessed object is replaced when there is no free space in cache). Object can be fixed in cache by `Fix()` method. But fixing large number of objects can cause exhaustion of system memory. Persistent object can refer other object only through object identifier (GOODS doesn't perform pointer swizzling).

This approach of accessing database object is very error prone. There are three kinds of possible errors:

1. Mix pointers and using virtual pointer instead of physical and visa.
2. Modify object, which was not marked as been changed by `operator[ ]` method.
3. Accessing object through direct pointers without fixing it in memory.

Error of both of these types are very difficult to detect and debug. That is why GOODS provides other approach of accessing database.

3.2 Smart pointers

GOODS provides more convenient and reliable approach of accessing persistent object. With this approach instead of normal C++ pointers, programmer should use objects of special classes - smart pointers. GOODS provides two implementations of smart pointers — using templates and using preprocessor. There are two kinds of smart pointer:

`Ptr<T>`
pointer to object in the storage containing OID)

`Tie<T>`
pointer to object in the memory (containing direct object address)

Inside persistent classes only Ptr pointers can be used. And to optimize access to the object it is possible to tie object (fix it in memory) using Tie class. Such type of pointers can be used for local variables or components of transient objects. All kinds of conversions between both types of pointers are performed automatically. Smart pointers always refer to default database.

Check of smart pointer for `NULL` can be done by `Nil` method:

```
friend bool Nil( Ptr& P );
friend bool Nil( Tie& P );
```

Object can be deleted by `Delete` method:

```
friend void Delete( Tie& P );
```

`NULL` value can be assigned to smart pointer either by normal `=` operator or by `Set_Nil` method:

```
void Set_Nil();
```

Reference to root object in the database can be retrieved by `Root` method:

```
static Ptr Root();
```

Object can not be thrown away from object cache until `Smart` pointers free programmer from fixing/unfixing objects in memory but can not detect object modification. Programmer still have to explicitly mark object as been modified by `File_Object::Modify()` method.

When persistent object constructor is executed, there are no references to the object from pointers of Tie type, so newly created object is not fixed in the memory. Newly created object can be thrown away from the object cache if other persistent object are created/accessed from the constructor. To prevent it, assign this pointer to `Tie` variable in first statement of constructor.

4 Database control methods

Class `Object_Database` defined the following methods for managing database session:

```
void  Open  ( void ); // open database
void  Close ( void ); // close database
void* Start ( void ); // get virtual address of root object
large Length( void ); // size of default storage
bool  Empty ( void ); // check if database is empty
void  With  ( Dbs_Store& Store ); // set default storage
void  With  ( int  Number );      // set default storage
void  Flush ( void );             // save modified objects
void  Throw ( Physical Address ); // remove object from cache
void  Near  ( Virtual  Address ); // specify preferable area
```

Class `Object_Database` has also static component `Default`, which is used to refer default database. This field is used while object creation and also when smart pointer is dereferenced.

Sequence of action needed for database creation depends on whether virtual functions are used in application persistent classes. If `macroTYPE_INFO` is defined, then the following steps should be performed:

1. Define unique identifiers for persistent classes using enum construction and `Odb_Tag(CLASS)` macro.

2. Define persistent classes and insert by means of macro `Odb_Constructor(CLASS)` declarations of database methods in the class.

3. For each persistent class define initialization functions by means of `Odb_Define_Initializer(CLASS)`

4. Create table of initialization functions using macro `Odb_Initializer(CLASS)`. Index of initialization function in the table should be the same as correspondent class identifier.

5. Create database class, specifying in its constructor array of storages, number of storages and pointer to initialization table.

If virtual functions are not used and so macro `TYPE_INFO` is not defined, then steps 1-4 can be skipped and last parameter of database object constructor (initialization table) is not needed.

5 File IO

Abstraction of file is used to provide system independent interface to files and also for eliminating operation system restrictions for number of opened files and maximal file size. Pool of channels is used to multiplex file descriptors. If while opening file is is detected that there are no free channels in the pool, then least recently used channel is closed and is used for new file. Channel pool is implemented by `Dbs_Channel` class and maximal number of channels in the pool should be specified in its constructor.

GOODS uses pages pool to provide efficient disk IO operations. Pages are managed by class `Dbs_Pool`. At the moment of pool creation it is possible to specify in contractor number of pages in the pool (default value is 4096). Pages can have size 512 bytes, 1k, 2k, 4k, 16k or 32k. Normally one pages pool is used for all storages.

GOODS allows your to split one logical file into several physical segments (operating system files). Class `Dbs_M_File` is derived from `Dbs_File class`. Its constructor accepts list of segment descriptor (`Dbs_Segment`), specifying name and length of each segment. Only last segment of multifile can be dynamically extended.

6 Fault protected storage

If system or application fault happens during database session, database can be left in inconsistent state. If you want to protect your database from faults, you should use fault tolerant storage `Dbs_P_Store`. This storage keeps undo transaction log, where original images of all modified pages are stored. Method `Flush()` writes all modified pages to the disk and truncates the log file. If fault is happened before all data is flushed, then next time database will be opened, recovery procedure will start. During database recovery all modified pages are replaced with original pages from the transaction log, so all uncommitted changes are undone and consistent database state is restored.

When fault tolerant storage is created, then in addition to the data file, it is necessary also to specify transaction log file in storage constructor. Distributed transactions (i.e. transactions involving objects from several storages) are not supported. If first storage flushes all modified pages to the disk and truncated log file and system fault takes place before second storage flushes all modified data and truncates the log, then rollback of transaction will be performed only for objects from the second storage.

7 System parameters

There are several parameters of the GOODS, which allows you to select needed database configuration:

`TYPE_INFO`
store information about object type in object header

`SMART_POINTERS`
use smart pointers for persistent object access

`NO_TEMPLATES`
do not use templates for smart pointer implementation

Object cache parameters can be set in `oodbs.cpp` file:

`ODB_CACHE_SIZE`
maximal number of objects in the cache

`ODB_CACHE_LIMIT`
limitation for total size of all objects in the cache

Size of the object allocated in the storage is aligned by GOODS on allocation quantum boundary. is located in file `store.hpp`. Default value of allocation quantum is 32 bytes. Constant `Dbs_Slot_Bits`, declared in file `store.hpp`, specifies power of two corresponds to the size of allocation quantum. As far as with 32 byte allocation quantum least significant 5 bits of object address within file are not used, then it is possible to use this bits to store storage number. So maximal number of storage in database is limited by number of unused bits in object address. Specifying smaller allocation quantum reduces maximal number of storages.

Changing of `Dbs_Slot_Bits` parameter should be done with care. Increasing this quantum, you also increases speed of memory allocation/deallocation (allocation algorithm have to look through smaller number of bits) and maximal number storages, but it also cause large internal fragmentation and leads to growth of database file. Decreasing this values reduces internal fragmentation, but makes allocation algorithms less efficient and also increases external fragmentation (a lot of small holes).

In file `store.hpp` it is possible to redefine type `o_size_t`, used for storing object size in the object header. By default unsigned `short type` is used.

8 Examples

There are four simple tests `test1`, `test2`, `test3`, `test4` which illustrates different database modes. All these tests do the same thing, when database is initialized theses tests read lines from standard input and store them in database. At all successive executions of test programs, they will output lines stored in database. File `test1.cpp` illustrates work with database as with array of objects, by means of redefined `( ) [ ]` methods. File `test2.cpp` illustrates usage of smart pointers (without templates). In `test3.cpp` template smart pointers are used. And `test4.cpp` is an example of storing classes with virtual functions.

Example `guess.cpp` is very simple game with some elements of artificial intelligence. Template smart pointers are used in this example.

The file `dic.cpp` is test for extensible hash table class `Dbs_Str_Hash_Table`.

Below is template of GOODS application:

```
#include <stdio.h>
#include "fstor.hpp" // use simple storage
#include "oodbs.hpp"

class My_Object : public File_Object {
    // declare some fields
};

Dbs_Channel DB_Vio   (10);  // pool for multiplexing 10 channels
Dbs_Pool    DB_Cache (64);  // page pool of 64 pages of 4k size
Dbs_File    DB_File  ("test.dbs"); // database data file
Dbs_F_Store DB_Store (DB_File, DB_Cache); // database storage
Dbs_Store*  DB_List = &DB_Store;     // used to provide pointer to pointer
Object_Database Db( 1, &DB_List );   // initialize database object

int main ()
{
    Db.Open();  // open database
    if (Db.Empty()) { // check if database was not yet initialized
        // initialize database
    } else {
        Tie<My_Object> root = Ptr<My_Object>::Root();
        // do something with database
    }
    Db.Close(); // do not forget to close database
    return EXIT_SUCCESS;
}
```

Look for new version at my homepage	E-mail me about bugs and problems
`http://www.ispras.ru/~knizhnik`	`knizhnik@altavista.net`