ObjectFile (V4.10)

Application Developers Manual

Contents

1. Introduction

ObjectFile was developed in order to satisfy the need for object persistence within application programs. This need is often far removed from that of a full blown OODBMS system. For many applications a full OODBMS would be a burden, both to application performance and to the development process.

ObjectFile is designed to be just another set of classes in your application. It has no pre-compiler and no binaries to link with. This makes it easy to integrate into your development environment.

1.1 Notes on this Manual

It is assumed that the reader has a reasonable knowledge of the **C++** programming language.

All of the code examples given in this manual are incomplete. This is done for clarity and conciseness. They are however, all extracted from the real examples supplied with ObjectFile. You should refer to these examples if necessary while reading the manual.

The word ‘file’ is often used instead of ‘OFile’.

1.2 Example Program

We will jump in at the deep end, by looking at an example program that uses ObjectFile. Do not worry if there is anything you do not understand. It will become clear after reading the rest of the manual.

The program is a simple address database. It allows you to add, delete and list the addresses. It uses the Person class, which follows.

The ObjectFile related code is printed in bold text.

```
#include "odefs.h"
#include <cstring.h>
#include <fstream.h>
#include "oufile.h"
#include "ox.h"
#include "person.h"

main(){

    try
    {

    cout << "ObjectFile Address Database Demo V0.0\n\n";

    // Create the file if it does not already exist, otherwise open it.
    OUFile *addressDB = new OUFile("address.db",OFILE_OPEN_FOR_WRITING|OFILE_CREATE);

    while(true)
    {
        char buf[256];
        char  menuItem;

        cout << " 1. Add\n"
             << " 2. Remove\n"
             << " 3. List\n"
             << " 4. Exit\n"
              << "Select ?";
        cin >> menuItem;

        switch(menuItem)
        {
        case '1' :  // Add
            {
                string firstName,surname,address,district,city,country;
                OId spouseId = 0;

                cin.getline(buf,256);
                cout << "First name ? ",cin.getline(buf,256),firstName = buf;
                cout << "Surname ? ",cin.getline(buf,256), surname = buf;
                cout << "Address ? ",cin.getline(buf,256), address = buf;
                cout << "District ? ",cin.getline(buf,256), district = buf;
                cout << "City ? ", cin.getline(buf,256), city = buf;
                cout << "Country ? ",cin.getline(buf,256), country = buf;
                cout << "Spouse id.(0 = none) ? ",cin >> spouseId;

                // See if there is a spouse in the file.
                Person *spouse = (Person *)addressDB->getObject(spouseId);

                // Create the object.
                Person *p = new Person(
                                        firstName.c_str(),
                                        surname.c_str(),
                                        address.c_str(),
                                        district.c_str(),
                                        city.c_str(),
                                        country.c_str(),
                                        0,0.0,0.0,0,
                                        spouse,0);
                // Add object to file.
                addressDB->attach(p);

                if(spouse)
                    spouse->setSpouse(p);

                // Commit unwritten objects. This means data can never be lost.
                addressDB->commit();
            }
            break;
        case '2' :  // Remove
            {
                OId id = 0;

                cout << "Remove record id ?",cin >> id;

                Person *p = (Person *)addressDB->getObject(id,cPerson);
                if(p)
```

```
                {
                    // Detach the object
                    addressDB->detach(p);

                    // Make sure the spouse does not point to a deleted person.
                    Person *spouse = p->spouse();
                    if(spouse)
                        spouse->setSpouse(0);

                    delete p;
                }
                else
                    cout << "Record not found !\n";
            }
            break;
        case '3' :  // List
            {
                // Person iterator
                Person::It it(addressDB);
                Person *p;

                // Iterate over all Person objects.
                while(p = it++)
                {
                    // Print ascii representation of the object.
                    p->oPrint(cout);
                }
            }
            break;
        case '4' :  // Exit

            // Commit unwritten objects
            addressDB->commit();
            // Save file.
            addressDB->save();

            // Close the file
            delete addressDB;

            return 1;

        default:
            break;
        } // end switch
    } // end while


    } // end try
    catch(OFileErr x)
    {
        // Primitive error handling
        cout << x.why() <<'\n';
        return 0;
    }

}
```

2. Concepts

2.1 Object-Oriented Application Development

There are many books on object oriented design. It seems that more are being added every day. This manual will not attempt to cover the topic. If you are new to the subject then this is not the place to start.

2.2 Class Identity

Every persistent class in ObjectFile must be given a unique identity. This must take the form:

```
const OClassId_t cMyClass = <n>;          //  10 < n <  32000
```

By convention the constant name is the class name with a c at the beginning. But any other convention can be used. It is best to concentrate constants for all the classes in a single header file.

2.3 Object Identity

Every persistent object in ObjectFile has a unique identity within the **OFile** to which it is attached. By default the identity is a 32-bit unsigned integer. This gives 2^32 possible identities (more than 4 billion). For most purposes this gives a more than adequate lifetime to an application. However since **OId** (Object Identity) is a typedef, it can be set to a 64 bit integer, or any other structure, to give a far longer lifetime.

The object identity allows the application to maintain relationships to objects that are not in physical memory. An identity can be translated to a object pointer at any time. This causes it to be brought into memory, if it is not already there.

An object that is not attached to a file, is identified only by its memory address.

You may be tempted to use the **OId**'s of objects for various purposes not directly connected with ObjectFile. Some advice - don't.

2.4 The Persistent Object

The persistent objects used by ObjectFile must be created from classes derived from **OPersist**. **OPersist** provides the basic functionality needed by ObjectFile to manage the persistence of the object. In addition, derived classes may have to override certain virtual functions in order to support special attributes present in the sub-classes.

A persistent object is very similar to any other **C++** object. It has a few extra additions, all of them standard **C++** statements. (ObjectFile avoids the use of preprocessor macros wherever possible)

ObjectFile objects can be of variable length. There is no overhead involved in changing the length of an object. ObjectFile automatically recalculates the size of the object, before writing it to the file.

Consequently there is no problem writing the following class definition:

```
class Employee : public OPersist
{
    char * _name;     // or better: string  _name;
    long _salary;
};
```

This gives a lot of flexibility when designing persistent objects.

ObjectFile objects can contain both transient and persistent attributes. Which attributes are persistent, is defined by the **OPersist::oWrite()** method and the read-constructor.

Dirty Objects

An object is said to be dirty, when some of its data in memory, is different from its data on the disk file. The developer decides when this is the case by calling **OPersist::oSetDirty()** for the object. There is no harm in calling this many times for an object.

When **OFile::commit()** is called, only those objects marked 'dirty' will be updated in the physical file. Others will remain unchanged.

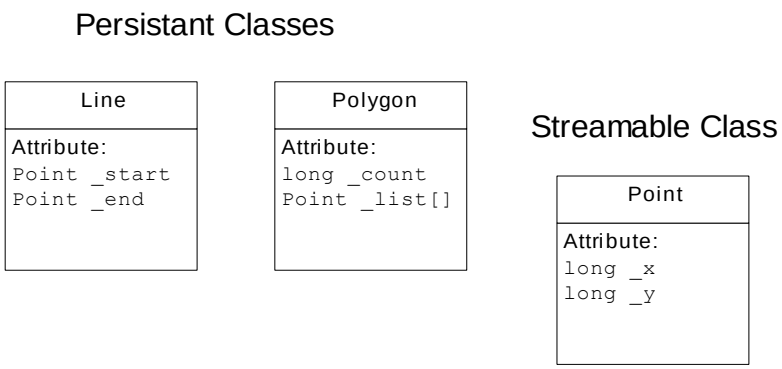
Purgeable Objects

When you make an ObjectFile object purgeable, you are saying "as far as I am concerned it can be thrown out of memory". In **C++** this means that any pointers to it are immediately invalidated. For the sake of efficiency ObjectFile does not immediately throw the object out of memory. It does so when the application program calls **OFile::purge()**. It is prudent to assume that any pointers are invalid, the moment the object is marked purgeable.

An object is made purgeable by the **OPersist::oSetPurgeable()** method.

Not every application will need to purge objects. Others will only need the feature in certain classes. In these cases, the methods available for purging, can just be ignored or used in very selective places.

2.5 The Streamable Object

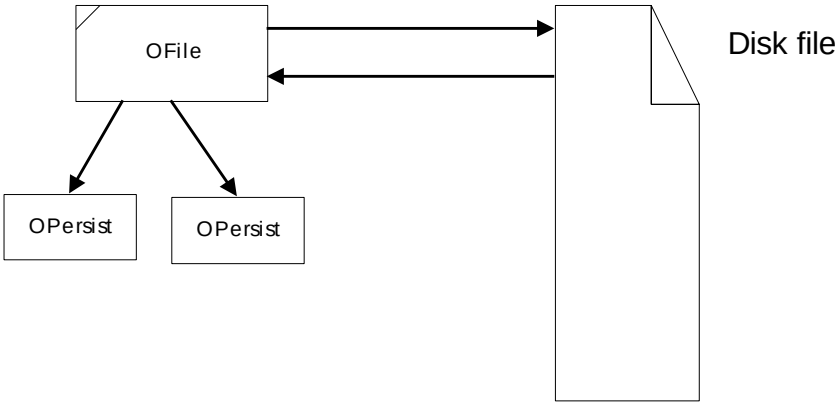


Not every object in a C++ application needs to be indexed separately. A good object-oriented application often has many objects that are only used as components or attributes of larger objects. A typical example would be the Point object, used to compose various types of geometric entities. A Line object would need two points to define its start and end. A Polygon object would need a list of Points. A Point however would never exist on its own. It is unnecessary and inefficient to hold indexes to every Point object. Point would therefore be made streamable. This involves adding a read constructor, and an **oWrite** method. The object of which Point is a component can then call these methods.

Making an object Streamable effectively gives you a means of creating arbitrary persistent data types.

Another major advantage of this is that you can make your favorite data types streamable, either by directly adding the read-constructor and write methods, or by creating a sub-class, and adding them there. You can also just initialize and write them from the enclosing classes methods.

2.6 The File



OFile is responsible for overall management of the file. It can be regarded as a container of persistent objects. It provides a set of services that allow its users to attach and detach objects from it. These objects must be derived from OPersist.

OFile provides only the most basic and essential file management services. This helps keep it lean and mean. Other, more sophisticated, and perhaps application specific services, can be added by sub-classing OFile.

Multiple OFile objects may be opened, to manage multiple files. An object may be attached to only one OFile at a time. OFile maintains a list of currently existing **OFile's**. Functions are provided to iterate over this list.

Each OFile can designate a single object as its root object. The designated object is no different from any other persistent object. It is not necessary to designate a root in order to access objects.

OFile maintains collections for each class. These can be iterated over in order to access objects. This often saves the application having to maintain collections.

OFile has a **fastFind** option, which causes finding by identity, on a deep inheritance hierarchy to work much faster. The biggest effect of **fastFind** can be on loading time of a file, particularly when there are many object references to be resolved. After loading it can be switched off to release the memory it uses.

OFile has full responsibility for managing file space.

ObjectFile recognizes that certain aspects of file management are an application dependent activity. For this reason it implements one of many possible strategies not in OFile itself, but in a subclass of OFile. Other strategies can be implemented by similarly sub-classing OFile. The file management strategy provided by ObjectFile is implemented in **OUFile**.

2.7 Object Existence

The normal **C++** object exists in memory, or not at all. The ObjectFile object can also exist in file. The possible states for an ObjectFile object is:

- in memory only
- in memory and in file
- in file only

As for any **C++** object, the constructors and destructor are always used when creating and deleting an object in memory.

Once an object is attached to a file, ObjectFile is responsible for managing its existence in memory. This means that you should never explicitly delete an object that is attached to a file. When you close the OFile, ObjectFile will delete all its persistent objects from memory. When **OFile::purge()** is called, it may also delete objects that are marked as purgeable.

Detaching an object from the file does not delete it from memory. The application program can now do so, if it wishes.

People who are new to developing with persistent objects, often find it difficult at first to distinguish between the two lives of an object. Remember the constructor/destructor combination take it in and out of memory. The read-constructor and oWrite method take it in and out of the file.

2.8 Object Relationships

C++ does not define the nature of relationships between objects. It has a very general concept to describe any type of relationship, the pointer. The main advantage of the pointer is efficiency. There is no faster or memory-efficient method of maintaining and handling a relationship that is in memory.

In a real objected oriented application, there is usually a more precise meaning to a particular relationship. Ownership is an important example. If an object owns another object and maintains the relationship by use of a pointer, it must delete that object in its destructor. On the other hand if it just points to the object and does not own it, it must not delete it.

Persistent objects must also be able to maintain relationships. At the moment an object that owns another object is attached to the file, then the owned object must also get attached somehow.

The problem here is that only the programmer knows the nature of the relationship. The behavior must be coded somewhere.

OPersist has two virtual methods **OPersist::oAttach** and **OPersist::oDetach.** These are called when an object is attached or detached from a file. If an object maintains a persistent relationship, then the methods may be overridden. If necessary they call the same methods on the related objects. The **deep** parameter can be used to determine whether to pass on the attach or detach to the related objects. If **deep** is **false** then it should not be passed on.

Relationships between objects that may not be in memory at the same time can be maintained, among other ways, by using the **ODemand** object as a *smart pointer*.

On-demand Relationships

A persistent object in an open OFile does not have to be in memory. It only needs to be read, when it is first used. This is very useful when the number and size of objects are likely to exceed the amount of available virtual memory.

If regular C++ pointers connected all objects, there would be no way of doing this, since a memory pointer can only point to a memory address. ObjectFile has a special pointer object, **OnDemand**, which is capable of either holding a memory address or holding a persistent object identity. If the OnDemand has once resolved the object, it simply returns a pointer to it. Otherwise it goes to the **OFile** to resolve it. This may involve reading it from the disk file. The application program never has to explicitly use the object memory pointer, because the -> operator of OnDemand is overridden to return it. This is sometimes referred to as pointer swizzling.

One-to-Many Relationships

The one-to-many relationship is a fundamental concept in the modeling of object systems. It is usually represented as some form of collection.

The ObjectFile methodology allows the developer to use the class library of his choosing in his persistent classes. Collections are usually a major feature of any

particular class library. ObjectFile purposely does not force the developer to use a particular collection, to represent one-to-many relationships.

Many developers however want a ready-made solution. ObjectFile supplies this in the **OSet** and **OnDemandSet** classes. These are streamable persistent classes, derived from the Standard Template Library **set**. Their interface is identical to the STL **set** with additions for persistence.

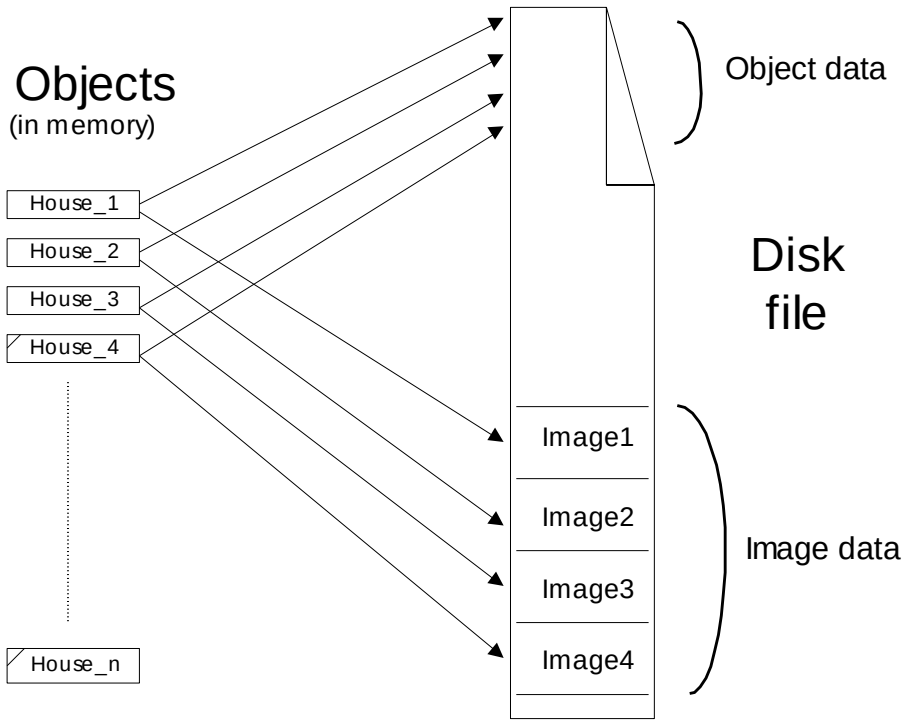
Developers that want to use their own collections can regard them as a design pattern. Any non-persistent collection can be sub-classed to produce a streamable persistent collection.

2.9 BLOB's

When large amounts of binary data must be stored, it is best stored in an **OBlob**. An **OBlob** only brings the data into memory when it is needed. It can then be purged from memory independently of its container. In this way, a large application file containing bitmaps or sound data, for example, can be run in very limited memory.

As an example, consider a database that contains data on houses for sale. The object model is very simple. There is a class to represent a house. It has attributes to represent the various attributes of the house - address, price, number of rooms etc. This textual and numerical data is quite concise and so uses very little memory. However a picture of the house is to be included. Even with compression this may be 100 Kb per house. A database containing 1000 houses will be 100 Mb in size. With today's disk storage this is no problem, however it is clear that manipulating that amount of data, even in virtual memory will not be easy. One possible solution would be to store the images in separate files, and only hold references to these files in the database. But this is a complicated and messy solution, since it means having to manipulate files and filenames.

Using **OBlob** the solution is easy. The image of the house will be stored in an **OBlob** attribute of the house class. The house object itself can be in memory all the time. Only when the image is required, for example in order to display it, does it need to be read into memory. It can then be immediately purged. In this way a maximum of one image will be in memory at any time.



2.10 The Object Cache

Most modern operating systems provide a sophisticated virtual memory management system. This ensures that application data can grow even beyond the limits of physical RAM. Some applications however require data storage that stretch or go beyond those limits. ObjectFile is designed from the ground up to handle this type of application.

An object of a class derived from **OPersist** that is attached to an **Ofile**, does not have to reside in physical memory. It can be marked as purgeable. The object cache ensures that when a pre-defined number of objects are in memory (the object threshold), then all or some of the objects marked purgeable, will be cleared out of memory.

The strategy for purging objects is under the control of the developer. This can ensure minimal access time for all the varying circumstances

Most application programs will not need to use the object cache. They do not deal with amounts of data that reach anyway near the amount of virtual memory available.

2.11 Multi-threaded applications

Many applications are single-threaded. It is much simpler to program in a single-threaded environment, because the programmer does not need to take account of various types of conflicts that can occur.

ObjectFile can also work in a multi-threaded environment. This is particularly useful for web-server applications. For the sake of efficiency these are often deployed as a shared library or DLL. In this case the web server may make multiple calls to the application simultaneously. It does this by creating a thread for each call.

ObjectFile is very fast. If used correctly in a shared library it can be used to write very fast web applications. This can mean the difference between a server running smoothly, or coming to a complete standstill under heavy load.

3. Using ObjectFile

3.1 Template for a Persistent Object

The following shows a template for a class derived from OPersist. Commented code can be removed or un-commenented, as needed.

templat.h

```
#ifndef TEMPLAT_H
#define TEMPLAT_H

#include "opersist.h"
//#include "oiter.h"


const OClassId_t cTemplateClass = 10;

class TemplateClass : public OPersist
{
typedef OPersist inherited;
public:
    // Read constructor
    TemplateClass(OIStream *c):OPersist(c){}
    ~TemplateClass(void){}
    // Other constructors
    TemplateClass(void){}

    // Remove comments if you need to override.
//  void oAttach(OFile *,bool);
//  void oDetach(OFile *,bool);

    OMeta *meta(void)const{return &_metaClass;}
```

```
    // Define iterator for this class(if you need one)
//  typedef OIteratorT<TemplateClass,cTemplateClass> It;

protected:
    void oWrite(OOStream *)const;
private:
    // ObjectFile Instantiation function
    static OPersist *New(OIStream *s){return new TemplateClass(s);}

    static OMeta _metaClass;

    // Persistent data
};

#endif
```

templat.cpp

```
#include "odefs.h"
#include "templat.h"

// The constructor of the meta class associates the class identity with
// the class.
OMeta TemplateClass::_metaClass(cTemplateClass,
                (Func)TemplateClass::New,cOPersist,0);


void TemplateClass::oWrite(OOStream *out)const
// Write persistent data to the stream. This should always be overidden
// when there is persistent data. The first method called should be the
inherited
// oWrite().
{
    inherited::oWrite(out);

}
```

3.2 Creating an OUFile

OUFile *file = new OUFile(0,OFILE_CREATE); // Create an unnamed file

This creates an unnamed file. You access it through its pointer **file**. In actual fact OUFile creates a temporary file, so you can perform incremental commits from time to time if you want. If a name is specified as the first parameter, a named file will be created.

3.3 Opening an OUFile for update.

```
OUFile *file = new OUFile("person1.ofl",OFILE_OPEN_FOR_WRITING);
```

This opens an existing file named 'person.ofl'. It creates a temporary copy of the file so that incremental commits will not alter it.

3.4 Opening an OUFile for reading.

```
OUFile *file = new OUFile("person1.ofl",OFILE_OPEN_READ_ONLY);
```

This opens an existing file names person.ofl. It opens the file directly and does not create a temporary copy. It is therefore much quicker then when opening for writing. You may do anything with it, except to commit it or save it.

3.5 Attaching Objects

The **OFile::attach** method is used to attach an object derived from OPersist to the file. The object is then said to be 'in file'.

```
// create an object
Person *husband = new Person("John","Doe","13 Farmers Lane","South
Bronx","New York","USA",23456,70,1.80,33,0,

// attach to file
file->attach(husband);
```

3.6 Detaching Objects

The **OFile::detach** method is used to detach an object from the file.

```
file->detach(husband);
```

3.7 Accessing Objects

Single Objects

A pointer to an object can be obtained through its identity. This is useful because it means an object relationship can be maintained, even though the object is it not yet in memory.

```
OPersist *ob = file->getObject(10);  // get object with id = 10
```

All Occurrences - Iteration

An OFile can be regarded as a container of objects. More precisely it can be regarded as a set of sets of objects organized according to their inheritance hierarchy. By using the **OIteratorT** template of ObjectFile, the sets can be iterated through at any level of the class hierarchy.

OIterator takes the form of a template in order to provide type-safe access to any particular class hierarchy. To declare an OIterator for a class MyClass, the following declaration would be made in the class declaration:

```
typedef OIteratorT<MyClass,cMyClass> It;
```

The iterator would be used as follows:

```
MyClass *mc;
MyClass::It mit(file);     // declare iterator instance for file
while(mc = mit++)          // Iterate over all instances of
           // MyClass in file
    mc->print();       // call print() for every instance
```

Pre and postfix forward iterators are provided. When the iterator returns **0**, the end of the set has been reached.

A second parameter in the iterator constructor, **deep**, allows you to specify whether or not sub-classes should be traversed. This is a default parameter so it does not have to be specified.

If you do not require type-safe access, you can use the predefined type **OIterator**.

In order to maintain compatibility with future versions of ObjectFile, no assumption about ordering of the objects within a set should be made.

A reverse iterator could have been provided since ObjectFile's current back-end, supports reverse iteration. It was not provided in order to retain the possibility of using a back-end that does not support it. However reverse iterators are rarely needed.

3.8 Committing the file.

This is the act of bringing the disk file up to date with the state of the objects in memory. Every application must decide when to do this. It can be done every time a single object has been changed, or in response to a user command. Commit in ObjectFile is incremental. That is to say that only dirty objects are actually written to the disk file. This means that there is very little overhead involved in committing an almost unchanged file.

The disk file is always in a valid state between commits. If for some reason your application crashes during a commit, the file may not be in a valid state. For this reason, if data integrity it vital, it is advisable to sub-class OFile and implement a 'workfile' and/or a 'backup' file management strategy. OUFile implements both of these strategies, so if a crash occurs during a commit, only the 'workfile' is corrupted.

Various operating system events can occur during a file commit. The 'disk full' situation is one such event. Handling of these situations is application dependent. ObjectFile throws an exception in these cases. It is therefore advisable to protect the **commit()** function by a try/catch statement. If the commit fails by throwing an exception, the file may be in an inconsistent state. Only a subsequent successful commit will return it to a consistent state.

```
// Write the file. This could fail if the disk is full.
try{
    // Commit the file
    file->commit();
    return cSUCCESS;
}catch(OFileErr x){
    // Handle the failure by printing a message.
    cout << x.why() << '\n';
    return cERR_FAIL;
}
```

3.9 Accessing the OFile list

ObjectFile can maintain multiple instances of OFile. These are maintained automatically in a null terminated linked list. The list can be accessed in the following way:

```
// Get the first file
OFile *f = OFile::getFirstOFile();
// Iterate over files
while(f)
{
    f-> <do something>;
    f = f->getNextOFile();
}
```

3.10 Determing an objects OFile

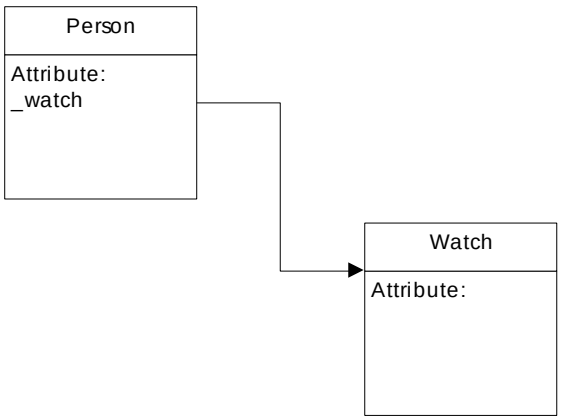
To determine which file holds the object pointed to by ob:

```
OFile *f = OFile::oFileOf(ob);
```

3.11 Relationships

The **oAttach** and **oDetach** virtual functions of OPersist are used to handle relationships between objects.

Example 1



The Person object may or may not own a Watch object (i.e. a conditional relationship). The Watch, if it exists, is attached and detached from the file along with the Person. Note that the inherited method must also be called. In **oAttach** it is called first; in **oDetach**, last. This may or may not be important in your class, but is good practice. The inherited **oAttach** method handles the current object.

```
void Person::oAttach(OFile *file,bool deep)
// Attach this object to the file. That is, make the object persistent.
{
    inherited::oAttach(file,deep);

    if(deep)
        if(_watch)_watch->oAttach(file,true);
}

void Person::oDetach(OFile *file,bool deep)
// Erase this object from the file. That is, make the object non-
persistent.
{
    if(deep)
        if(_watch)_watch->oDetach(file,true);

    inherited::oDetach(file,deep);
}
```

Example 2

In this example a streamable collection (derived from a STL collection) object passes on oAttach and oDetach to its object list.
Note that there are no inherited methods to call, and the functions are not virtual. This is because MySCollect is not derived from OPersist (being streamable).

```
void MySCollect::oAttach(OFile *file,bool deep)
{
    if(deep){
        // for every element in the list
        for(MySCollect::iterator it = begin();it != end(); ++it)
            // Attach object to the file
            (*it)->oAttach(file,deep);
    }
}

void MySCollect::oDetach(OFile *file,bool deep)
{
    if(deep){
        // for every element in the list
        for(MySCollect::iterator it = begin();it != end(); ++it)
            // Erase from file but not from this collection
            (*it)->oDetach(file,deep);
    }
}
```

3.12 Persistent/Transient Objects

The ObjectFile methodology allows objects to be persistent at one point in time, and transient at another point. This is very useful in real applications. For example when implementing 'Cut' behavior in an editor, the object being 'Cut', must be made transient, and if 'Pasted', persistent again.

This presents a small problem. Remember that if an object is attached to a file you must not delete it explicitly. When you have objects related by ownership, usually the destructor of the owner, deletes the owned object, by doing a **delete** on its pointer. So we have a situation where if the owner is transient, it must delete its owned object, and if it is attached, it must not.

The solution is as follows. **OPersist** objects have a method **oAttached**. This returns **true** if the object is attached to a file, and **false** if not. Only delete the owned object if **oAttached** returns **false** for the owner.

```
Person::~Person(void)
// _watch is a pointer to an object owned by Person.
{
    // If the object is not in the file, destroy owned persistent
objects.
    if(!oAttached())
        delete _watch;
}
```

Be careful to check the current object and not the owned object, because if persistent, the owned object may have already been deleted by ObjectFile.

In fact, for this reason, you should never access a pointer to a persistent object in the destructor of another persistent object. This is rarely necessary.

3.13 Writing Objects

C++ allows the developer to define data types of any degree of complexity. It would be extremely inconvenient for a persistent framework to force on the developer data types of its own. ObjectFile gives the application developer complete freedom to use any data types he wishes. The disadvantage of this is that the writer of a persistent object must specify how the data-type will be written to the file. ObjectFile makes

this process as simple as possible, by providing an output stream to which data must be sent.
The virtual function **oPersist::oWrite** must be overridden in any persistent object that wants to add data attributes of its own. First the inherited method is called, ensuring that the super classes data is written. Then the objects persistent attributes are sent, one by one, to the stream. Functions are provided for each of the basic data types. Application defined types that are streamable should also implement **oWrite** methods that can be called.

Example

```
void Person::oWrite(OOStream *out)const
// Write persistent data to the stream.
{
    inherited::oWrite(out);

    _watch.oWrite(out);        // _watch is a streamable object
    out->writeCString256(_firstName.c_str());
    out->writeCString256(_familyName.c_str());
    out->writeLong(_zipCode);
    out->writeFloat(_weight);
    out->writeShort(_age);
}
```

Writing Relationships

Relationships are defined in **C++** by a memory address. They would lose their meaning if written directly to the storage media. However ObjectFile persistent objects have a persistent identity, which can be used instead. OOStream provides a method writeObject(OPersist *) which can be used to extract and write that identity.

Example

This writes an array m[4] of pointers to persistent objects.

```
void MyClass::oWrite(OOStream *out)const
{
    inherited::oWrite(out);

    for(int i = 0;i < 4;i++)
        out->writeObject(m[i]);
}
```

If the identity of an object is the only thing in memory then:

```
out->writeObjectId(OId id);
```

can be used instead.

3.14 Reading Objects

C++ provides a sophisticated mechanism for object construction - constructors! One of the methodological advantages of constructors is that if used correctly, they ensure that simply by instantiating an object, it is created in its invariant state. In order to respect this concept, ObjectFile uses an object constructor to read the data of the object. The constructor ordering rules of **C++** ensure that data is always read in the same order, without the developer having to call the inherited methods explicitly. The developer must however respect this ordering in the oWrite method that he writes.

OPersist provides a special read constructor. Every derived class that reads data must provide a similar constructor. This constructor must always call the super classes read-constructor(s).

An input stream OIStream is provided to extract the various data types. Streamable classes should also provide a read constructor.

Example

```
Person::Person(OIStream *in):OPersist(in),_watch(in) // Call super
class and contained                                  // class read-
constructors.
// Read constructor
{
    _firstName = in->readCString256();
    _familyName = in->readCString256();
```

```
    _zipCode = in->readLong();
    _weight = in->readFloat();
    _height = in->readFloat();
    _age = in->readShort();
}
```

Reading Relationships

Relationships are defined in **C++** by a memory address. Objects existing on storage media do not yet have memory addresses. ObjectFile persistent objects have a persistent identity, which can be used instead. OIStream provides a method **OPersist *readObject()** which can be used to resolve that identity.

Example

This reads an array m[4] of pointers to persistent objects.

```
MyClass::MyClass(OIStream *in): OPersist(in)
// Object file read constructor
{
    in->readCString(dum,sizeof(dum));
    for(int i = 0; i < 4;i++)
        m[i] = (MyClass *)in->readObject();
}
```

If the identity of an object is the only thing required:

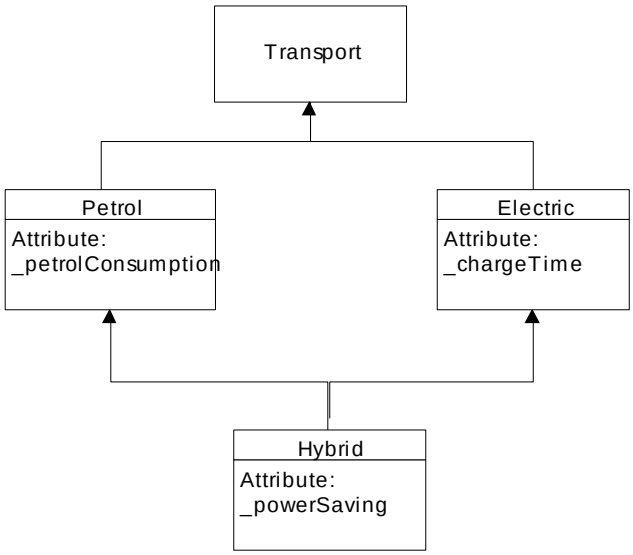
```
    OId id = in->readObjectId();
```

can be used instead.

3.15 Multiple Inheritance

Multiple inheritance is handled in ObjectFile simply by extending the methodology used for single inheritance. When a class has multiple base classes, the read-constructor and **oWrite** method, just calls the base classes methods one after the other. It is important to respect the **C++** ordering of constructor calls of the read-constructor, in the oWrite method.

To illustrate, take the following example. Two types of Transport are derived, Petrol and Electric. Hybrid inherits from both of them.



The oWrite method would look like this:

```
void Hybrid::oWrite(OOStream *out)const
{
    // Write both superclasses
    Petrol::oWrite(out);
    Electric::oWrite(out);
```

```
    out->writeFloat(_powerSaving);
}
```

3.16 Virtual Base Class

Virtual base classes ensure that an object only has a single copy of the attributes of the classes that it is derived from. This requires a little extra work in the **oWrite** methods. They must check whether to call the **oWrite** method of the virtual base. This is done with the **OOStream::VBWrite()** method.
The first call of VBWrite() for an object returns true. All subsequent calls for that object will return **false**. As long as all **oWrite** methods that inherit from a virtual base class use this convention, only one copy of the base classes attributes will be written.

```
void Petrol::oWrite(OOStream *out)const
{
    // Check whether to write the virtual base
    if(out->VBWrite())
        inherited::oWrite(out);

    out->writeFloat(_petrolConsumption);
}

void Electric::oWrite(OOStream *out)const
{
    // Check whether to write the virtual base
    if(out->VBWrite())
        inherited::oWrite(out);

    out->writeFloat(_chargeTime);
}
```

3.17 Object Evolution

Throughout the lifetime of an application, it is inevitable that the structure and types of classes will evolve. The application user, does not usually want to be aware of this. ObjectFile provides a methodology for object evolution that does not require user intervention. The following schema changes can be made:

1. Changes to the contents of a class
    1.1 Changes to an attribute
        1.1.1 Add a new attribute to a class
        1.1.2 Drop an existing attribute from a class
        1.1.3 Change the name of an attribute of a class
    2.1 Changes to methods
        2.1.1 All changes to methods
2. Changes to the superclass/subclass relationship
    2.1 Adding a class
    2.2 Removing a class

UserSourceVersion

Every file has an attribute called the userSourceVersion. This is the application programs source code version. It should be assigned before opening a file as follows:

OFile::setUserSourceVersion(3);

By default it is 1. It is stored in the file, when it is committed.
The userVersion of a file is the userSourceVersion with which the file was committed. This can be obtained as follows:

long ver = file->userVersion();

though it is most often used in the read constructor, in which case it should be obtain as follows:

long ver = in->userVersion();

With these two values you always know the version of your file, and the version of your current application. They can be used to build automatic schema changes into your application. The governing principle of conversion is that when an object is read, it is immediately converted to the current source code version. For this reason most of the work is done in the read-constructor of the classes that need conversion The method **OUFile::convert** can be redefined or overridden to provide application specific conversions.

Add a new attribute to a class

Below is the read constructor of a class having two attributes. A string **_name** and a float **_weight**. This is version 1 of our file.

```
EvPerson::EvPerson(OIStream *in): OPersist(in)
{
    _name = in->readCString256();
    _weight = in->readFloat();
}
```

We now want to add another attribute **_height**. We re-write the read constructor as follows:

```
EvPerson::EvPerson(OIStream *in): OPersist(in)
{
    _name = in->readCString256();
    _weight = in->readFloat();

    if( in->userVersion() > 1)
        _height = in->readFloat();      // Read height only if file
verion                                     // > 1
    else{
        _height = 0.0;                  // Otherwise set a default
        oSetDirty();                    // Ensure the object gets written
    }
}
```

When reading a version 1 file, the new attribute _height will be set to 0.0, and the object set dirty. When reading a version 2 file, the _height will be read from the stream.

Drop an existing attribute from a class

We then decide that we only really want to store the name, and not the height or weight. We re-write the read constructor as follows:

```
EvPerson::EvPerson(OIStream *in): OPersist(in)
{
    _name = in->readCString256();

    if( in->userVersion() < 3){
        in->readFloat();   // weight

        if( in->userVersion() > 1)
            in->readFloat();      // Read height only if file verion > 1
        oSetDirty();              // Ensure the object gets written
    }
}
```

It is important to always be able to read **all** previous versions of the file and not just one previous version. For version 3 files we only read the _name. For the other versions we read the attributes, but do not assign the read values to anything.

Change the name of an attribute of a class

No changes required as long as the type of the attribute does not change.

All changes to methods

Since only object data is stored in the file, changes in the **C++** code, to class methods will not affect the data stored in a file.

Adding a class

Just add the new class.

Removing a class

All objects of the given class must be removed first. This is best done in the redefined or overridden **OUFile::convert** method.
If you want to be able to read files that used this object, you will not be able to remove the class from the application. It can however be cut down to just the methods required by ObjectFile.

3.18 Storing an ObjectFile in an OLE Compound Document

An ObjectFile file can be stored in an OLE Compound Document almost as easily as in a regular file. A special constructor is provided for opening an OFile on an **IStorage**. The file management class, OUFile, cannot be used for OLE storage.
To compile ObjectFile with OLE support you must make the following #define for compilation in **odefs.h**.

```
#define OF_OLE
```

The following example is taken from olemain.cpp test program:

```
   OFile *f = new OFile(istorage,mbName,0,
STGM_DIRECT|STGM_READWRITE|STGM_CREATE|STGM_SHARE_EXCLUSIVE);
```

3.19 Exporting Object Data in XML Format

Exporting object data to another format from ObjectFile requires very little work. By writing a sub-class of **OOStream** you can re-use the **oWrite()** methods that already exist for all your classes. See **OOStreamXML** in the reference section for an example of how to do this.

3.20 Using the Object Cache

The object cache is a mechanism for automatically managing the amount of memory used by an ObjectFile application program. It works in terms of objects and not bytes.

This facility will not be needed by most applications, because they do not deal with amounts of data that reach anyway near the amount of virtual memory available.

There are two main programming considerations to bear in mind in order for the object cache to be effective:

1. Objects must be marked purgeable before they can be removed from memory. This is done by calling the **oSetPurgeable()** method on the object. It is important that you do not hold any memory pointers to the object at this stage. See **OnDemand** for one way of maintaining relationships without holding memory pointers.

2. If dirty objects are to be removed from memory, the file to which they are attached must be marked **autoCommit**. This is done by calling **setAutoCommit()** on the OFile. This allows ObjectFile to write dirty objects to the file at its discretion. By using OUFile as your file management strategy, **commit()** will write objects to a workfile. You would then perform a **save(),** to save to your application file.

To start using the object cache an object threshold must be set. The object threshold is defined as the number of objects derived from **OPersist** that can be in physical

memory at one time. This includes all objects in all open files, whether attached to a file or not.

When this threshold is reached a method known as the **OFile::new_handler** will be called. This purges any purgeable objects from memory. The new_handler can be redefined at any time. The mechanism is very similar to the mechanism available in C++, to handle the memory exhausted situation.

If many objects have not changed since they were last committed, that is they are not dirty, then there will be no problem purging them. However if many objects are dirty then they cannot be purged, since the information contained in them would be lost. To avoid this situation call **setAutoCommit()** on the file to which the objects are attached. This allows the new_handler to write dirty objects to the file and then remove them from memory.

The following extract from the **bm_db** test program illustrates the use of the object cache:

```
// Set the object threshold to 10,000 objects.
OFile::setObjectThreshold(10000);

// Create a file
OFile f("ofile.tst",OFILE_CREATE);

// Set the default handler that is good for sequentially
// writing a large file.
OFile::set_new_handler(OFile::new_handler);

// Automatically save if needed
f.setAutoCommit();

// Attach
for(long i = 1; i <= nObjects ; i++)
{
    MyClass10 *p = new MyClass10(avSize);
    f.attach(p);
    p->oSetPurgeable();
    cout << i << " objects written.\r";
}

// Commit objects that were not automatically committed.
f.commit();
```

If there are classes of objects that you do not want to include in the object cache, then you can define a new and delete method for them, that does not count them as part of the cache.

The exact memory configuration and load of the applications target platform may not be known before deployment. For this reason, it might be a good idea to allow the object threshold to be a user configurable parameter.

Error Handling

It may happen, that when a new objects is created, the object cache cannot purge itself to make room. This is analogous to C++ 'out of memory' situation. ObjectFile handles it in a similar way. The exception **OFileThresholdErr** is thrown.

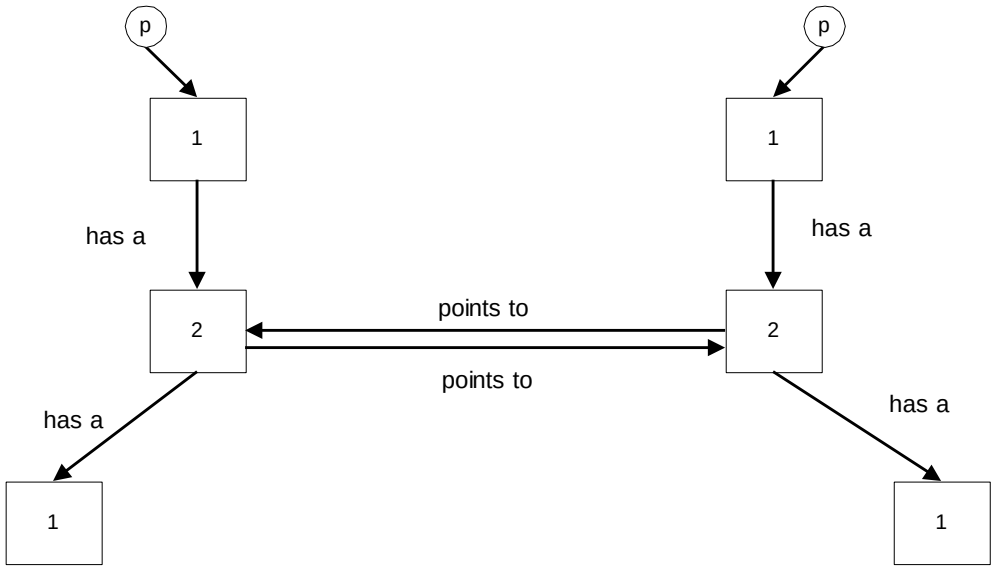
Managing 'out-of-memory objects'

Managing objects so that they can be either in-memory or out-of-memory, can be a non-trivial task. The programmer must add extra code to tell the file manager that an object is no longer being pointed to by a memory pointer. This is often referred to as *reference counting*. It can be very difficult to maintain reference counts correctly. Even more so, when using exceptions. ObjectFile takes the point of view that if you do not need this facility, you should not need to add any code. In most applications, all but a few, if any, objects need this kind of management. So the simplest level of memory management is the do nothing model.

Having said that, if you do need to manage the objects, there are two levels at which you can do it. The first of these is the non-reference counting model. This simply says, that if you once set an object purgeable, it can be thrown out of memory. This is fairly simple. When you have finished with the object just call **oSetPurgeable** on it. In some applications, especially those with complicated relationships or multi-threading, this model can be too simplistic.

The highest level is the reference counting model. In this case a reference count is maintained for each object. You would use this model in cases where you would need a reference count anyway to manage the object, or when writing a thread-safe application. To enable reference counting you must compile with the flag **OF_REF_COUNT** defined in **odefs.h**.

The following diagram shows a number of related objects. The numbers represent the total number of references to the object i.e the reference count.



None of the objects can be removed from memory, because none of them have a reference count of zero. All have at least one memory pointer to them. As can be seen by the labels on the links, there is a notion of ownership between some objects, and a weaker relationship between others. In the reference counting model, setting an object purgeable reduces its reference count by 1. Only when it reaches zero, is it really made purgeable.

It would be nice if an object would automatically manage the purgeabilty of its related objects. To do this we must override the oSetPurgeable function. The overridden function sends oSetPurgeable to the related objects. In the case of a 'has a' relationship we only want to pass it on if the deep parameter is 'true'. In the case of a weaker relationship we do not want to set the related object purgeable at all, so we simply remove the reference that we own, by calling removeRef(). So for the classes of objects with reference count 2, in the above diagram, the method will look like this:

```
void oSetPurgeable(bool deep,OFile *file)
{
    pointsTo->removeRef();              // Remove reference
    if(deep)
        hasA->oSetPurgeable(deep,file);  // Pass on to owned object
    else
        hasA->removeRef();              // Remove reference only
    inherited::oSetPurgeable(deep,file); //  This object.
}
```

Of course, in C++ it is entirely up to the developer to define the nature of a relationship., so there is some flexibility in what can be done here. The important thing is to be consistent. Only remove references where they exist.

A reference exists in the following cases:

1. An object pointer was obtained in the read-constructor.
2. An object pointer was obtained by **OFile::getObject(), OFile::getRoot()** or **OIterator**.
3. **addRef()** was called for the object.

By removing too few references, objects will not be removed from memory. By removing too many, an object may be inadvertently removed, resulting in a crash.

If using OnDemands to represent the relationships, the syntax of the above function would be:

```
void oSetPurgeable(bool deep,OFile *file)
{
    pointsTo.oSetPurgeable(false,file);  // Purge relationship only
    hasA.oSetPurgeable(deep,file);       // Pass on to owned object
    inherited::oSetPurgeable(deep,file); //  This object.
}
```

3.21 Memory Management

In a modern operating system, the amount of physical and virtual memory available at any moment, may vary enormously. Under conditions of high load the amount of virtual memory available may become low. A strong program must take account of the situation when it runs out. In **C++** this is done by writing a 'new_handler'. This is installed by the standard function **set_new_handler**. Many applications simply decide to terminate gracefully, in the case the new_handler is called. The default action is to terminate. However other programs try to continue. Freeing unnecessary memory and returning from the **new_handler** does this.

What can be freed, and how it is freed, is of course application specific. An application using ObjectFile can often free memory by purging its OFile or OFile's. This is done by calling **OFile::purge().** This will only achieve results if objects have been made purgeable. An object is made purgeable by calling its **oSetPurgeable()** method.

3.22 Error Handling

ObjectFile uses two types of error handling. The first is the use of assertions. These are used to detect misuse of ObjectFile methods. They indicate a programming error, and should not occur in a debugged application. The macro **oFAssert** can be redefined (in **odefs.h**) to suit any existing development environment.

The second type of error that might occur is a C++ exception. These can occur with normal use of an application. They usually are related to a system event. Trying to open a read-only file in read/write mode is one example. Writing when the disk is full is another. A good application should catch these exceptions and handle them appropriately. The reference section denotes which methods throw exceptions.

3.23 Programming techniques with ObjectFile

Do not use pointers wherever possible

ObjectFile was written with good C++ technique in mind. Using **new** to create an object, gives you the possibility to create two types of bugs. One is to forget to delete the object (quite easy when using exceptions) The other is to delete it twice. For indexed objects derived from OPersist there is no alternative, but for streamable contained objects, there is no problem just declaring them as members of the

containing object. As long as you write a read-constructor, they can be instantiated from the read-constructor of the container.

Write portable programs

Although you may not envisage porting your application to another hardware platform, anyone who has been in the software industry long enough, will tell that such a requirement is often just around the corner.
ObjectFile is designed to be cross-platform. However if the application program using it does not keep to a few basic rules, you will have difficulty porting both your application and its data files.
There will be no attempt here to write a guide to writing portable programs. There is plenty of literature elsewhere. However bear in mind the following.
Always use the appropriate stream method to read and write you data. In this way the appropriate conversion can be performed when reading a foreign file. Do not for example, try to write longs in a character string.

inherited

C++ lacks a standard way of referring to the super class. This is often needed when dealing with hierarchies of objects. There are several ways to overcome this. ObjectFile uses the most elegant, so it may be worthwhile to adopt it in all your classes. On the first line after the class declaration write a typedef statement:

```
class b : public a {
typedef a inherited;
public:
```

This effectively defines a keyword **inherited,** which is private to the class. Now you can write:

```
inherited::oWrite();
```

in order to call the super class method.

3.24 Debugging Techniques

Printing Objects

The ObjectFile application file takes the form of a binary file. The advantage of this is compactness and fast access. During the development stage it is often useful to see what it contains. Although it could be read by a good hexadecimal editor, and some imagination, it would be better to be able to produce a textual representation of the data contained in the objects.
The **OOStreamXML** class can be used to output an XML represetation of the persistent data. This can be read by an ordinary text editor or an XML browser. It utilizes the **oWrite()** methods of persistent objects, so you do not have to add any code to your classes. The following example demonstrates its use:

```
// Open a file stream
fstream out("person1.xml",ios::out);

// Open a print stream on the file stream
OOStreamXML print(file,out);
OIterator it(file);

OPersist *p;
while(p = it++)
{
    print.start(p);
    p->oWrite(&print);
    print.finish();
}
```

3.25 Multi-threaded applications

ObjectFile can be built for multi-threaded application as follows:

i. Define the flag **OFILE_MULTI_THREAD** on the compile line. Put it in **odefs.h**, if you will be using it permanently.

ii. Make sure any other libraries you are using are multi-threaded enabled.

iii. Link against your compiler vendor's multi-threaded libraries.

There is an important difference in the way objects are made purgeable in the ObjectFile multi-threaded model. A reference count is maintained in each OPersist object.

The object is born with a reference count of 1. Every time **oSetPurgeable()** is called the reference count is reduced. Only when it reaches zero, is it actually set purgeable. In the single threaded model it is always set purgeable, when oSetPurgeable() is called.

Any time it is obtained from the OFile (with **getObject()** or by iterating) the reference count is incremented. This means you should always match calls to oSetPurgeable to calls of getObject(). There is an assert in OPersist to prevent you making too many calls to oSetPurgeable().

The reason for all this is as follows. Two threads are using an object. One finishes with it, and sets it purgeable. We now purge. We will be purging the object while the other thread is using it. With the reference counting scheme that cannot happen.

Just as with the single threaded model you do not need to worry about memory leaks, since closing the OFile will delete all objects, whatever their reference count.

ObjectFile cannot protect against every possible multi-threaded scenario. For example if one thread closes the OFile, while the other is using it, there will be a crash. However with some common sense there should be few problems.

3.26 Hints for writing Web-server applications.

To make the application data available on the Internet, you need to add some code that formats the data into HTML format. This could be done by building your application as a CGI program.

The problem with CGI is that it requires the operating system to invoke a process each time a client requests data. In addition, since the program runs from start to finish, all files it uses must be opened and closed each time. Though this works, it leads to very poor throughput of the web-server.

There is a second method that is far more efficient. It uses Microsoft's **ISAPI** or Netscape's **NSAPI** interfaces. The web application is built as a **UNIX** shared library or a **Windows DLL**. The web server software then simply makes calls to this, instead of invoking a process. Since multiple clients can request data at the same time, the server software is multi-threaded. This means that any server application must be thread-safe.

You can make any application thread-safe by blocking at the entry points. This will however be very inefficient. In a good thread-safe application, you should be able to do several tasks simultaneously. One prime example in a web-server application, is that one thread should be able to transmit data, while another is preparing data for transmission. This is because the transmission is often blocked by transmission delays over slow lines.

ObjectFile is thread-safe, so it can easily be used in a web-server application. One client's thread can be looking up data; another preparing an HTML page and another 5 transmitting the page. All of them working from the same open file.

There are several alternatives for building your application as a web-server application. Which you choose, will depend on your applications requirements and its design:

1. Build the whole application as a shared library or DLL. Then just add server entry points and HTML formatting code.
2. If you use the model/view architecture, place the model into a separate shared library. Then add server entry points and HTML formatting code. This will cut down the size of the application enormously over method 1.
3. Compile and link only the parts of the application required for the server-application into a completely separate application.

It is best to choose the method most likely to cut down on program maintenance.

Keeping the File Open.

The most significant thing you can do in order to make the application efficient is to keep the **OFile** open, from one request to the next. **OFile** already maintains a static list of open files. In the case of the application using more than one file, you should iterate through this list until you find the one you need. When you exit the server call, do not close the file.

The Object Cache.

You should consider using the object cache for large files.

Closing the File.

You can be friendly to the operating system by closing all files when it exits. ISAPI defines a terminate method. If you do not have one then you can write something like this:

```
class AutoCloser
{
public:
    AutoCloser(void){}
    ~AutoCloser(void){OFile::closeAll();}
};

static AutoCloser autoclose;
```

4. ObjectFile Class Reference

This reference section documents the classes that can be used directly or sub-classed by developers. It does not document classes or methods that are for internal use only. For the sake of conciseness functions documented by a super-class are not repeated in the sub-class, unless there is a notable difference in functionality.

OBlob, OBlobT, OBlobP (oblob.h,oblobt.h,oblobp.h)

Purpose

OBlob is used to store large blocks of freeform data known as blobs. BLOB actually stands for Binary Large Object. Blobs can be used to store bitmap images, sound clips, text files or any other large set of freeform data. OBlob has several features that make it suitable for this purpose:

- The data is only brought into memory on demand. Even massive blobs will not hog memory or slow application startup. They can be purged after being used and re-read when needed again.
- They are read directly into memory, with no translation, making them very fast.

OBlob is a streamable object. It must have a container derived from **OPersist**, in order for it to be made persistent. The persistent container must call the read/write and attach/detach functions. If you want to make an OBlob into an indexed object you can always multiply inherit from an **OPersist** based object and from **OBlob**.

Typical blob data can often be compressed greatly. The most suitable type of compression is dependent on the type of data. For this reason ObjectFile does not supply any compression. Applications can subclass **OBlob** to do the compression and de-compression.

Portability consideration: If you want your application data files to be portable across various processor formats, do not use OBlob's for typed data such as **long, short** etc.

Methods

OBlob(void)
Default constructor. Creates an OBlob with no data.

OBlob(OIStream *in)
Read from file constructor. The blob data is not actually read. It will be read only when accessed.

OBlob(char* blob,oulong size)
Existing data constructor. **blob** is a pointer to memory containing data. **size** is the size in bytes of the data. Takes over responsibility for managing the data pointed to by blob. OBlob will delete the data from memory when the object is deleted.

OBlob(oulong size)
Empty blob constructor. Builds data of the given size. The data is not zeroed.

OBlob(const OBlob &from)
Copy-constructor. Creates an OBlob identical to **from**.

~ OBlob(void)
Destructor.

void read(OIStream *in)
Read the blob. This should only be used on an empty unattached blob. It is useful for creating a two phase read construct. For example, if a collection of OBlobs are to be read into a list, it can be done as follows:

```
// Part of some read constructor, of an object that holds a list of
OBlobs
for(int j = 0; j < count; j++)
{
    OBlob b;                    // Empty OBlob
    _blobList.push_back(b);     // Copy it to list
```

```
    _blobList.back().read(in);// Read it straight into list
}
```

OBlob& operator =(const OBlob& from)
Assignment operator. Copies the contents of **from**.

char* getBlob(void)const
This gets a pointer to the memory of the OBlobs data. You can then manipulate the data in any way you want. It is your responsibility not to write beyond the end of the data. You can check the size of the data with the size() method.
If you do change the data, you must call oSetDirty for the OBlob, to tell it the data has been changed.

const char* const_getBlob(void)const
This gets a const pointer to the memory of the OBlobs data. This is useful if you know that you will not need to change the data, because the compiler will ensure that you do not make a mistake.

void setBlob(char* blob,oulong size)
Set the data of the OBlob to **blob**. The OBlob takes over responsibility for managing the data pointed to by blob. OBlob will delete the data from memory when the object is deleted.

void copyToBlob(const char* blob,oulong size)
Copy the data pointed to by **blob** of size, **size,** to the OBlob.

oulong size(void)const
Returns the size of the OBlobs data in bytes.

void oAttach(OFile *file)
Attach the OBlob to a file. This should be called by the containing objects overridden OPersist::oAttach method.

void oDetach(OFile *file)
Detach the OBlob from a file. This should be called by the containing objects overridden OPersist::oDetach method.

void oSetDirty(bool dirty = true)
Tell the OBlob that it is dirty and must therefore be written to the file on the next commit.

void oWrite(OOStream *,const char * label = 0)const
Write the OBlob to the stream. This must be called by the containing objects overridden OPersist::oWrite method.

bool purge(void);
Purge the data of the OBlob from memory. Returns true if it succeeded. If it returns false then the OBlob is probably dirty. In which case it cannot be purged.

BlobAllocator ballocator
This is the OBlobs memory allocator. Any memory passed to to OBlob should be allocated by this allocator.

char *allocator.allocate(long n)
Allocate n bytes and return its address.

void allocator.deallocate(char *p)
Deallocate memory allocated at address p.

char * allocator.address(char *x)

Obtain the address of memory allocated at x. This will make more sense after reading the next section.

Other Types of Blobs

By studying the source code in **oblobt.h**, you will notice that ObjectFile blobs are not quite as simple as presented here. OBlob is actually an instance of a template **OBlobT**. The template allows one to design blobs based on various types of handles and allocators.
As an example **OHGBlob** (ohgblob.h) is a blob based on the MS Windows global memory handle HGLOBAL. By using this, for example DIB's can be read directly from disk into the DIB memory structure, without needing any translation.

OBlobP in files **oblobp.cpp** and **oblobp.h** is a non-template version. It is somewhat simpler to compile than the template version. It is not suitable for 16-bit systems. It uses a char * to access data.

OFile (ofile.h)

Purpose

OFile is the container of all persistent objects. It has only those features necessary for file management. This keeps it lean and mean. More sophisticated file management can be implemented in a subclass (see OUFile)

OFile can optionally maintain an object cache. The cache is per executable and not per file. It includes all OPersist objects (and those derived from it), whether attached to a file or not. The cache is configurable at run-time. The method of purging objects is also configurable, using a mechanism that is very similar to that used by C++, to handle the memory-full situation. This allows you to use the optimal strategy according to what you are doing.

OFile maintains a static null terminated list of all OFiles that currently exist.

Methods

OFile::OFile(const char *fname,long operation, const char *magicNumber = 0)

Constructor. **fname** is a pointer to the full path name of the file.
operation defines how to open the file. It can take the following values:
The following three flags are mutually exclusive.

OFILE_CREATE - Create a new file. This will overwrite an existing file of the same name.

If the file cannot be created an exception will be thrown.

OFILE_OPEN_FOR_WRITING - Open an existing file for writing. If the file does not exist an exception will be thrown.

If or'ed with **OFILE_CREATE** and the file exists, the existing file will be opened.

OFILE_OPEN_READ_ONLY - Open an existing file readonly. If the file does not exist an exception will be thrown.

The following flags may be or'ed together with the previous flags.

OFILE_FAST_FIND - An extra index will be built, that will allow objects, of unspecified classes, to be found much more quickly than otherwise. This extra speed is particularly useful when many objects have to be connected in a file that is being loaded. It can improve file loading speed enourmously. The index requires extra memory, but this can be released by calling **fastFindOff()** after the objects have been connected, or at any other time. Use of this option is probably best left to the optimization stage of development.

magicNumber is a 4 byte character string that can optionally be used to identify a file type. If it is specified, ObjectFile will use it to test if the file being read has the same magic number as the parameter. If not, an exception will be thrown. The same magic number can be used by operating systems to distinguish the file from other files. For this purpose the ofset in the file must be known. ObjectFile always stores it magic number at offset 28 bytes in the file.

Exceptions: OFileErr is thrown on failure to open the file.

OFile::OFile(IStorage *istorage,
const char *fname,long operation,
unsigned long istorage_mode,
const char* magicNumber)

Constructor. **istorage** is the OLE IStorage used to store OFile's data. **fname** is a pointer to the name of the stream to be opened or created. *This must be a UNICODE string if using WIN32*.

operation can be **OFILE_FAST_FIND**. **istorage_mode** can be a combination of the following Windows defined values:
STGM_DIRECT 0x00000000L
STGM_TRANSACTED 0x00010000L
STGM_SIMPLE 0x08000000L
STGM_READ 0x00000000L
STGM_WRITE 0x00000001L
STGM_READWRITE 0x00000002L
STGM_SHARE_DENY_NONE 0x00000040L
STGM_SHARE_DENY_READ 0x00000030L
STGM_SHARE_DENY_WRITE 0x00000020L
STGM_SHARE_EXCLUSIVE 0x00000010L
STGM_PRIORITY 0x00040000L
STGM_DELETEONRELEASE 0x04000000L
Please refer to OLE documentation for full details of each parameter.
Exceptions: OFileErr is thrown on failure to open the IStorage.

~OFile(void)
Destructor. Closes the file and deletes all persistent objects from memory. If you have not committed, then all changes will be lost.

void fastFindOff(void)
Switch off fastFind and release any memory used by it.

void commit(const bool compact = false ,bool wipeFreeSpace = false)
Write any dirty objects to the physical file. This brings the physical file up to date with what is in memory.
compact is now obsolete. Free space in the file that was generated by the deletion of objects, must be removed by a user written method. This is not usually necessary for most applications, because new objects fill any 'holes'. However one way of compacting the file, is to detach objects from the **OFile**, and attach them to another **OFile**.
wipeFreeSpace causes a character to be written over the free space in the file. This should improve the compression ratio when the file is compressed by an external utility**.** It should not be used all the time, since it will degrade file write performance significantly.
The commit process can fail. This is because it relies on a physical device. Any robust application should catch exceptions when calling this function.
Exceptions:
OFileErr is thrown on failure to commit the file. One of the following messages may be returned. In both cases the file will be in a consistent state.
"Failed to allocate space on the disk for the file."
"Attempt to increase the file beyond the maximum permitted."
OfileIOErr. If the following message is returned, the file may not be in a consistent state.
"Write failure."
This may be caused by a physical disk problem. After a failed commit the following conditions will apply:
(I) Successfully written objects will no longer be dirty. Unsuccessfully written objects will still be dirty.
(II) This disk file may be in an inconsistent state.
Consistency can be regained by a subsequent successful commit. However you should never try to reopen a file that had a failed commit as its last action.

void attach(OPersist *ob,bool deep = true)
Attach an object to the file.
If **deep** is true then all contained objects are also attached to the file. The virtual function **OPersist::oAttach()** is called for the object. It should interpret **deep**.

void closeAll(void)

Close all OFile's.

void detach(OPersist *ob,bool deep = true)
Detach an object from the file. This will invalidate any iterator pointing to that object.
If **deep** is true then all contained objects are also detached from the file. The virtual function **OPersist::oDetach()** is called for the object. It interpret should interpret **deep**.

long objectCount(OClassId_t id,bool deep = true)
Return the number of objects of the class with class identity **id** in the file. If **deep** is **true,** include all its subclasses.

OPersist *getObject(const OId oId,OClassId_t cId = cOPersist)
Return a pointer to the object with identity **oId**. The **cId** parameter can be the class identity of the object being sought, or a superclass of it. The more precisely it is specified, the faster the function will work. If **fastFind** is on then it will always work very fast.

This function always adds a persistent reference to the object it returns. That is to say, it sets it not purgeable. The reference can be removed by calling the objects **oSetPurgeable()** method.

static OFile *getOFile(int fileId);
Return a pointer to the OFile identified by fileId.

int id(void)
Return a unique identifier for the OFile.

OPersist *restore(OPersist *ob)
Restore an object with its original data from the file. The address of the object is invalidated and a new address returned.
ob is the object to be restored.

long purge(OClassId_t cId = cOPersist,bool deep = true,long toPurge = LONG_MAX)
Purge from memory, objects of the file of type cId that are marked purgeable. If **deep** is **true** then purge sub-classes as well. Stop purging after **toPurge** objects have been purged. Return the number of objects purged.

void setRoot(OPersist *root)
Set the root object of the file. **root** is a pointer to a persistent object, or 0 to clear the root. It is not necessary to designate a root object.

OPersist *getRoot(OClassId_t cid = cOPersist)const
Return a pointer to the root Object. **cid** is a super-class id of the root object. 0 is returned if there is no root object or the root object is not a sub-class of the specified class id.

This function always adds a persistent reference to the object it returns. That is to say, it sets it not purgeable. The reference can be removed by calling the objects **oSetPurgeable()** method.

bool isDirty(void)
Return **true** if the state of the file in memory, is different to that in the disk file. This method checks whether any objects have been attached or detached since the file was last committed. If not, it iterates over all objects in the file until it finds one that is dirty.
isDirty() may be typically used to determine, whether the user of the application should be prompted to save the file, before closing it.

bool isReadOnly(void)const
Return **true** if this OFile has been opened in read-only mode.

static OFile *getFirstOFile(void)
OFile maintains a null terminated list of all OFiles that currently exist. **getFirstOFile** returns a pointer to the first OFile in the list.

OFile *getNextOFile(void)const
Return a pointer to the next OFile in the list of all OFiles. Return 0 if there are no more in the list.

static OFile *oFileOf(OPersist *ob)
Return the file in which the object **ob** exists. Undefined if the object is not in any file. This should only be called for objects that are attached to a file.

void setAutoCommit(bool autoCommit = true)
Permit the file from time to time to call the **commit()** method. This does not in anyway relieve the user of having to call commit. Its primary use is to allow the object cache to free up memory by writing dirty objects to the file.

static void setObjectThreshold(long t)
Set the maximum number of OPersist objects that can be in memory. This includes objects in all files and those not yet attached.

static long getObjectThreshold(void)
Return the maximum number of OPersist objects that can be in memory.

static long getObjectCacheCount(void)
Return the number of OPersist objects currently in memory.

static void new_handler()
The default new handler. This is called when the object threshold is exceeded. It tries to free up space by purging objects from the cache. See **purgeAll()**.

static New_handler set_new_handler(New_handler newNewHandler)
Set a user defined new handler. Returns the previous new handler. The newNewHandler should be declared with type **OFile::New_handler**.

static long purgeAll(void);
Purge all files. If a file is marked autoCommit the purge process may also commit the file. The strategy implemented in purgeAll(), is to purge all objects in memory. This is good when sequentially writing a large file.
Returns the number of objects purged.

void setRetainIdentity(bool retainIdentity)
Set retainIdentity on if true, off if false.
This feature allows detached objects to retain their unique file identity. This can be useful when moving them to a new OFile, if you have relationships embedded as object identity. It should however be used with care to avoid the possibility of duplicate identities.

bool retainIdentity(void)const
Return true if retainIdentity is set.

static void setUserSourceVersion(const long v)
This sets the version number of the the current source code. That is, files committed using the current source code will have this version number. It should be called before any files are opened.

The following methods are most often called in the read constructor to perform object evolution. A pointer to the objects OFile can be obtained by the **OIStream::file()** method.

long userSourceVersion(void)const

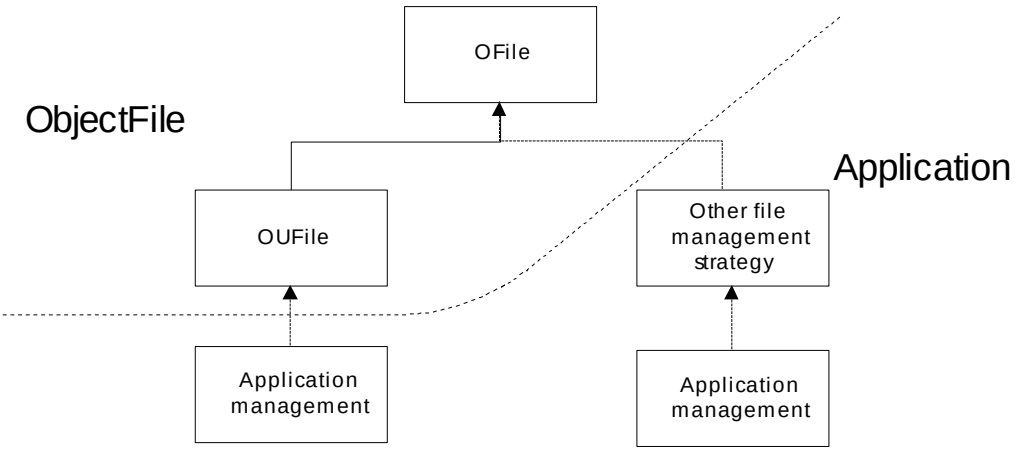
Obtain the version number of the current source code. This is set by **setUserVersion().**

long userVersion(void)const

Obtain for an existing OFile its version number. This is the number that was set by setUserSourceVersion() before committing the file. It can be used when reading objects, to determine what attributes are different in the varying versions of the file that are supported.

Extending OFile

OFile is designed to be extended by deriving from it. A derived class can contain all the application management functionality required, in order to manage application files. **OUFile** is a sub-class that adds a particular file management strategy. Other strategies can also be designed by sub-classing OFile.



Of course, you are free to extend **OFile** in other ways apart from sub-classing.

OFileErr, OFileIOError, OFileThresholdErr (ox.h), OFMutex, OFGuard (ofthread.h)

OFileErr

Purpose

This is the base class for all exceptions thrown by ObjectFile. When it is thrown, it indicates a failure of the method from which it was thrown.

Methods

const char *why(void)

Returns a pointer to a string containing a description of the error that caused the exception.

OFileIOError

Sub-class of OFileErr. This is thrown when an IO error occurs. For example if the disk is full or a file has been damaged.

OFileThresholdErr

Sub-class of OFileErr. This can be thrown by the **new operator,** of any class derived from **OPersist**. It can only occur if the object cache is turned on.

Example

```
try{
    f->commit(false);
}catch(OFileErr x){
    cout << x.why() << '\n';
}
```

OFGuard

OFGuard(OFMutex &mutex)

Guard the enclosing block with a mutually exclusive semaphore. When this goes out of scope, i.e the destructor is called, the semaphore is released.

OFMutex

OFMutex()

Declare a semaphore.

Example

```
static OFMutex mutex;
{
    OFGuard lock(mutex);
    // Critical section
    gSomeGlobal++;
} // End of critical section
```

OIStream (oistrm.h)

Purpose

OIStream is used to read data from the file. It is passed to, and used inside the read-constructor of a persistent object. The functions should be used to read data written by the equivalent functions of OOStream. See the section Reading Objects for an example of usage.

Methods

OFile *file()const
Return a pointer to the OFile that is being read. Returns 0 if not reading from an OFile. There should usually be no reason to use this function.

O_LONG readLong(void)
Read a long word (4 bytes)

O_LONG64 readLong64(void)
Read a 64 bit long word (8 bytes)

float readFloat(void)
Read a float (4 bytes)

double readDouble(void)
Read a double (8 bytes)

O_SHORT readShort(void)
Read a short word (2 bytes)

char readChar(void)
Read a single byte character.

O_WCHAR_T readWChar(void)
Read a single wide character.

bool readBool(void)
Read a boolean value.

char * readCString256(void)
Read a null terminated string.
String is limited to 256 chars including null terminator. The return value is overwritten on the next call.

O_WCHAR_T *readWCString256(void)
Read a null terminated string of wide characters.
String is limited to 256 wide characters including null terminator. The return value is overwritten on the next call.

The following **readCString** functions cane all be used to read strings written by **OOStream::writeCString**.

void readCString(char * str,unsigned int maxlen)
Read a null terminated string.
String is limited to 64k.
Parameters: str - buffer in which to put string. (Must be long enough)
maxlen - maximum string length. This is used as a check only.

void readWCString(O_WCHAR_T * str,unsigned int maxlen)

Read a null terminated string of wide characters.
String is limited to 64k.
Parameters: str - buffer in which to put string. (Must be long enough)
maxlen - maximum string length. This is used as a check only

char *readCString(void)
Read a null terminated string.
String is limited to 64k.
Return value: char buffer allocated by the function using **new char[],** containing string. User is responsible for deleting it.

O_WCHAR_T * readWCString(void)
Read a null terminated string of wide characters.
String is limited to 64k.
Return value: char buffer allocated by the function using **new O_WCHAR_T[],** containing string. User is responsible for deleting it.

char *readCStringD(void)
Read a null terminated string.
String is limited to 64k.
Return value: char buffer containing string. Deleted automatically on next call to the function. User must **not** delete it.

O_WCHAR_T * readWCStringD(void)
Read a null terminated string of wide characters.
String is limited to 64k.
Return value: char buffer containing string. Deleted automatically on next call to the function. User must **not** delete it.

void readBytes(void *buf,int len)
Read a number of bytes.
buf - buffer in which to put bytes. **len** is thenumber of bytes to read.

void readBits(void *buf,int len)
Read a number of bits.
buf is the buffer in which to put bytes. **len** is the number of bytes to read.

void readObject(OPersist **obp)
Read an object deferred until finish()
obp is the address of a pointer in which to place the reference to the object.
This is along with readObjectId is the most efficient way of reading relationships.
Note: The reference, obp is only initialized after the read constructor has terminated.

OPersist *readObject(void)
Read an object immediatly.
Return value: Pointer to an object or 0 if null reference.
Note: In the special case of bi-directional persistent references (i.e. two persistent objects pointing to each other) the object to which the return value points, may not be fully formed. It can be immediatly regarded as a pointer to an OPersist, but to its full object, only after the OFile is fully constructed. The reason for this, is that in C++, it is impossible to fully construct two objects simultaneoulsy.

OId readObjectId(void)
Read an object identity.

long userVersion(void)const
Return the version of the file that is being read. See the chapter on Object Evolution.

long userSourceVersion(void)const
Return the current version of the source code.

Sub-classing OIStream

OIStream is an abstract base class for a stream that reads objects from a file. The interface it provides is independent of the input media. This means that it is possible to derive other types of stream from it. The read-constructor of objects derived from OPersist can then be used to create objects from some other media. One example would be to receive objects from a network connection, another from the system clipboard.
This can 'pay' for the investment you have made in writing read-constructors and oWrite methods for your objects.

OIterator (oiter.h)

Purpose

This is the means by which the file contents are accessed. It can be used on a particular type of object, or the hierarchy of a particular type. It takes the form of a template to allow type safe access to the objects.

Methods

OIteratorT(OFile *ofile,bool deep = true)
Construct an iterator starting at the first object of the iterators type, in the file. If deep is true, then traverse all sub-classes.

void reset(void)
Set the iterator to the first object in the list.

OPersist * object(OId id)
Return pointer to the object with identity id. If it does not exist return 0. This does not alter the position of the iterator.

T* operator*()
De-reference the iterator to obtain a pointer to the object. Returns zero on reaching the end of the list.

T* operator++()// prefix ++a
Increment the iterator and return a pointer to the object in the new position. . Returns zero on reaching the end of the list.

T* operator++(int i) // postfix a++
Increment the iterator and return a pointer to the object in the old position. . Returns zero on reaching the end of the list.

The non template class OIterator has a slightly different constructor.
OIterator(OFile *ofile,OClassId_t cId = cOPersist,bool deep = true)
Construct an iterator, starting at the first object of the type denoted by OClassId_t cId, in the file. If deep is true, then traverse all sub-classes.

Example

An iterator is declared for the class Person.

```
class Person{
public:
// Define iterator for this class
typedef OIteratorT<Person,cPerson> It;
};
```

The iterator is used to access the Person objects in the file.

```
Person::It it(addressDB);
Person *p;

// Iterate over all Person objects.
while(p = it++)
{
// Print ascii representation of the object.
p->oPrint(cout);
}

Person *husband = (*it);   // dereferencing the iterator
```

OMeta (ometa.h)

Purpose

Many of the mechanisms in ObjectFile require that certain information about the persistent classes is available, even when an instance does not exist. This information is known as meta-data, and is stored in a meta-class. The information is available to the application developer through **OMeta**.

Methods

OMeta(OClassId_t id,Func f,...)

Constructor. There must be an instance of an **OMeta** for each persistent class used in ObjectFile. It can be instantiated as a static in the .cpp file containing the code for the implementation of the class.
id is the class identity of the class being represented. **f** is a function which creates an instance of the class. It should take the following form:

static OPersist *New(OIStream *s){return new T(s);}

T is the class being represented.
Finally the variable number of parameters are the identities of the superclasses of the call being represented. It must be terminated by a parameter of 0.

~OMeta()

Destructor.

static void initialize(void);

Initialize the data structures of OMeta. This must be called after all the instances of OMeta exist. It may be called again if classes are dynamically added to an existing system. It is called in the constructor of **OFile**. If you want to use **OMeta** before an **OFile** is opened then **initialize()** must be called by the developer explicitly.

OClassId_t id(void)const;

Return the class identity of the class repesented by this **OMeta**.

OPersist *construct(OIStream &in)const

Create an object of the type represented. Take its data from **in**.

static OMeta *meta(OClassId_t id)

A static method to retrun the **OMeta** object of a class by identity.

const Classes &classes(bool deep=true)const

Return a reference to the set (STL) of subclasses (class id's) of the class represented. If **deep** is **false** only this OMeta's class is in the set.

Examples

```
OMeta Person::_metaClass(cPerson,(Func)Person::New,cOPersist,0);

OMeta
Transport::_metaClass(cTransport,(Func)Transport::New,cOPersist,0);
OMeta Petrol::_metaClass(cPetrol,(Func)Petrol::New,cTransport,0);
OMeta Electric::_metaClass(cElectric,(Func)Electric::New,cTransport,0);
// Multiply inherits from Electric and Petrol
OMeta
Hybrid::_metaClass(cHybrid,(Func)Hybrid::New,cElectric,cPetrol,0);
```

Extending OMeta

OMeta contains all the information about the class that is needed by ObjectFile. Application developers may wish to add more application dependent information. This can be done by sub-classing OMeta and using that sub-class in place of OMeta throughout the class hierarchy.

OnDemand (odemand.h)

Purpose

OnDemand is a holder for persistent objects that do not need to be in memory until accessed. This can save memory and load time for an application.

The overloaded -> operator allows you to work with the held object as though you are using a pointer. For this reason OnDemand's are sometimes referred to as smart pointers.

Any methods defined by OnDemand, that have the same names as corresponding methods in OPersist, must be used rather than those in **OPersist** (**oAttach, oDetach, oWrite, oSetPurgeable**).

Whenever the held object is referenced it is automatically set as not purgeable. To set it purgeable use **OnDemand::oSetPurgeable**.

The memory cost for using OnDemand, rather than a pointer is an extra 4 bytes.

Methods

OnDemand(OIStream *in)
ObjectFile read-constructor

OnDemand(OPersist *p)
Default constructor

Destroy(void)
Delete from memory the held object. Remember not to destroy objects that are attached to a file.

OPersist *object(void)const
Return the object. 0 is returned if no object is held.

OPersist *set(OPersist *p)
Set object and return the previous object.

OPersist *operator ->(void)const
Return the object. Asserts if there isn't one.

OPersist & operator *()const
De-reference the object.

OId oId(void)const
Return the persistent identity of the object. This does not cause the object to be read into memory, if it is not already there. Asserts if the held object is not persistent.

void oWrite(OOStream *out)const
Write the OnDemand to the stream. This must be called by the containing objects overridden OPersist::oWrite method. It only actually writes the object identity.

void oAttach(OFile *file,bool deep)
Attach the OnDemand to a file. This should be called by the containing objects overridden OPersist::oAttach method.

void oDetach(OFile *file,bool deep)
Detach the OnDemand from a file. This should be called by the containing objects overridden OPersist::oDetach method.

void purge(OFile *file = 0)const

Purge the memory reference. Leaves a persistent reference. This does not effect the object in any way. i.e. it does not make it purgeable. Next time an attempt is made to access the held object, its memory address will be re-resolved. **purge** should always be called if there is a chance that the object will be removed from memory.
file is the OFile to which the object is attached. If it is 0, it will be evaluated automatically.

void oSetPurgeable(bool deep = true,OFile *file = 0)const
Set the held object as purgeable. Also purges the OnDemands memory reference to the object, by setting it purgeable.
file is the OFile to which the object is attached. If it is 0, it will be evaluated automatically.

Other Types of OnDemand

OnDemand is just one instance of the template **OnDemandT**. This can be instantiated for any persistent class type. This gives typesafe access to the held object.

Example

Declare an object attribute _spouse as a member of Person. The _spouse object is not read into memory when Person is read.

```
OnDemandT<Person> _spouse;
```

The read constructor of Person would look like this:

```
Person::Person(OIStream *in):OPersist(in),_spouse(in){}
```

The oWrite method of Person:

```
void Person::oWrite(OOStream *out)const
{
    inherited::oWrite(out);
    _spouse.oWrite(out);
}
```

The OnDemand is set like this:

```
_spouse.set(new Person("Samantha"));
```

The -> operator can be used to access the object,

```
cout << _spouse->name();
```

or a regular pointer can be obtained, and operations performed on that.

```
Person *p = _spouse.object();
cout << p->name();
```

OnDemandSet (odset.h)

Purpose

Sets are often used in object systems to represent one-to-many relationships. Sets and other types of collections are sometimes required to hold large numbers of objects. More than can be held even in virtual memory, at any one time.

OnDemandSet is one possible implementation of a persistent streamable collection that can be used to hold objects, some of which are in memory, and some of which are not. It does this by holding OnDemand objects. See the reference section of OnDemand for further details.

OnDemandSet is derived from the Standard Template Library **set**, so its programming interface is almost identical.

Like any other ObjectFile object, **OnDemandSet** can be attached to a file, or not attached. However, not being attached to a file means that all objects in the set must be in memory. Detaching brings them into memory. If your intention was to store more objects than can be held in memory, it is probably not a good idea to try to detach your **OnDemandSet** from its OFile.

Methods

Since OnDemandSet is derived from STL **set**, only the additional methods used to support persistence are documented here. Refer to STL documentation for the rest of the interface.

OnDemandSet(void)
Default constructor.

OnDemandSet(OIStream *in)
ObjectFile read-constructor.

void destroy(void)
Delete the objects in the set.
Call this function only if not attached to a file, since it is forbidden to directly delete a persistent object.

void oWrite(OOStream *out)const
Write persistent data to the **out** stream. This should be called by the containing objects overridden OPersist::oWrite method

void oAttach(OFile *file,bool deep)
Attach all objects contained by the **OnDemandSet** to a file. This should be called by the containing objects overridden OPersist::oAttach method. . The **deep** parameter indicates whether referenced objects should also be attached. The precise meaning of this is application specific.

void oDetach(OFile *file,bool deep)
Detach all objects contained by the **OnDemandSet** from a file. This should be called by the containing objects overridden OPersist::oDetach method. . The **deep** parameter indicates whether referenced objects should also be detached. The precise meaning of this is application specific.

Other types of OnDemandSet

OnDemandSet is just one instance of the template **OnDemandSetT**. This can be instantiated for any persistent class type. This gives typesafe access to the contained objects.
OnDemandSet is defined as follows:

```
typedef OnDemandSetT<OnDemand> OnDemandSet;
```

Example

Declare an attribute of a class:

```
OnDemandSet _voters;
```

or a typesafe attribute:

```
typedef OnDemandT<Person> Voter;   // declare typesafe OnDemand
OnDemandSetT<Voter> _voters;       // declare set of OnDemands
```

Now use it just as any other STL set:

```
Voter voter(new Person("Joe"));         // create a new object
voter.oAttach(file);                    // make persistent
voter.oSetPurgeable();                 // set it purgeable so that it
can be removed from memory (once committed).
_voters.insert(voter);                  // add to set
oSetDirty();                            // set container dirty
```

OOStream (ostrm.h)

Purpose

OOStream is used to write data to the file. It is passed to and used inside the **oWrite** method of a persistent object, to write the persistent attributes of an object. It is an abstract class from which concrete sub-classes may be derived. See the section Writing Objects for an example of usage.

The optional **label** parameter, may be specified in order to describe the attribute that is being written. This may be useful when using a sub-class of OOStream, that needs to describe the data it is writing. **OOStreamXML** is an example of such a stream.

Methods

The following two methods are best called by declaring an instance of **ODefineObject** within a block. See **OOStreamXML** for an example of its use.

beginObject(const char * label)
Identify the beginning of an embedded object or structure. Must call endObject to identify the end.

endObject()
Identify the end of an embedded object or structure. Must call beginObject to identify the beginning.

OFile *file()
Return a pointer the OFile that is being written or 0. A subclass of OOStream can be used to write to something other than an OFile. In this case the return value will be 0.

void writeLong(O_LONG data,const char *label = 0)
Write a long word (4 bytes)

void writeLong64(O_LONG64 data,const char *label = 0)
Write a 64 bit long word (8 bytes)
To use this **OFILE_64BIT_LONGS** must be defined in odefs.h.

void writeFloat(float data,const char * label = 0)
Write a float (4 bytes)

void writeDouble(double data,const char * label = 0)
Write a double (8 bytes)

void writeShort(O_SHORT data,const char * label = 0)
Write a short word (2 bytes)

void writeChar(char data,const char * label = 0)
Write a single byte character (1 byte).

void writeWChar(O_WCHAR_T data,const char * label = 0)
Write a single wide character (2 bytes).

void writeBool(bool data,const char * label = 0)
Write a boolean value.

void writeCString(const char * str,const char * label = 0)
Write a null terminated string.
The maximum length is 64k.

void writeWCString(const O_WCHAR_T * str,const char * label = 0)
Write a null terminated string of wide characters.

The maximum length is 64k wide characters.

void writeCString256(const char * str,const char * label = 0)
Write a null terminated string **str**.
The maximum length is 256 characters, including the null terminator.

void writeWCString256(const O_WCHAR_T * str,const char *label = 0)
Write a null terminated string of wide characters.
The maximum length is 256 wide characters, including the null terminator.

void writeBytes(const void *buf,int nBytes,const char * label = 0)
Write an array of bytes.
buf is a pointer to the array. nBytes is the number of bytes to be written.

void writeBits(const void *buf,int nBytes,const char * label = 0)
Write an array of bits.
buf is a pointer to the array. nBytes is the number of bytes to be written.

void writeObjectId(OId id,const char * label = 0)
Write an object identity **id**.
Assumes OId is 4 bytes.

void writeObject(OPersist *ob,const char * label = 0)
Write an object identity.
ob is a pointer to an object attached to the file, or 0 for no object.

bool VBWrite(void)
Check whether to write virtual base class.
Return value: true if it should be written; false if it has already been written.

Sub-classing OOStream

OOStream is an abstract base class for a stream that writes to a file. The interface it provides is independant of the output media. This means it is possible to derive other types of stream from it. The oWrite method of objects derived from OPersist can then be used to write to some other media. One example would be to send objects over a network, another to the system clipboard.
This can ‘pay’ for the investment you have made in writing read-constructors and oWrite methods for your objects.

OOStreamXML – Sub-class of OOStream (oosxml.h)

Purpose

OOStreamXML is used to produce a textual description of an object in XML format. It makes use of the **OPersist::oWrite()** methods of objects derived from OPersist. Apart from being useful in its own right, it demonstrates an important feature of ObjectFile. The streaming methods can be used for many purposes, simply by sub-classing OOStream. This can be a very powerful tool. It is possible to output the object data into any format. Just write a new sub-class of OOStream. If you have many different classes you save a lot of coding (and consequently executable footprint), because nothing needs to be added to the classes themselves.

The **label** parameter of OOStream's write methods should be used to give meaning to the data in the XML output. In OOStreamXML it is used to supply the tags. One possible scheme is to give the object attribute variable names.

Often within your ObjectFile objects you may have structures or other objects embedded. ObjectFile does not care about this structure, only the order in which the attributes are written. However XML or any other textual output should be as readable as possible, so it would be nice to output this structure information. This is done by the methods beginObject()/ endObject() and the class ODefineObject that makes sure both are called. To illustrate this, let's say we have a structure:

```
struct point{
short x;
short y
}
```

In the oWrite method it would be written:

```
out->writeShort(point.x,"x");
out->writeShort(point.y,"y");
```

However the fact that x and y are part of point is not passed onto the stream. So we can write:

```
{
    ODefineObject d(out,"point");
    out->writeShort(point.x,"x");
    out->writeShort(point.y,"y");
}
```

Now the stream knows that x and y are part of point and can output an extra tag <point> that encloses the tags <x> and <y>.

Methods

Most of the methods are described under the reference page for **OOStream**.

OOStreamXML**(OFile *file,ostream &out)**
Constructor. **file** is the OFile on which the stream is being opened. **out** is a C++ standard library stream to which the output is to be written.

~ OOStreamXML**(void)**
Destructor

void writeObjects(const char *appName,OClassId_t classId,bool deep = true, const char *encoding=0)
Write all the objects in the **file** (given in constructor) of class **classId**. Also their sub-classes if **deep** is true(default).

appName is the XML document tag. If it is none zero then produce a document of name **appName**.
If **encoding** is non-zero(default 0) then use the string as the character encoding of the document, otherwise use "UTF-8". It is the programmers responsibility to ensure that the encoding string is one of strings permitted by the XML standard, and that all non-Unicode strings in the objects, suit that encoding.
Exceptions: Throws OFileError if there is an output stream error. In that case the stream may be only partially written.

void setXMLStyle(XMLStyle style)
Set the style of the XML file. Should be called before **writeObjects()** in order to change the default **XMLStyle::cBest**.
The various styles affect the way objects and references appear as XML.
By default, if a reference to an object that is owned by some other object is written before the object is written, then the object will appear in the XML and not the reference to it. This may make the file appear unbalanced. Imagine writing a tree structure. The XML will have a hierarchy like the tree structure itself. To get around this you can specify “o:” as the first two characters of the label parameter in **OOStream::writeObject()**. This will ensure that only a reference to the object is written.
Further control can be achieved by specifying a style other than the default. The possible values of **XMLStyle** are:
cBest Take account of "o:" only (default).
cSAX SAX readable. Objects always appear in the output before references to them. A SAX parser does not store the parsed data, so it cannot resolve object references later. This is actually the most efficient way of reading XML, because it has very little memory overhead. To support this, objects must appear before their references in the file. This option ensures that objects get written the first time that they are referenced.
cFlat All references appear as references (even if "o:" is used in the label)

void start(OPersist *ob)
Start writing an object.

void finish(void)
Finish writing an object.

void reset(void)
Reset the stream. The stream will not write objects that have been written already. This is to avoid writing the same object multiple times. To do this is maintains a list of written objects. Resetting the stream empties this list, allowing the objects to be written again.

Example

Writing the whole OFile to a text file.

```
// Open a file stream
fstream out("person1.xml",ios::out);

// Open a print stream on the file stream
OOStreamXML print(myOfile,out);
print.writeObjects("person1");
```

This shows how you might iterate over all objects, in order to write them to a text file.

```
// Open a file stream
fstream out("person1.xml",ios::out);

// Open a print stream on the file stream
OOStreamXML print(myOfile,out);
OIterator it(myOfile);

OPersist *p;
while(p = it++)
{
    print.start(p);
    p->oWrite(&print);  // Call the oWrite method of the object passing
                        // the stream as the parameter.
    print.finish();
}
```

The following would be the contents of the file person1.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<person1>
<Person type="10" ID="id1" ver="1">
    <_spouse IDREF="id3"/>
    <_watch IDREF="id2"/>
    <_firstName>John</_firstName>
    <_familyName>Doe</_familyName>
    <_street>13 Farmers Lane</_street>
    <_district>South Bronx</_district>
    <_city>New York</_city>
    <_country>USA</_country>
    <_zipCode>23456</_zipCode>
    <_weight>70</_weight>
    <_height>1.8</_height>
    <_age>33</_age>
</Person>
</person1>
```

OPersist (opersist.h)

Purpose

This is the base class from which all persistent classes are derived. It provides virtual functions that can be overridden to define the persistent behavior of objects.
Exceptions: **OFileThresholdErr** can be thrown from the constructors if the object cache is full and cannot be purged.

Methods

OPersist(void);
The default constructor.

OPersist(OIStream *);
The ObjectFile read constructor. This is used when ObjectFile reads objects from the file. Every subclass of OPersist should have a constructor with the same parameter that calls this constructor. It is this constructor's responsibility to read in any persistent data from the stream.

virtual ~OPersist(void);
Virtual destructor. Note that this means that any class you derive from OPersist will also have a virtual destructor.

OId oId(void)const;
Returns the objects identity only if the object is attached to a file, otherwise it returns an undefined result.

virtual void oAttach(OFile *,bool deep)
Attach this object to the file. That is, make the object persistent. This should be overridden if the object manages the persistence of other objects derived from OPersist. The first method called should be the inherited oAttach. The **deep** parameter indicates whether referenced objects should also be attached. The precise meaning of this is application specific.
This method would not normally be called directly, except by a derived class. . To attach objects to a file, **OFile::attach()** would be used.

virtual void oDetach(OFile *,bool deep)
Detach this object from the file. That is, make a persistent object transient. This should be overridden if the object manages the persistence of other objects derived from OPersist. The last method called should be the inherited oDetach. . The **deep** parameter indicates whether referenced objects should also be detached. The precise meaning of this is application specific.
This method would not normally be called directly, except by a derived class. To attach objects to a file, **OFile::detach()** would be used.

virtual OMeta *meta(void)const;
Object file requires that every persistent class has some information about it available. Often this is known as meta data. It is not usually needed by applications, but in some cases can be useful. This method is the accessor for the meta data of an objects class. The meta data can also be accessed without an object. See OMeta.

bool oDirty(void)const;
Returns true if the object has been changed.

void oSetDirty(void);
The application uses oSetDirty to tell ObjectFile that the object has changed in some way. ObjectFile will only write objects that are dirty. The object should usually call this method in any of the objects accessor methods that change persistent attributes. New OPersist objects are created already dirty.

virtual void oSetPurgeable(bool deep = true,OFile *file = 0);

Tell ObjectFile that the object can be removed from memory. It is not immediately removed, but only if and when the **OFile::purge** method is called. This means that no effort is wasted, purging and re-reading, if there are enough resources to maintain it in memory.

The application should only call this method if it is sure that it will not reference the object by any **C++** pointers it holds. If it wants to access the object again it will have to obtain another pointer to it through **OFile::getObject** or **OIterator**. When a new pointer is obtained, the object is automatically set not-purgeable.

By overriding this method, derived objects may pass it on, typically to owned objects. Remember that there must not be any memory pointers to a purgeable object, even in another purgeable object. Do not forget to call the super class (last).

If **deep** is **true** (the default) then owned objects will also be set purgeable. **file** can be used to indicate the objects file. This should usually not be set, as the default, 0 is adequate.

If the **OF_REF_COUNT** pre-processor definition is defined oSetPurgeable() works slightly differently. You should define this if you intend writing an application that will be used in a multi-threaded environment. OPersist maintains a reference count. A call to oSetPurgeable() only decrements that count. If the count reaches 0, the object is set purgeable. OPersist objects are born with a reference count of 1. The reference count is incremented by any call to **OFile::getObject().** In this environment you should match up calls to getObject() with OSetPurgeable(). See the section entitled Multi-threaded Applications for a fuller explanation.

bool oAttached(void)const;

Return **true** if the object is attached to a file. OPersist objects are valid objects for all purposes, whether they are attached to a file or not.

virtual void oWrite(OOStream *)const;

Write persistent data to the stream. This should always be overridden when there is persistent data. The first method called should be the inherited **oWrite**. The method is purposely **const** to avoid changing the object when writing it.

void addRef(void)const

Add a reference to the object. This method is only defined when **OF_REF_COUNT** is defined at compilation time.

void removeRef(void)const

Remove a reference to the object. This method is only defined when **OF_REF_COUNT** is defined at compilation time.

OProtect (opersist.h)

Purpose

OProtect is a simple class that can be used as an exception safe way of making objects purgeable. It is exception safe because the object is made purgeable by the destructor. For this to work it should be declared on the stack and not created with **new**.

Methods

OProtect(OPersist *ob)

Constructor. **ob** is the object to be set purgeable on destruction of the OProtect object.

~OProtect(void)

Destructor. When this is called the object specified in the constructor is made purgeable.

Example

OProtect would typically be used as follows:

```
{
    Person *person = getObject(id);
    // Set person purgeable when we go out of scope
    OProtect  pr(person);
    // Call a function from which an exception could be thrown.
    person->doSomething();
}
// person will be purgeable when we get to here, or if we are
// thrown out by an exception throw.
```

OSet (oset.h)

Purpose

Sets are often used in object systems to represent one-to-many relationships. Other types of collections can also be used for this purpose. **OSet** is one possible implementation of a persistent streamable collection. It is derived from the Standard Template Library **set**, so its programming interface is almost identical.

The ObjectFile methodology allows the developer to use the class library of his choosing in his persistent classes. Collections are usually a major feature of any particular class library. It would not be consistent to insist on using a collection provided by ObjectFile to represent one-to-many relationships. For this reason OSet should be regarded as an example of how to derive your own streamable collections from your chosen collection library.

OSet holds simple **C++** pointers to its contained objects. This means that the objects must be in memory when the OSet is in memory. This may be sufficient for many needs. More sophisticated memory handling can be obtained by using **OnDemandSet**.

Methods

Since OSet is derived from STL **set**, only the additional methods used to support persistence are documented here. Refer to STL documentation for the rest of the interface.

OSet(void)
Default constructor.

OSet(OIStream *in)
Object file read constructor.

void destroy(void)
Delete the objects in the set.
Call this function only if caller is not persistent, since it is forbidden to directly delete a persistent object.

void oWrite(OOStream *out)const
Write persistent data to the **out** stream. This should be called by the containing objects overridden OPersist::oWrite method

void oAttach(OFile *file,bool deep)
Attach all objects contained by the **OSet** to a file. This should be called by the containing objects overridden OPersist::oAttach method. . The **deep** parameter indicates whether referenced objects should also be attached. The precise meaning of this is application specific.

void oDetach(OFile *file,bool deep)
Detach all objects contained by the **OSet** from a file. This should be called by the containing objects overridden OPersist::oDetach method. . The **deep** parameter indicates whether referenced objects should also be detached. The precise meaning of this is application specific.

Other types of OSet

OSet is just one instance of the template **OSetT**. This can be instantiated for any persistent class type. This gives type-safe access to the contained objects.
OSet is defined as follows:

```
typedef OSetT<OPersist> OSet;
```

Example

Declare an attribute of a class:

```
OSet _children;
```

or a typesafe attribute:

```
OSet<Person> _children;
```

Now use it just as any other STL set:

```
Person *child = new Person("Jeremy")); // create a new object
file->attach(child);                   // make persistent
_children.insert(child);               // add to set
oSetDirty();                           // set container dirty
```

OUFile – Sub-class of OFile (oufile.h)

Purpose

OUFile is a subclass of OFile. It adds a file management strategy to the minimal facilities provided by OFile.

ObjectFile recognizes that the requirements for the management of the application file are application dependent. Applications may require various degrees of backup, incremental saves, or integration with an application development framework. Sub-classing **OFile** can provide all these. **OUFile** is one possible subclass that implements one of many possible strategies. It can be used as it is, or be seen as an example of how to implement another strategy.

OUFile's file management strategy involves, working with a work file. There are several advantages to this. The application user only expects his file to be updated when he gives the 'Save' command. However we want to give him maximum security against losing valuable work, in case of a power failure or application crash. If we work against a work file we can perform commits after every major function the user performs. The temporary file is continually updated. In case of failure, when the application is restarted, it can look to see if a work file exists, and can ask the user to reload the file.

If a file is opened **readonly**, then the overhead of building a work file is unnecessary, and it is not built.

If a file is being created, and the **fname** parameter is 0 (i.e. an unamed file is being created), then an empty temporary file is created. The application can then be named with a 'SaveAs'.

If a work file name is specified in the constructor, then it is used as the name of the workfile. If 0 is specified, a temporary file name is generated.

It is possible to specify that a backup file will be created. This is done by specifying a paremeter to the **save()** method.

Methods

OUFile is a subclass of OFile. It therefore inherits methods fom OFile. See the page for OFile for details of those methods.

OUFile(const char *name,long operation,const char *workName = 0,const char *magicNumber = 0)
Constructor.
fname is a pointer to the full path name of the file. It may be 0 to indicate an as yet unamed file.
operation defines how to open the file. It can take the following values:

OFILE_CREATE - Create a new file. This will overwrite an existing file of the same name.

If the file cannot be created an exception will be thrown.

OFILE_OPEN_FOR_WRITING - Open an existing file for writing. If the file does not exist an exception will be thrown. If or'ed with **OFILE_CREATE** and the file exists, the existing file will be opened.

OFILE_OPEN_READ_ONLY - Open an existing file readonly. If the file does not exist an exception will be thrown.

The following flags may be ore'ed together with the previous flags.

OFILE_FAST_FIND - An extra index will be built, that will allow objects, of unspecified classes, to be found much more quickly than otherwise. This extra speed is particularly useful when many objects have to be connected in a file that is being loaded. It can improve file loading speed enormously. The index requires extra

memory, but this can be released by calling **fastFindOff()** after the objects have been connected, or at any other time.

OFILE_AS_COMPOUND - Force an OLE Compound Document to be created.

workName - The name of the work file can be specified. If this is 0, then a temporary file name is generated. This is useful for implementing a recovery scheme.

magicNumber is a 4 byte character string that can optionally be used to identify a file type. If it is specified, ObjectFile will use it to test if the file being read has the same magic number as the parameter. If not, an exception will be thrown. The same magic number can be used by operating systems to distinguish the file from other files. For this purpose the offset in the file must be known. ObjectFile always stores it magic number at offset 28 bytes in the file.

Exceptions: OFileErr is thrown on failure to open file.

~OUFile()

Destructor. Closes the file and deletes all persistent objects from memory. If you have not committed and saved, then all changes will be lost.

static OUFile *getOUFile(const char *fileName)

Return a pointer to the OUFile with the name fileName. Note that this must be specified as it was specified when the file was opened.

const char *fileName(void)const

Return a pointer to the file name of this OUFile.

bool isDirty(void)

Return true if the state of the last saved file differs from that of the last commit or from the objects in memory. The OUFile will be not dirty when 2 conditions are met:
(i) No objects are dirty.
(ii) The file has been saved since the last commit.

void makeWritable(const char *workName)

Make a file that is open for reading, open for writing.
workName - The name of the work file can be specified. If this is 0, then a temporary file name is generated.

void save(const char *backupFile = 0);

Save the file with the current file name. **OFile::commit()** should be first called if dirty objects are to be saved. If **backupFile** points to a null terminated string, a backup file will be created, with the string as its file name extension.

void saveAs(const char *name)

Save the file with a new file name. . **OFile::commit()** should be first called if dirty objects are to be saved. This will assign a file name to the object or change the name, if it already has one.

const char *workFileName(void)const

Return a pointer to the work file name of this OUFile.

5. Implementation Notes

5.1 The Back-end

ObjectFile uses Standard Template Library(STL) containers to maintain its vital data structures. This means that those containers can be replaced with any other containers conforming to the STL interface.

5.2 STL Library

An implementation of STL is supplied with ObjectFile. If your development environment has its own good STL implementation, then we encourage you to use that, instead of the one supplied.

STL contains some functions called ::allocate(). These are located in defalloc.h. They handle memory allocation failures very simplistically. In fact, they just print a message and terminate the program. This would normally not be good enough in an industrial strength application. Since memory failure is an application dependant function, it is left to the developer to incorporate his application wide scheme, into these functions.

One possible strategy is to call **OFile::purge** to free up memory and then try to allocate again.

5.3 ObjectFile as a DLL

There is nothing in ObjectFile preventing it being built and used as a 16 or 32-bit MS-Windows DLL. In particular it does not depend on any globals.
To do this in MS-Windows a macro EXPORT should be placed after the class statement in every class header. This will allow the class and method definitions to be exported.
It could also be built as a UNIX shared library.

5.4 Using ObjectFile with other Object Frameworks

OPersist objects are the base class of all persistent objects. As delivered OPersist is not derived from anything. However there is no reason why it should not be derived from your object frameworks base class. For example, in Microsoft Foundation Classes this would be **CObject**.

Another possibility, one that would not require any changes in ObjectFile, is to multiply inherit your derived classes from both **OPersist** and the frameworks base.

Neither of these options have as yet, been tried.

5.5 Porting ObjectFile

ObjectFile is designed to be highly portable through use of standard **C++** . The standard io library of **C** does not define file sharing properties. In particular it may be that a file opened by one application as read-only can be opened by another as writable. This can obviously cause problems. For this reason ObjectFile has implemented native io for some operating systems (including **WIN32**). The developer using an operating system not covered, may want to implement his systems native io. There is a well-defined interface for this, along with the existing implemetations in **oio.cpp**.

Some parts of ObjectFile require the protection of **semaphores** to work properly in a multi-threaded environment. These are usually not portable. ObjectFile wraps them in the classes **OFMutex** and **OFGuard**. They are defined in **ofthread.h**. If you define **OF_MULTI_THREAD,** and an implementation of these classes, for your

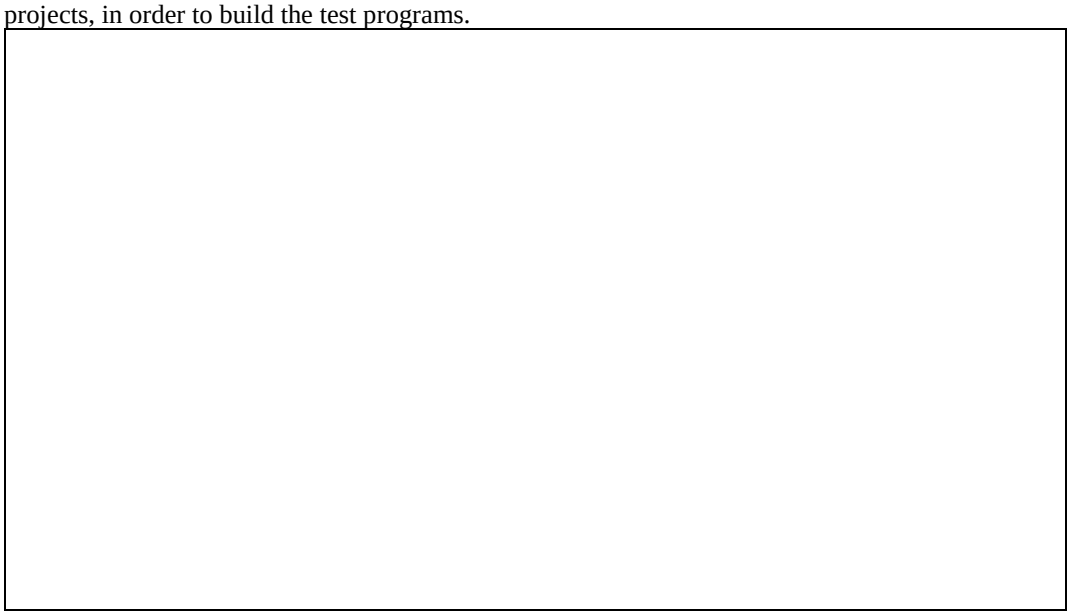
platform, does not exist in ofthread.h, you will have to add one. This should be very trivial.

5.6 Wide Character Support

ObjectFile writes 2 bytes for each wide character, independent of your platforms memory representation of **wchar_t**. Since it is not uncommon for developers to define their own type for wide characters the typedef **O_WCHAR_T** is used throughout ObjectFile code. This is defined in **odefs.h** and can be redefined by the developer.

6. Installation Notes

ObjectFile is delivered as a single zip file. When unzipped the following directory structure will be built. Unfortunately this is the only way ObjectFile can be supplied at the moment. If you have a non-PC platform you will have to install on a PC and transfer the files yourself.
ObjectFile is designed to be fully platform and compiler portable, but not been tested on every single platforms.
A Microsoft Visual C++ 6.0 project is supplied. A Borland project can be supplied on request. Other platform developers will have to develop their own Makefiles and projects, in order to build the test programs.



6.1 TEST

This directory contains a variety of test and demonstration programs for ObjectFile. All the programs are written in standard C++ and do not use any framework specific libraries. For this reason they are very bland looking. They do however exercise quite well ObjectFile.
A Borland IDE project can be provided on request. This contains targets for all the test programs.
The test programs are composed as follows:

address.exe

address.cpp,person.cpp,ofile.lib

ofile.exe

mycolct.cpp,myblob.cpp,main.cpp,myclass.cpp,ofile.lib

bm.exe

bm.cpp,mmyclass.cpp,ofile.lib

bm_db.exe

bm_db.cpp,mmyclass.cpp,ofile.lib

bm_mt.exe (Borland dependent)

bm_mt.cpp,mmyclass.cpp,ofile.lib
Use thread safe compilation and libraries. Define OF_MULTI_THREAD. In Borland add rwstd.cpp.

odsettst.exe

odsettst.cpp,myclass5.cpp,ofile.lib

settest.exe

settest.cpp,myclass.cpp,myblob.cpp,ofile.lib

ofile2.exe

trans.cpp,permain.cpp,person.cpp,ofile.lib

blobtest.exe (Does not work on MSVC due to use of dirent.h)

blobfile.cpp,blobmain.cpp,ofile.lib

evolve.exe

evperson.cpp,evmain.cpp,ofile.lib
This with the next two programs should be run one after the otehr in the same directory. They demonstrate class evolution.

evolve2.exe

evmain2.cpp,evper2.cpp,ofile.lib

evolve3.exe

evmain3.cpp,evper3.cpp,ofile.lib

odrel.exe

odrelm.cpp,odrel.cpp.cpp,ofile.lib

ole.exe

olemain.cpp,trans.cpp,person.cpp,ofile.lib
For OLE support define **OF_OLE** when compiling all files, including ofile.lib. Obviously this is only relevant to Windows platforms.

6.2 BM_OWL

This directory contains a benchmark program for ObjectFile. It uses Borland OWL to display the results. No attempt has been made to make it portable. Please feel free to port it to other platforms.
A Borland IDE project for the benchmark can also be provided on request.
bm.exe is a portable version using standard I/O.

6.3 OFILE

This directory contains the ObjectFile sources.

6.4 PROJECTS

This directory contains projects for various environments. Currently there are projects for Microsoft Visual C++ 6.0 and Borland C++ Builder 5.0.

6.5 STL

This directory contains the Standard Template Library include files. It is the Silicon Graphics Portable version (2.0), which seems to be the best and most portable implementation around. It is freely available on the Internet (http:\\www.stlport.org). It can be replaced by any other implementation, including that supplied with your compiler.
Newer implementations of STL use namespace. Not all compilers support this, so it is not by default switched on in ObjectFile. STL implementations that do use namespace, usually have a compilation switch to compile it without. In Borland 5.0(RogueWave) the switch is #define RWSTD_NO_NAMESPACE . This is already defined in odefs.h.

6.6 TEMPLATE

This directory contains templates for your classes .cpp and .h files. They should be used as a starting point for writing persistent classes. They can be copied and modified, as needed.

6.7 Borland 4.5(and probably others) bool problem

There is a clash between Borland 4.5 and STL header files. **bool** is defined differently in both of them. This is an annoying problem that should disappear in the next generation of compilers, since the latest C++ standard defines **bool** as a type. There are no doubt several imperfect ways of overcoming the problem in the meantime. One way is to comment out the line:

```
#define BI_NO_BOOL              // boolean type                 DEFS.H
```

in the Borland classlib\compiler.h header file.
In Borland 5.0 onwards, using the Borland STL this problem does not exist.

6.9 Visual C++ 6.0

ObjectFile has been tested using the SGI portable STL(Version 3.12). This can be freely obtained from (http:\\www.stlport.org).
It also works with the Microsoft STL (as delivered with Visual C++ 6.0)though you must use the standard library in its std namespace. It also is known to have other problems. It is worth reading http://www.dinkumware.com/vc_fixes.html before committing to use it.

6.10 Making ObjectFile

There are so many different development frameworks available. It is impossible to supply a Makefile or Project file for them all. A MSVC 6.0 IDE project file only, is supplied.
ObjectFile has no special requirements for making it. It is recommended that you build the files in the OFILE /OFILE directory into a link library. This can then be linked to any other source code. In this way, only the object code used will be included when linked.

Pre-processor options (odefs.h)

One or more of the following defines may be defined. If defined, then all modules using ObjectFile must be compiled with them. To ensure this it is best to define them in the **odefs.h** file.
OF_OLE - define for OLE support.
OF_REF_COUNT - causes OPersist objects to be referenced counted.
OF_MULTI_THREAD - define this for multiple threads. It also causes OF_REF_COUNT to be defined.
OFILE_STD_IN_NAMESPACE - define this if your standard library is defined in the **std** namespace.
OFILE_64BIT_FILE_ADDRESSES - On Windows 2000/XP you can write files up to 2**44 - 64Kb. On 64 bit platforms you can also write large files. The IO in oio.cpp is implemented for Windows 2000/XP so you can use it right away, as long as your compiler supports __int64 or an equivalent. It will not work on other Windows platforms. Note that defining this will make the file format only compatible with files written with the same define. All file addresses become 8 byte, so objects require an extra 4 bytes of storage. Tests show almost no detrimental effect on performance.
OFILE_64BIT_LONGS - If you want to store 64 bit longs then define this. Your compiler must have the type __int64 or an equivalent.

Constants

cCOMaxClasses - defines the maximum number of persistent classes that may be defined. Your ClassId must not exceed this number less one. The default is 80. It can be increased if necessary.
COFileMaxLength() - defines the maximum file length. The default is 2GB. 4GB can be supported by WIN32. However be careful, because files that grow beyond 2GB will not be readable on a system that supports files of 2GB.

List of files to compile in order to build ofile.lib

ofile2.cpp, oflist.cpp, oio.cpp, oistrm.cpp, oiter.cpp, ometa.cpp, opersist.cpp ostrm.cpp, oufile.cpp, ox.cpp, obuf.cpp, oconvert.cpp, ofile.cpp, oblobp.cpp, oosxml.cpp

6.11 Trouble shooting

Experience shows that the most difficult aspect of getting ObjectFile to work in a given environment is the STL (standard template library), which it uses extensively. This is mainly because STL has been undergoing a process of standardization and the compilers have been undergoing a process of catching up. If you find that ObjectFile does not compile at the first attempt, do not lose hope. Try to get some simple STL programs compiled. Once you have done this use the same defines/compiler settings etc. to build ObjectFile. Start with the address.exe test program.

Appendix A: ObjectFile Binary File Layout

The layout of the binary file is shown here for the purpose of debugging only. It may differ in subsequent releases of ObjectFile.