Building Object-Oriented Server-Type DLLs for Windows 3.x using MFC 2.5 and OWL 2.0

by Ted Faison

Ted Faison has written several books and articles on C++ and Windows. He is president of Faison Computing, a firm that develops C++ class libraries for MS-DOS and Windows, and can be reached at tedfaison@msn.com.

Client-server systems are popular these days. Servers are programs designed to provide (what else?) services to client programs. Servers can be executing on the same computer as the clients, or on a separate computer. Client-server architectures simplify large systems, because complex functionality can be bottled up into a separate process for clients to access through a predefined and (hopefully) simple interface. In this article, I will discuss the implementation of a generic server, capable of providing some kind of service to local application programs. One of the things that makes servers different from ordinary programs is their ability to be called by multiple application programs. The server needs to have an internal structure that enables it to keep manager and track the requests from different applications.

There are several standard server DLLs that Windows applications make use of, like the Print Manager and the ODBC (Open Database Connectivity) server. The type of server DLL I will describe is like the ODBC server, because it is capable of being called by any number of application programs (the clients), and utilizes an internal data structure to manages its clients.

Designing a server is not much different from developing other Windows programs, with a couple of exceptions. First, servers are often implemented as DLLs, to allow multiple client applications on the same machine to share server code. Second, servers often need to allocate and manage memory on behalf of the client applications, but without knowledge or intervention from the clients. Consider how the ODBC server is used: client applications make requests to the server to connect to databases, log on to databases, use the databases, and log off. The server must remember which databases each client is connected to, and which state the clients are in. All this knowledge must be stored somewhere inside the server for each client. The *somewhere* is a block of memory allocated for each client. Clients have nothing to do with the management of this memory, know nothing about it, nor have access to it. As far as the clients are concerned, the server provides a service. Internal details of how the service is provided are of no concern to the clients.

By default, global memory allocated in a DLL is shared by all the calling application programs, which I'll refer to generically as *tasks*. A server-type DLL is a DLL that allocates a separate block of memory for each calling task. There are several ways for server DLLs to manage task-dependent memory. One method is for the DLL to allocate memory directly in the calling task's data segment. This method is used in the implementation of the E*xtension DLLs* in the MFC library. Extension DLLs have objects of class `CDynLinkLibrary` that internally manage a linked list of task-dependent data. To create your own extension DLL, you derive a class from `CDynLinkLibrary`. When memory is accessed in the DLL, the `CDynLinkLibrary` linked list is searched to get the memory associated with the calling task. One limitation of MFC Extension DLLs is that the calling task must be an MFC application. For a server to be truly generic, it should be callable from applications developed with any class library (including none) and any language.

Having a DLL use memory allocated either by a client or in a client's data segment can cause trouble in certain situations. Consider the client-server database example. If the client program aborts in the middle of a transaction, the server may be required to carry on some clean-up processing after the client has terminated. The problem is that the Windows kernel will have deallocated the client's memory, causing a probable RIP the next time the server tries to access it. To obviate the problem, the server must allocate, manage and deallocate client-dependent memory on its own.

To develop a server-type DLL under Windows 3.x, the DLL must avoid the use of static or global variables for application data. The former wind up on the stack of the calling application. The latter are placed in the DLL’s data segment, which is shared by all the DLL’s callers. A server DLL must instead allocate memory dynamically for each client.

One for Microsoft, One for Borland

The technique for creating the server DLL discussed in this article is relatively generic, and I will discuss two different implementations, using Microsoft and Borland compilers. One implementation will use Visual C++ 1.5 with MFC 2.5, the other will use Borland C++ 4.0 with OWL 2.0. The implementations are equivalent, and differ primarily in the use of the vendor-specific container classes used to manage the task-dependent memory. Both are called SERVER.DLL, are very simple, and allow an application program to store and retrieve strings and integer values from memory inside the server. I also developed two client applications that look and act the same way. The first is implemented with MFC and is called TEST.EXE, the second uses OWL and is called SERVTEST.EXE. They have a simple menu for reading and writing the data stored in the server DLL. Figure 1 shows the Write popup menu, figure 2 the Read menu.

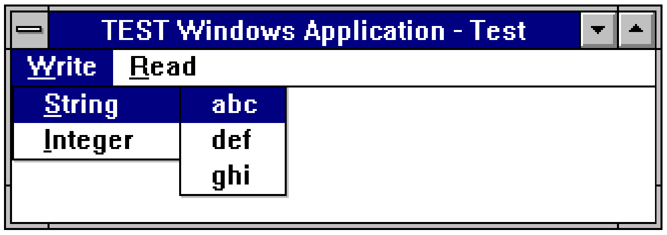


Figure 1 - The menu used to write data into the server DLL memory.

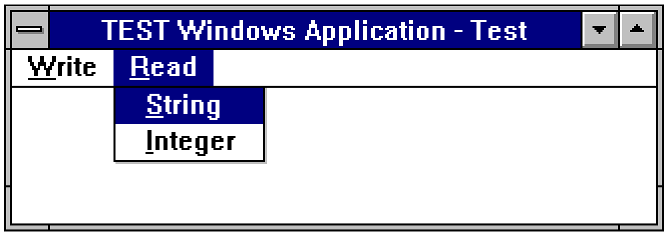


Figure 2 - The menu used to read the server DLL memory.

To facilitate the reading of this article for those of you interested in only one of the implementations, I will discuss the Microsoft and Borland examples separately.

The MFC Implementation

The first step for the server DLL is to identify tasks (i.e. clients) that use the DLL. Each time a new task calls the DLL, a new block of global memory will need to be allocated and set aside for that task. When this task later calls the DLL again, the task's memory will be used. But there is a small problem that must be solved. If memory is allocated using `GlobalAlloc()`, the memory will be owned by the calling task. When the task terminates, the kernel will automatically free this memory. To allow the server to access the task's memory after it terminates, the memory block must not be unloaded until any cleanup required is completed. The solution is to allocate task memory using the `GMEM_SHARE` flag, so Windows will free the memory automatically only when the server DLL is unloaded from memory, or when the DLL explicitly frees it.

A server DLL must be able to set up and manage task-related memory without any intervention or help from the client tasks. In my implementation, the server allocates block of memory to each task, and manages the blocks using C++ container objects. When a task makes a call to one of the server functions, the server will check the memory block container to see if memory has been allocated to the task. If so, the memory is used, otherwise a new block of memory is allocated and its handle stored in the container.

A dictionary container is used to store and manage associations of Windows task handles and global memory handles. Associations are also containers, and manage two objects that are linked by some relationship. Associations don't store information about the relationship itself. It is up to the programmer to keep track of what relates the two objects. The first object is called the *key* of the association, and the second is called the *value*. The memory handle for data allocated to a task is stored as the *value* of the Association object. The handle of the task that owns the memory is stored as the key value. Figure 3 shows a graphical description of the way client memory is managed in the server DLL described in theis article:

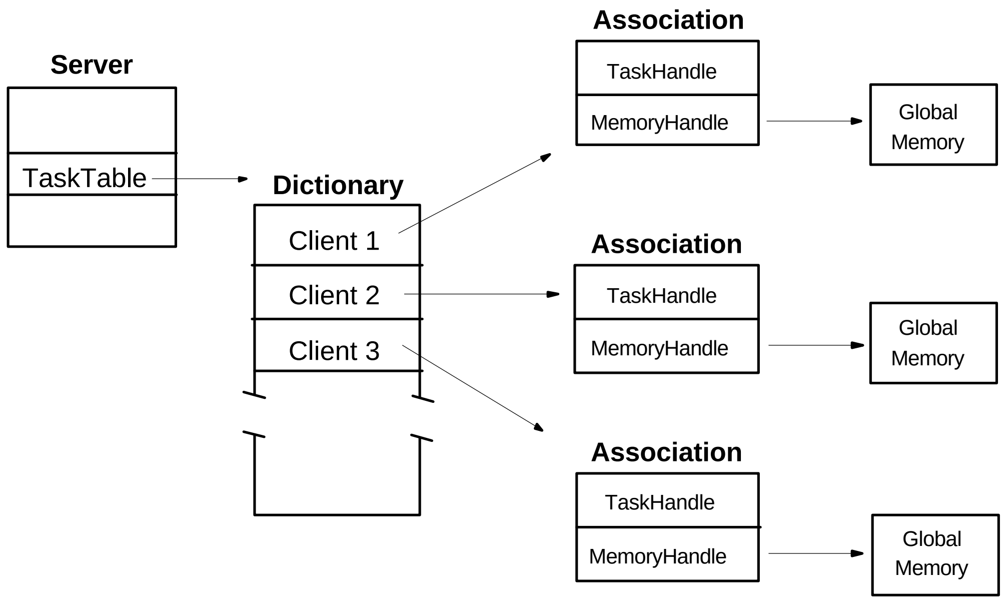


Figure 3 - How the server DLL keeps track of task-dependent memory.

Dictionaries are also called *maps* in MFC parlance, and MFC has a number of specialized maps to handle different kinds of associations. For the server in this article, I used the MFC class `CMapPtrToPtr`, which associates a pointer to a pointer. This is the most generic kind of dictionary, supporting associations bertween arbitrary 32-bit entities. Since the map is used to manage task handles and global memory handles, some typecasting is necessary to store and retrieve handles from the `CMapPtrToPtr` dictionary.

To implement a DLL in MFC, you typically start by creating a class derived from CWinApp. This class will take care of some of the low level DLL details, such as LibMain() and WEP(). My implementation uses a called `CServerDLL`, and the dictionary is declared a data member of the class. The declaration for `CServerDLL` is shown in listing 1:

```
class CServerDLL : public CWinApp {

  BOOL AllocateMemory(HTASK);

  // container to hold task-dependent data
  CMapPtrToPtr TaskTable;

public:

  void DeallocateMemory(HTASK);

  CServerDLL(const char* name) : CWinApp(name) {}
  HGLOBAL VariablesOf(HTASK);
};
```

Listing 1 - The declaration of the server DLL class.

Class `CServerDLL` is responsible for managing the `LibMain()` and `WEP()` details of the DLL, yielding a very clean implementation. The `TaskTable` data member is the dictionary container that manages the task/memory handle associations. The member function `VariablesOf(HTASK)` returns a handle to the global memory allocated to a given task, allocating new memory if necessary. `VariablesOf (HTASK)` is shown in listing 2.

```
HGLOBAL CServerDLL::VariablesOf(HTASK task)
{
  // see if this task has already been registered
  void* memoryHandle;
  if (TaskTable.Lookup( (void*) task, memoryHandle) )
    // yes - return a handle to its memory
    return (HGLOBAL) _FP_OFF(memoryHandle);

  else {
    // no - allocate new memory for the task, and return
    // a handle to it
    AllocateMemory(task);
    if (TaskTable.Lookup( (void*) task, memoryHandle) )
      return (HGLOBAL) _FP_OFF(memoryHandle);
    else
      // memory couldn't be allocated to the task: fatal error
      return NULL;
  }
}
```

Listing 2 - The function which looks up the memory handle for a given task.

`VariablesOf(HTASK)` uses the `TaskTable` dictionary to lookup the handle of the given task. The member function `CMapPtrToPtr::Lookup(void*, void*)` does all the work. The function returns `TRUE` if the task handle is found, otherwise `FALSE`. If the handle is found, the associated memory handle is retrieved in `VariablesOf()` and returned, otherwise a new block of memory is allocated and stored with the task handle in the dictionary. The memory handle is then returned to the caller. Memory is allocated by a call to the member function `CServerDLL::AllocateMemory(HTASK)`, whose code is shown in listing 3.

```
BOOL CServerDLL::AllocateMemory(HTASK taskHandle)
{
  // see if this task has already been registered
  void* memoryHandle;
  if (TaskTable.Lookup( (void*) taskHandle, memoryHandle) )
```

```
    return TRUE;

  // the task hasn't been registered:
  // allocate a new block of memory for the task
  HGLOBAL hMemory =
      GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE | GMEM_ZEROINIT,
              sizeof(LOCAL_VARIABLES) );

  if (!hMemory) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, "Fatal Error",
             "SERVER.DLL could not successfully "
             "allocate memory for the current task.",
             MB_OK | MB_SYSTEMMODAL | MB_ICONSTOP);
    return FALSE;
  }

  // associate this memory handle with the task handle
  TaskTable [ (void*) taskHandle ] = (void*) hMemory;

  // install a callback for the task, so we can
  // be called when the task terminates
  NotifyRegister(taskHandle, CallBack, NF_NORMAL);
  return TRUE;
}
```

Listing 3 - The server member function that allocates memory to a task.

`AllocateMemory()` checks the server dictionary to see if a task handle-memory block handle association for that task is already in the container. If so, the task has memory already allocated to it and no further action is taken. If not, a new block of global memory is allocated with the code:

```
HGLOBAL hMemory =
      GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE | GMEM_ZEROINIT,
                  sizeof(LOCAL_VARIABLES) );
```

and the memory handle is saved in the container with the task handle using the code:

```
TaskTable [ (void*) taskHandle ] = (void*) hMemory;
```

The typecasting isn't pretty, but is necessary. The data member `TaskTable` is of class `CMapPtrToPtr`, and takes pointers as arguments. There is no specific container class to manage `HANDLE` variables directly - hence the (void*) typecasting. The Borland implementation, shown later, uses a template container class, and requires no typecasting.

Installing the callback notification function

In order for the server to free the memory allocated to each client task, it must know when its clients terminate. The installation of a ToolHelp notification callback function will make Windows notify the server every time a task is either started or terminated.

The last line in function `CServerDLL::AllocateMemory(HTASK)` installs a task notification callback function, using the code:

```
NotifyRegister(taskHandle, CallBack, NF_NORMAL);
```

where `taskHandle` is the handle of the task whose termination we wish to monitor. The second parameter is the address of the callback function. Since we're already in a DLL, we don't need to create an instance thunk for the callback function, so there is so need to invoke `MakeProcInstance()`. The last parameter tells Windows to invoke the callback function for "normal" events, which excludes events for task switching or system debugging.

Notification callbacks are not supported directly by Windows 3.x. To use functions such as `NotifyRegister()` and `NotifyUnRegister()`, you must enlist the services of the ToolHelp library, which is shipped as a standard component with Windows 3.1. To use the ToolHelp function, you must have the line:

```
#include <toolhelp.h>
```

in your code. ToolHelp maintains a chain (i.e. list) of notification callback functions. When a significant event occurs in the system, such as a task termination, ToolHelp searches the notification chain for a callback associated with the terminating task. If a callback is found, it is invoked. ToolHelp uses the callback's return value to decide whether to invoke other callbacks installed for the same task. If the return value is zero, the next callback in the chain is called. If the value is non-zero, no other callbacks are called for the event.

Why is a notification callback needed? In order for the task table to be useful, its contents must be kept up-to-date. When a new task calls the server DLL, it is added to the server DLL's task-memory dictionary -- a process I call *registration*. When a registered task terminates, the memory assigned to it must be deallocated. TooHelper will invoke a task's notification callback function just before the Windows kernel terminates the task. The callback function looks like this:

```
extern "C" BOOL FAR PASCAL _export
CallBack(WORD eventType, DWORD)
{
  if (eventType != NFY_EXITTASK)
    return FALSE;

  // delete any memory allocated to the terminating task
  server.DeallocateMemory(GetCurrentTask() );
  return FALSE;
}
```

The statement:

```
server.DeallocateMemory(GetCurrentTask() );
```

calls a member function of `server`, which is a global object of class `CServerDLL` declared like this:

```
CServerDLL NEAR server("server.dll");
```

The `NEAR` modifier is necessary to guarantee that the object will use a `NEAR` pointer for the `this` pointer, irrespective of the memory model being used, for MFC compatibility. When a C++ member function is called, an invisible `this` pointer is passed to it as the first parameter. The size of this parameter is determined by the way the associated object is declared: if the object is NEAR, a 16 bit `this` pointer is passed. If the object is FAR, a 32 `this` pointer is passed. MFC makes internal calls to member functions of the object server, passing a 16 bit `this` pointer, thus the need for server to be declared NEAR

When the callback is invoked by Windows for a terminating task, the task is still running, and will be terminated right after the callback function returns. Calling `GetCurrentTask()` in the callback will therefore return a handle to the task that is about to terminate. The callback uses this handle to determine whether the terminating task was registered in the DLL or not. If so, the task's assigned memory is deallocated, and the task's entry in the DLL's task table is deleted. The function `CServerDLL::DeallocateMemory(HTASK)` takes care of freeing the task memory, and is shown in listing 4:

```
void CServerDLL::DeallocateMemory(HTASK task)
{
  // see if the terminating task was our client
  void* memoryHandle;
  if (!TaskTable.Lookup( (void*) task, memoryHandle) )
    return;

  // deallocate the task's memory
  GlobalFree( (HGLOBAL) _FP_OFF(memoryHandle) );

  // remove the task handle from the container
  TaskTable.RemoveKey( (void*) task);

  // kill the callback function
  NotifyUnRegister(task);
}
```

Listing 4 - Deallocating task-dependent memory.

The function `CMapPtrToPtr::Lookup()` is used to find the memory handle associated with a given task. If a memory handle is found, the statement:

```
GlobalFree( (HGLOBAL) _FP_OFF(memoryHandle) );
```

is used to convert the `void*` stored in the dictionary into a memory handle, and to free the memory. Once memory has been freed for a task, we no longer need to monitor the task. The call to `NotifyUnRegister()` removes the callback function from the notification callback chain. ToolHelp will remove only the callback for the given task, so other tasks will continued to be monitored.

Accessing the task-dependent memory

Servers may or may not allow client applications to access their task-dependent memory directly. A typical database server would probably not allow so, but I'll write a couple a simple access functions, to show how a client might read and write server data. To be generic, assume the server allocates a blocks of task-dependent memory declared like this:

```
struct LOCAL_VARIABLES {
  int a;
  char string [1000];
};
```

The struct can be changed to accommodate any type and number of variables required. I wrote two access function to read and write the integer field of `LOCAL_VARIABLES`, and two functions to read and write the string field. The four functions are shown in listing 5:

```
// modify the task-dependent memory
void FAR PASCAL _export
SetA(int value)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // update the task data
  variables->a = value;
  GlobalUnlock(handle);
}

// read the task-dependent memory
int FAR PASCAL _export
GetA()
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return 0;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return 0;

  // return the task data
  int value = variables->a;
  GlobalUnlock(handle);
  return value;
}

// modify the task-dependent memory
void FAR PASCAL _export
SetString(LPSTR s)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // update the task data
  strncpy(variables->string, s, sizeof(variables->string) );
  variables->string [sizeof(variables->string)-1] = 0;
  GlobalUnlock(handle);
}

// read the task-dependent memory
void FAR PASCAL _export
GetString(LPSTR s, int size)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // return the task data
  strncpy(s, variables->string, size);
  s [size-1] = 0;
  GlobalUnlock(handle);
}
```

Listing 5 - The server's access functions called by client applications.

The four access functions are straightforward, and all call the member function `CServerDLL::VariablesOf(HTASK)` to obtain a global memory handle for the calling task. `GetCurrentTask()` is used to determine the task handle for the calling application. `SetA(int)` and `SetString(LPSTR)` write the client data into this global memory. `GetA()` and `GetString(LPSTR, int)` read data from the memory. The global memory is locked only for the time it takes to read or write to it.

The complete DLL code

Having seen most of the non-trivial code for the DLL, it's time to see the whole picture. The executable DLL is called SERVER.DLL, and its source is contained primarily in the file SERVER.CPP, which is shown in listing 6:

```
#include <dos.h>
#include <afxwin.h>
#include <toolhelp.h>

#include "server.h"

// the structure of task-dependent memory blocks
struct LOCAL_VARIABLES {
  int a;
  char string [1000];
};

// MFC-derived class to handle the DLL code
class CServerDLL : public CWinApp {
  BOOL AllocateMemory(HTASK);

  // container to hold task-dependent data
  CMapPtrToPtr TaskTable;

public:

  void DeallocateMemory(HTASK);

  CServerDLL(const char* name) : CWinApp(name) {}
  HGLOBAL VariablesOf(HTASK);
};

// create a global object of type CServerDLL
CServerDLL NEAR server("server.dll");

// callback function, invoked when significant events
// occur for a Windows task
extern "C" BOOL FAR PASCAL _export CallBack(WORD eventType, DWORD)
{
  if (eventType != NFY_EXITTASK)
    return FALSE;

  // delete any memory allocated to the terminating task
  server.DeallocateMemory(GetCurrentTask() );
  return FALSE;
}

// given a task handle, allocate memory for the task,
BOOL CServerDLL::AllocateMemory(HTASK taskHandle)
{
  // see if this task has already been registered
  void* memoryHandle;
  if (TaskTable.Lookup( (void*) taskHandle, memoryHandle) )
    return TRUE;

  // the task hasn't been registered:
  // allocate a new block of memory for the task
  HGLOBAL hMemory =
      GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE | GMEM_ZEROINIT,
              sizeof(LOCAL_VARIABLES) );

  if (!hMemory) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, "Fatal Error",
             "SERVER.DLL could not successfully "
             "allocate memory for the current task.",
             MB_OK | MB_SYSTEMMODAL | MB_ICONSTOP);
    return FALSE;
  }

  // associate this memory handle with the task handle
  TaskTable [ (void*) taskHandle ] = (void*) hMemory;

  // install a callback for the task, so we can
  // be called when the task terminates
  NotifyRegister(taskHandle, CallBack, NF_NORMAL);
  return TRUE;
}
```

```
void CServerDLL::DeallocateMemory(HTASK task)
{
  // see if the terminating task was our client
  void* memoryHandle;
  if (!TaskTable.Lookup( (void*) task, memoryHandle) )
    return;

  // deallocate the task's memory
  GlobalFree( (HGLOBAL) _FP_OFF(memoryHandle) );

  // remove the task handle from the container
  TaskTable.RemoveKey( (void*) task);

  // kill the callback function
  NotifyUnRegister(task);
}

// return a handle to the memory allocated to a given task
HGLOBAL CServerDLL::VariablesOf(HTASK task)
{
  // see if this task has already been registered
  void* memoryHandle;
  if (TaskTable.Lookup( (void*) task, memoryHandle) )
    // yes - return a handle to its memory
    return (HGLOBAL) _FP_OFF(memoryHandle);

  else {
    // no - allocate new memory for the task, and return
    // a handle to it
    AllocateMemory(task);
    if (TaskTable.Lookup( (void*) task, memoryHandle) )
      return (HGLOBAL) _FP_OFF(memoryHandle);
    else
      // memory couldn't be allocated to the task: fatal error
      return NULL;
  }
}

// modify the task-dependent memory
void FAR PASCAL _export
SetA(int value)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // update the task data
  variables->a = value;
  GlobalUnlock(handle);
}

// read the task-dependent memory
int FAR PASCAL _export
GetA()
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return 0;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return 0;

  // return the task data
  int value = variables->a;
  GlobalUnlock(handle);
  return value;
}

// modify the task-dependent memory
void FAR PASCAL _export
SetString(LPSTR s)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // update the task data
  strncpy(variables->string, s, sizeof(variables->string) );
  variables->string [sizeof(variables->string)-1] = 0;
  GlobalUnlock(handle);
}

// read the task-dependent memory
void FAR PASCAL _export
GetString(LPSTR s, int size)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // return the task data
  strncpy(s, variables->string, size);
  s [size-1] = 0;
  GlobalUnlock(handle);
}
```

Listing 6 - SERVER.CPP, the file which contains the majority of the server DLL code.

There is another file in the server DLL project, called MAINFRM.CPP, that was created by AppWizard. It contains almost no code. After deleting the code dedicated to debugging, MAINFRM.CPP appears as in listing 7:

```
#include "stdafx.h"
#include "test.h"

#include "mainfrm.h"

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
      //{{AFX_MSG_MAP(CMainFrame)
            // No message maps
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

CMainFrame::CMainFrame() {}
CMainFrame::~CMainFrame() {}
```

Listing 7 - The AppWizard-generated support code for SERVER.DLL.

A sample client application

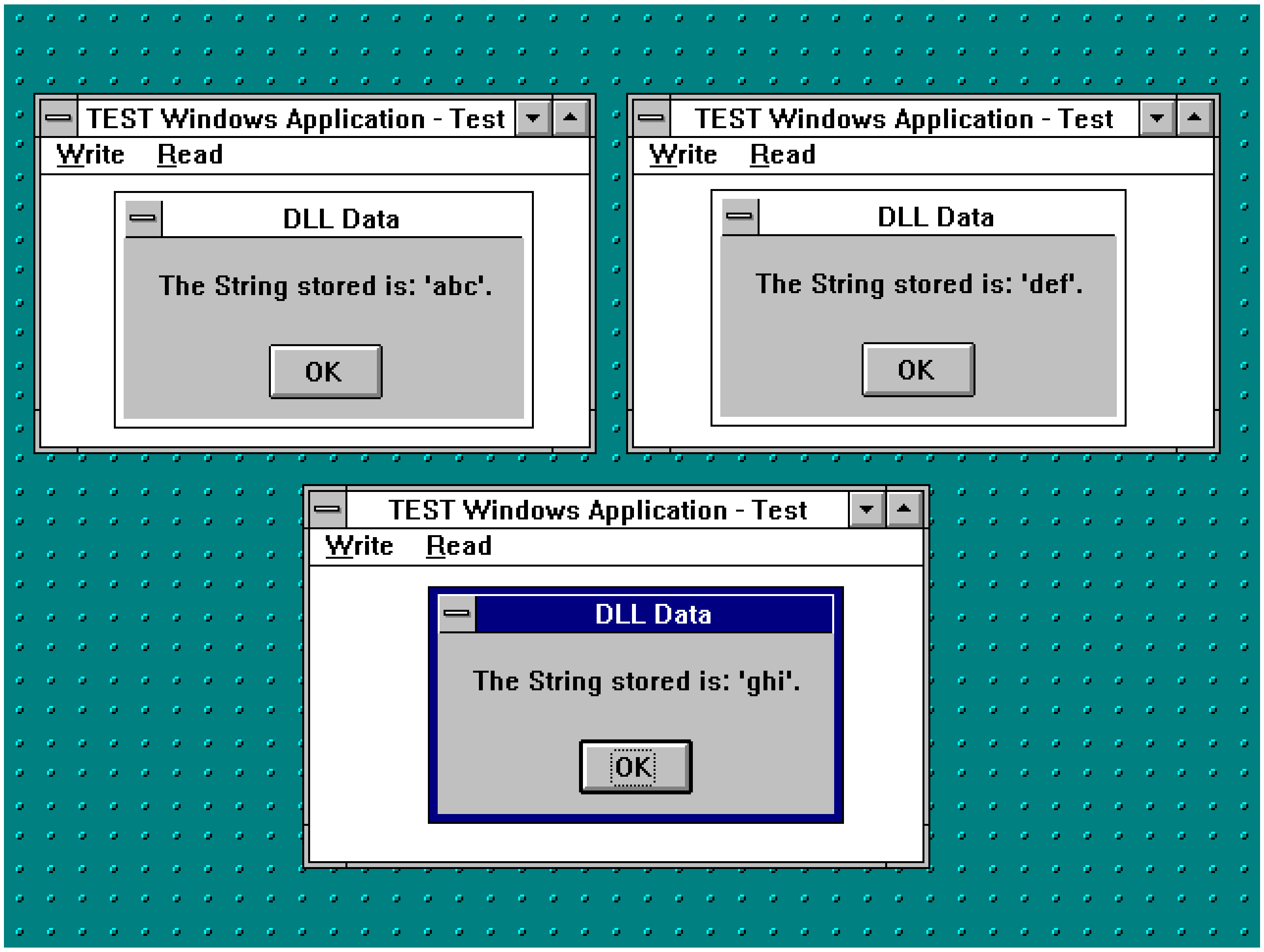
To test the server DLL developed in the previous section, I wrote a minimal Visual C++ application called TEST. Almost all the code was generated by AppWizard and ClassWizard. The application has a menu with two commands for reading and writing the server DLL data. The code that deals with the menu commands is in the file TESTVIEW.CPP, shown in listing 8:

```
// testview.cpp : implementation of the CTestView class

// testview.cpp : implementation of the CTestView class

#include "stdafx.h"
#include "test.h"

#include "testdoc.h"
#include "testview.h"
#include "..\\server.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////
/////
// CTestView

IMPLEMENT_DYNCREATE(CTestView, CView)

BEGIN_MESSAGE_MAP(CTestView, CView)
  //{{AFX_MSG_MAP(CTestView)
  ON_COMMAND(ID_WRITE_INTEGER_123, OnWriteInteger123)
  ON_COMMAND(ID_WRITE_INTEGER_456, OnWriteInteger456)
  ON_COMMAND(ID_WRITE_INTEGER_789, OnWriteInteger789)
  ON_COMMAND(ID_WRITE_STRING_ABC, OnWriteStringAbc)
  ON_COMMAND(ID_WRITE_STRING_DEF, OnWriteStringDef)
  ON_COMMAND(ID_WRITE_STRING_GHI, OnWriteStringGhi)
  ON_COMMAND(ID_READ_STRING, OnReadString)
  ON_COMMAND(ID_READ_INTEGER, OnReadInteger)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

CTestView::CTestView() { }

CTestView::~CTestView() { }

void CTestView::OnDraw(CDC* pDC)
```

```
{
  CTestDoc* pDoc = GetDocument();
}

/////////////////////////////////////////////////////////////////////////
/////
// CTestView message handlers

void CTestView::OnWriteInteger123()
{
  SetA(123);
}

void CTestView::OnWriteInteger456()
{
  SetA(456);
}

void CTestView::OnWriteInteger789()
{
  SetA(789);
}

void CTestView::OnWriteStringAbc()
{
  SetString("abc");
}

void CTestView::OnWriteStringDef()
{
  SetString("def");
}

void CTestView::OnWriteStringGhi()
{
  SetString("ghi");
}

void CTestView::OnReadString()
{
  char message [100];
  char string [20];
  GetString(string, sizeof string);
  sprintf(message, "The String stored is: '%s'.", string);
  MessageBox(message, "DLL Data");
}

void CTestView::OnReadInteger()
{
  char message [100];
  sprintf(message, "The integer stored is: '%d'.", GetA() );
  MessageBox(message, "DLL Data");
}
```

Listing 8 - TESTVIEW.CPP, a short MFC program that acts as a client application to the server DLL.

Using the **Write** menu command, TEST can store integer values or strings into the server memory. Using the **Read** menu, TEST displays the stored data in a message box. By running multiple instances of TEST, you can save different strings in the server for each client. Figure 4 shows 3 clients applications running, with messages boxes showing the string stored for each client.


TEST Windows Application - Test
Write
Read
DLL Data
OK
TEST Windows Application - Test
Write
Read
DLL Data
OK
TEST Windows Application - Test
Write
Read
DLL Data
OK

The String stored is: 'abc'.

The String stored is: 'def'.

The String stored is: 'ghi'.

Figure 4 - A session showing 3 client applications storing different data in the server DLL task-dependent memory.

The OWL Implementation

The OWL implementation I'll present here was created using Borland C++ 4.0 and OWL 2.0. The server DLL uses a dictionary to manage associations of task and memory handles -- just like the MFC server, but rather than using a container that manages `void` pointers, OWL uses a container class template. The dictionary class template is called `TDictionaryAsHashTable`, and can handle associations of arbitrary user types. The advantage of using templates is type-safe compilation. The compiler knows the types being handled, and no typecasting is ever needed with the items passed to and from the container template class. The drawback is a slight increase in complexity of the syntax. The use of a couple of `typedefs` minimizes the impact of the template syntax on the rest of the program. To declare a template class to handle associations of task and global memory handles, I used the `typedef`:

```
typedef TDDAssociation<HTASK, HGLOBAL> TaskData;
```

From this point on, the new type `TaskData` can be used just like a built-in type, and you can forget it is derived from a class template. There is one requirement that must be met to use your own data type with the template class `TDDAssociation`: there must be a global function called `HashValue()` that takes a parameter of your data type and returns an unsigned hash value. For type `HTASK`, the function must be declared like this:

```
unsigned HashValue(HTASK);
```

The value returned is used to store the association in a given slot of the hash table of a dictionary container. You don't necessarily have to return a hashed value, computed from your user data type. For example, if your `HashValue()` function returns the constant value 1, then all the user associations will be stored at the same entry in the dictionary's hash table, in an overflow linked list. The dictionary will still work, but will be less efficient in locating entries when there are many items in the container. For my OWL server example, the `HashValue()` function is defined like this:

```
unsigned HashValue(HTASK task) {return (unsigned) task;}
```

The task handle itself is returned as the hash value. Having defined a `HashValue()` function for `TaskData`, you can then use the `TaskData` association type to declare a dictionary template class, like this:

```
typedef TDictionaryAsHashTable<TaskData> Dictionary;
```

Now you can use the type `Dictionary` with the same ease as a non-template class -- but with all the advantages of type-safety. The server DLL code is primarily contained in the class `TServerDLL`. The class has a public data member of type `Dictionary` called `TaskTable`, declared like this:

```
Dictionary TaskTable;
```

The notation for instantiating a template-class container variable is the same as for ordinary scalar variables like `int` or `char`, so there is no obscure syntax to get in the way. The rest of class `TServerDLL` takes care of allocating and deallocating task-dependent memory. Class `TServerDLL` is declared in listing 9.

```
class TServerDLL {

  BOOL AllocateMemory(HTASK);
public:

  void DeallocateMemory(HTASK);

  // template-class types used to manage
  // tasks and task-dependent memory
  typedef TDDAssociation<HTASK, HGLOBAL> TaskData;
  typedef TDictionaryAsHashTable<TaskData> Dictionary;

  // container to hold task-dependent data
  Dictionary TaskTable;
  HGLOBAL VariablesOf(HTASK);
};
```

Listing 9 - The declaration of the OWL class `TServerDLL`.

The private member function `AllocateMemory(HTASK)` is called to allocate global memory for each new task that calls the server. The function is shown in listing 10:

```
BOOL TServerDLL::AllocateMemory(HTASK taskHandle)
{
  // see if this task has already been registered
  HGLOBAL memoryHandle;
  if (TaskTable.Find(TaskData(taskHandle, memoryHandle) ) )
    return TRUE;

  // the task hasn't been registered:
  // allocate a new block of memory for the task
  HGLOBAL hMemory =
      GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE | GMEM_ZEROINIT,
              sizeof(LOCAL_VARIABLES) );

  if (!hMemory) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, "Fatal Error",
             "SERVER.DLL could not successfully "
             "allocate memory for the current task.",
             MB_OK | MB_SYSTEMMODAL | MB_ICONSTOP);
    return FALSE;
  }

  // associate this memory handle with the task handle
  TaskTable.Add( TaskData(taskHandle, hMemory) );

  // install a callback for the task, so we can
  // be called when the task terminates
  NotifyRegister(taskHandle, CallBack, NF_NORMAL);
  return TRUE;
}
```

Listing 10 - Allocating shared global memory in the OWL server DLL.

`TServerDLL::AllocateMemory(HTASK)` uses the `TaskTable` container to see if a task has already been allocated memory. The function call:

```
TaskTable.Find(TaskData(taskHandle, memoryHandle) )
```

returns a pointer to a `TaskData` association if the given `taskHandle` is in the dictionary (i.e. the task is registered), and `NULL` otherwise. If the task has not been registered, memory is allocated to it using the statement:

```
HGLOBAL hMemory =
      GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE | GMEM_ZEROINIT,
              sizeof(LOCAL_VARIABLES) );
```

Notice the `GMEM_SHARE` flag used. By default, global memory allocated is assigned to the task that is currently running, which is not the server but the client task. DLLs are executed as extensions of application programs, and when applications terminate, Windows automatically frees all global memory allocated by them. Without the `GMEM_SHARE` flag, Windows would free the task-dependent memory allocated in the server when the client application terminates. By using the `GMEM_SHARE` flag, Windows will not free the memory automatically until the DLL is unloaded from memory, which will occur when all the client applications program terminate.

To add a task-memory association to the dictionary container, the statement:

```
TaskTable.Add( TaskData(taskHandle, hMemory) );
```

is sufficient. Notice that there is no typecasting whatsoever. Typecasting is not only ugly, but also dangerous, since it turns off the inherent type checking of the compiler. Every time you put a typecast into a program, you are adding a weakness, and could be taking your (program's) life into your own hands. I personally consider the use of typecasts to be a possible indication of poor OOP style. The use of class hierarchies, base class references/pointers and virtual functions usually render typecasts unnecessary.

Once memory is allocated to a task, a callback notification function is installed for it using the call:

```
NotifyRegister(taskHandle, CallBack, NF_NORMAL);
```

Details of the callback function were given in the MFC section. `TServerDLL` frees task-dependent memory with the member function `DeallocateMemory(HTASK)`, shown in listing 11.

```
void TServerDLL::DeallocateMemory(HTASK task)
{
  // see if the terminating task was our client
  HGLOBAL memoryHandle;
  TaskData* taskData = TaskTable.Find( (TaskData(task, memoryHandle) )
);
  if (!taskData)
    return;

  // deallocate the task's memory
  GlobalFree(taskData->Value() );

  // remove the task handle from the container
  TaskTable.Detach(*taskData);

  // kill the callback function
  NotifyUnRegister(task);
}
```

Listing 11 - The OWL server function to free task-dependent memory.

Given a task handle, `DeallocateMemory(HTASK)` checks the task dictionary to see if the task was registered, using the code:

```
TaskData* taskData = TaskTable.Find( (TaskData(task, memoryHandle) ) );
```

The argument of the function call is a temporary `TaskData` object, initialized with the parameters `task` and `memoryHandle`. The pointer returned by the member function `Find()` will be `NULL` if the given task is not found in the dictionary. If the task is found, a pointer to the association containing the task is returned, and used in the statement:

```
GlobalFree(taskData->Value() );
```

to free the task's memory. The expression `taskData->Value()` returns the global memory handle associated with a given task. As discussed earlier in the MFC example, the task dictionary contains associations, in which the task handles are the keys and the global memory handles are the associated values.

After a task's memory is freed, the task itself is removed from the dictionary with the code:

```
TaskTable.Detach(*taskData);
```

Since the dictionary owns the elements it contains, when an item is detached, it is also deleted. This is the default behavior for Borland containers. If you don't remember the default behavior, you can explicitly tell the dictionary to delete after detaching, using the statement:

```
TaskTable.Detach(*taskData, TShouldDelete::Delete);
```

`TShouldDelete` is a class used throughout the container library that controls how items are treated when a container goes out of scope or when an item is detached. After freeing the task memory, the notification callback is removed using the code:

```
NotifyUnRegister(task);
```

which was described earlier in the MFC section. Notification callbacks MUST be removed before the DLL is unloaded, otherwise the Windows kernel will RIP when ToolHelp attempts to send you a notification by calling code that is no longer in memory after the DLL is unloaded.

The last server function of interest is `TServerDLL::VariablesOf(HASK)`, which returns the handle of global memory allocated to a given task. Its code is shown in listing 12.

```
HGLOBAL TServerDLL::VariablesOf(HTASK task)
{
  // see if this task has already been registered
  HGLOBAL memoryHandle;
  TaskData* taskData = TaskTable.Find( (TaskData(task, memoryHandle) )
);
  if (taskData)
    // yes - return a handle to its memory
    return taskData->Value();

  else {
    // no - allocate new memory for the task, and return
    // a handle to it
    AllocateMemory(task);
    taskData = TaskTable.Find( (TaskData(task, memoryHandle) ) );
    if (taskData)
      return taskData->Value();
    else
      // memory couldn't be allocated to the task: fatal error
      return NULL;
  }
}
```

Listing 12 - The OWL function to retrieve the memory of a given task.

The function is equivalent to the code shown in the MFC example. The function `Find()` returns a pointer to a task/memory handle association. Using the association's `Value()` member function, the memory handle is retrieved. If a task wasn't registered, the function `AllocateMemory(HTASK)` is called to get memory for it and to register the task.

The complete DLL code

The code for the OWL server is contained in the file SERVER.CPP. Besides the class `TServerDLL`, the file contains the DLL access functions used by the client applications to access the server memory. The access functions `SetA(int)`, `GetA()`, `SetString(LPSTR)` and `GetString(LPSTR, int)` are virtually identical to the ones discussed in the MFC section. Listing 13 shows the code for SERVER.CPP:

```
#include <stdlib.h>
#include <dos.h>

#include <classlib\assoc.h>
#include <classlib\dict.h>

#include <toolhelp.h>
#include "server.h"
```

```
// A sample structure of task-dependent variables.
// A server can use any structure it wants
struct LOCAL_VARIABLES {
  int a;
  char aString [1000];
};

unsigned HashValue(HTASK task) {return (unsigned) task;}

class TServerDLL {

  BOOL AllocateMemory(HTASK);

public:

  void DeallocateMemory(HTASK);

  // template-class types used to manage
  // tasks and task-dependent memory
  typedef TDDAssociation<HTASK, HGLOBAL> TaskData;
  typedef TDictionaryAsHashTable<TaskData> Dictionary;

  // container to hold task-dependent data
  Dictionary TaskTable;
  HGLOBAL VariablesOf(HTASK);
};

// create a global object of type TServerDLL
TServerDLL server;

// callback function, invoked when significant events
// occur for a Windows task
extern "C" BOOL FAR PASCAL _export
CallBack(WORD eventType, DWORD)
{
  if (eventType != NFY_EXITTASK)
    return FALSE;

  server.DeallocateMemory(GetCurrentTask() );
  return FALSE;
}

// return a handle to the memory allocated to a given task
HGLOBAL TServerDLL::VariablesOf(HTASK task)
{
  // see if this task has already been registered
  HGLOBAL memoryHandle;
  TaskData* taskData = TaskTable.Find( (TaskData(task, memoryHandle) )
);
  if (taskData)
    // yes - return a handle to its memory
    return taskData->Value();

  else {
    // no - allocate new memory for the task, and return
    // a handle to it
    AllocateMemory(task);
    taskData = TaskTable.Find( (TaskData(task, memoryHandle) ) );
    if (taskData)
```

```
      return taskData->Value();
    else
      // memory couldn't be allocated to the task: fatal error
      return NULL;
  }
}

// given a task handle, allocate memory for the task,
BOOL TServerDLL::AllocateMemory(HTASK taskHandle)
{
  // see if this task has already been registered
  HGLOBAL memoryHandle;
  if (TaskTable.Find(TaskData(taskHandle, memoryHandle) ) )
    return TRUE;

  // the task hasn't been registered:
  // allocate a new block of memory for the task
  HGLOBAL hMemory =
      GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE | GMEM_ZEROINIT,
              sizeof(LOCAL_VARIABLES) );

  if (!hMemory) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, "Fatal Error",
             "SERVER.DLL could not successfully "
             "allocate memory for the current task.",
             MB_OK | MB_SYSTEMMODAL | MB_ICONSTOP);
    return FALSE;
  }

  // associate this memory handle with the task handle
  TaskTable.Add( TaskData(taskHandle, hMemory) );

  // install a callback for the task, so we can
  // be called when the task terminates
  NotifyRegister(taskHandle, CallBack, NF_NORMAL);
  return TRUE;
}

void TServerDLL::DeallocateMemory(HTASK task)
{
  // see if the terminating task was our client
  HGLOBAL memoryHandle;
  TaskData* taskData = TaskTable.Find( (TaskData(task, memoryHandle) )
);
  if (!taskData)
    return;

  // deallocate the task's memory
  GlobalFree(taskData->Value() );

  // remove the task handle from the container
  TaskTable.Detach(*taskData);

  // kill the callback function
  NotifyUnRegister(task);
}

// modify the task-dependent memory
void _export SetA(int value)
```

```
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // update the task data
  variables->a = value;
  GlobalUnlock(handle);
}

// read the task-dependent memory
int _export GetA()
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return 0;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return 0;

  // return the task data
  int value = variables->a;
  GlobalUnlock(handle);
  return value;
}

// modify the task-dependent memory
void _export SetString(LPSTR s)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // update the task data
  _fstrncpy(variables->aString, s, sizeof(variables->aString) );
  variables->aString [sizeof(variables->aString)-1] = 0;
  GlobalUnlock(handle);
}

// read the task-dependent memory
void _export GetString(LPSTR s, int size)
{
  // get the task data for the current task
  HGLOBAL handle = server.VariablesOf(GetCurrentTask() );
  if (!handle) return;
  LOCAL_VARIABLES* variables = (LOCAL_VARIABLES*) GlobalLock(handle);
  if (!variables) return;

  // return the task data
  _fstrncpy(s, variables->aString, size);
  s [size-1] = 0;
  GlobalUnlock(handle);
}

int FAR PASCAL LibMain(HINSTANCE, WORD, WORD, LPSTR)
{
  return 1;
}
```

Listing 13 - The complete OWL code for the server DLL.

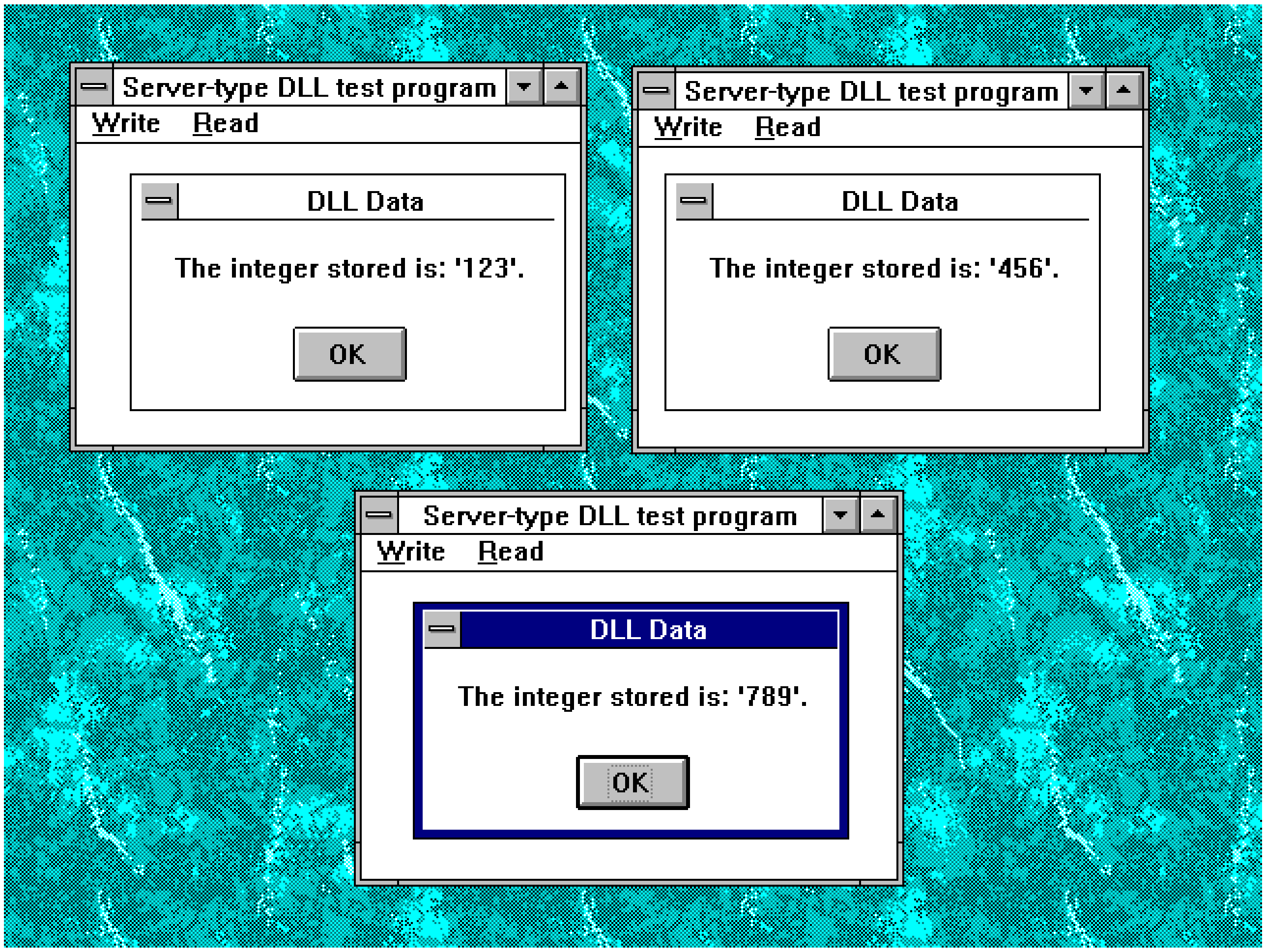
A sample client application

Although the OWL server DLL could be used with the Visual C++ MFC client application shown earlier, I developed a short OWL client application, just to show both sides of an OWL client server system. The program is called SERVTEST, and uses a message box to display the data stored in the server DLL. The program looks and feels the same as the one shown in the MFC section, and its code is shown in Listing 14:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <stdio.h>

#include "server.h"

// menu command IDs
const int
  ID_WRITEabc    = 101,
  ID_WRITEdef    = 102,
  ID_WRITEghi    = 103,
  ID_WRITE123    = 104,
  ID_WRITE456    = 105,
  ID_WRITE789    = 106,
  ID_READSTRING  = 107,
  ID_READINTEGER = 108;

class TWindowMain : public TWindow
{
public:

  TWindowMain() : TWindow(0, 0, 0) { }

  // message response functions
  void CmWrite123() {SetA(123);}
  void CmWrite456() {SetA(456);}
  void CmWrite789() {SetA(789);}

  void CmWriteabc() {SetString("abc");}
  void CmWritedef() {SetString("def");}
  void CmWriteghi() {SetString("ghi");}

  void CmReadInteger();
  void CmReadString();

  DECLARE_RESPONSE_TABLE(TWindowMain);
};

DEFINE_RESPONSE_TABLE1(TWindowMain, TWindow)
  EV_COMMAND(ID_WRITE123, CmWrite123),
  EV_COMMAND(ID_WRITE456, CmWrite456),
  EV_COMMAND(ID_WRITE789, CmWrite789),
  EV_COMMAND(ID_WRITEabc, CmWriteabc),
  EV_COMMAND(ID_WRITEdef, CmWritedef),
  EV_COMMAND(ID_WRITEghi, CmWriteghi),
  EV_COMMAND(ID_READINTEGER, CmReadInteger),
  EV_COMMAND(ID_READSTRING, CmReadString),
END_RESPONSE_TABLE;

void TWindowMain::CmReadInteger()
{
  char message [100];
  sprintf(message, "The integer stored is: '%d'.", GetA() );
  MessageBox(message, "DLL Data", MB_OK);
}

void TWindowMain::CmReadString()
{
  char message [100];
  char string [20];
  GetString(string, sizeof string);
  sprintf(message, "The String stored is: '%s'.", string);
  MessageBox(message, "DLL Data", MB_OK);
}


class TMyApplication: public TApplication
{
public:
  void InitMainWindow();
};

void TMyApplication::InitMainWindow()
{
  MainWindow = new
    TFrameWindow(0, "Server-type DLL test program",
             new TWindowMain);
  MainWindow->AssignMenu("MENU_MAIN");
}

int OwlMain (int , char* [])
{
  TMyApplication App;
  return App.Run();
}
```

Listing 14 - The code for a simple OWL client application program.

I ran 3 separate instances of SERVTEST, storing different data for each one in the server. Figure 5 shows the message boxes displayed by each instance, proving that the server indeed stores data on a per-client basis.



The integer stored is: '123'.

The integer stored is: '456'.

The integer stored is: '789'.

Figure 5 - Running 3 instances of SERVTEST together, showing how each instance has its own data in the server DLL memory.

Conclusion

Writing a DLL is not necessarily an arduous task, but does get somewhat complicated if you're developing a server. The use of C++ containers and other class libraries such as OWL or MFC can significantly reduce the amount of effort required, allowing you to concentrate more on higher level concepts than on the bits and bytes.