Chapter 27

Excel, C++ and Monte Carlo Integration

27.1 Introduction and Objectives

In this chapter we discuss how to integrate our code with other libraries and applications, in particular Excel. We create applications in which Excel is used as the front-end to Monte Carlo applications. We show how to create the software for user input, output, processing and communication between Excel and C++. We consider this chapter to be important because Excel is a popular application in finance and it is used on a daily basis by quantitative analysts and traders in many of the investment banks throughout the world.

There are two main techniques that allow us to write C++ applications that interoperate with Excel and we speak of Excel addins this context:

- **XLL addins** This is the oldest form of add-in. These addins are written in C using the Microsoft XLL application-programming interface. The code is compiled to a dynamic link library (DLL) having the extension .xll. They are useful for the creation of Excel worksheet functions or functions that we call using menus.

  The advantage in this case is that XLL is efficient and it works with all versions of Excel from Excel 95 upwards. To create XLL functionality we use the xlw framework. Using this framework it is easy to create C++ worksheet functions. It even generates usable interfaces to transform your already running C++ functions into functions that work within the Excel application. The framework can be downloaded at [xlw.sourceforge.net](http://xlw.sourceforge.net).

- **COM addins** These were introduced in Excel 2000 and are based on a generic COM (Component Object Model) interface for addins. All Office products use this interface as well as the Visual Studio and VBA development environments.

  Some remarks need to be made concerning this kind of addin. First, having created a COM addin you must register it in the registry for each host application (for example, Excel or Word) that it supports. This allows the host application to find the addin. In general terms, the registry is a hierarchical data store that holds many different kinds of data, for example DLLs and their file locations. Second, COM addins cannot be used to create Excel worksheet functions, at least not directly. If you wish to do so, then you should call the
CHAPTER 27. EXCEL, C++ AND MONTE CARLO INTEGRATION

COM addin function through an XLA (the VBA way of creating addins) addin worksheet function.

- **Automation addins** This option has been available since the introduction of Excel XP. It uses COM objects whose public functions are used as worksheet functions. Menus are not supported. An automation addin is always loaded on demand.

We examine some of the above options in detail. We provide a step-by-step account for creating an addin and we give several examples to show how to create an addin. In particular, we discuss worksheet functions for library-related functionality as well as complete applications that are started from within Excel.

We provide several complete applications of Excel interfacing on the CD.

27.2 Integrating Applications and Excel

In Duffy 2004 we introduced the topic of Excel and C++ integration. In particular, we traced the steps that are needed in order to create Automation and COM addins. In this chapter we extend the presentation by applying the results to the problem of integrating C++ code for the Monte Carlo method with Excel. Furthermore, we discuss some of the more theoretical foundations of the Active Template Library (ATL). This is Microsoft’s software framework for generating C++/COM code. It allows developers to write code with having to know the internals of COM. In fact, it uses macros to generate code stubs that the developer can subsequently change to suit his or her needs. Some of the features that ATL offers are:

- Class wrappers for a number of important data types in COM.
- Classes that provide implementations of a number of basic COM interfaces.
- Classes that manage COM servers and server lifecycle management.
- Wizards that generate stub code that the developer can edit to suit the current application needs.

The generated code can be intimidating when first encountered but it becomes easier to understand once we know what its intent is. This chapter discusses the underlying design of ATL to smoothen the transition to writing ATL-based applications for the Monte Carlo method.

27.3 ATL Architecture

In this section we give background information on what ATL is and how it is used.

27.3.1 Interfaces and Data Structures

COM objects implement interfaces and they communicate by data exchange. Microsoft has defined many standard interfaces that the developer can implement to create applications. Our interest lies in creating Excel-based applications and for this reason we concentrate on a few special interfaces. For the sake of completeness, we mention some of the supported interfaces for the following features:
27.3. ATL ARCHITECTURE

- Persistence: the ability to save an object’s state or properties to a permanent medium; the interfaces have methods for reading from and writing to a medium. The base interface is called IPersist and two derived interfaces are called IPersistFile and IPersistStream that allow us to interact with disk files and simple streams, respectively. Some applications may wish to load and save data and these interfaces are used for this purpose.

- Dispatching interfaces using Automation; Automation allows an application to interact with a COM server using a single standard interface called IDispatch. We shall deal with this interface in more detail when we create worksheet functions in Excel. Automation focuses on run-time type.

- IDTExtensibility2; this important interface hosts event notifications, such as when addins are loaded, unloaded and updated. This interface has a number of methods that we discuss in detail in section 27.3.3. We pay special attention to implementing the methods when we connect to and disconnect from Excel.

- IUnknown; this is the interface that all COM objects must implement. It has methods that allow us to determine if a COM object implements a given interface and methods for reference counting (these two methods ensure that the lifetime of objects are properly managed). As ATL programmer, you do not have to write the declarations of these methods because this is automatically taken care of by ATL.

- Connection points; we know that a COM object implements one or more interfaces. In this sense we say that the object has a number of provides interfaces. We now consider the reverse situation, namely the modelling of an object’s requires interfaces. In this case the connection point mechanism allows an object to expose its capability to call one or more specified interfaces. A connection has two parts and we can distinguish two roles; the first role is called the source or the connection point and the second role is called the sink because this is the object that receives the call and hence implements the interface. In general, a given source object may have connections to multiple sink objects.

We now give an introduction to the most important data types and data structures that are to be found in ATL. It is useful to know what they mean and our main concern is defining the data structures associated with an interface’s methods, for example its return type and its input and output parameters:

a) HRESULT; return type of most methods. This is a 32-bit value that is divided into three different fields as shown in Figure 27.1. You really only need to know the value of the most significant bit. This bit determines whether the call succeeded (or failed). The possible return codes are ($S_{-}$ corresponds to success, $E_{-}$ for failure):

- $E_{\text{ABORT}}$: the operation was aborted because of an unspecified error.
- $E_{\text{ACCESSDENIED}}$: a general access-denied error.
- $E_{\text{FAIL}}$: an unspecified failure has occurred.
- $E_{\text{HANDLE}}$: an invalid handle was used.
- $E_{\text{INVALIDARG}}$: one or more arguments are invalid.
- $E_{\text{NOINTERFACE}}$: the QueryInterface method of the interface IUnknown did not recognize the requested interface. The interface is not supported.
— **E\_NOTIMPL**: the method is not implemented.

— **E\_OUTOFMEMORY**: the method failed to allocate necessary memory.

— **E\_PENDING**: the data necessary to complete the operation is not yet available.

— **E\_POINTER**: an invalid pointer was used.

— **E\_UNEXPECTED**: a catastrophic failure occurred.

— **S\_FALSE**: the method succeeded and returned the boolean value **FALSE**.

— **S\_OK**: the method succeeded. If a boolean return value is expected, the returned value is **TRUE**.

We can check the return type in code by calling the **SUCCEEDED** and **FAILED** macros as the following examples show:

```
HRESULT hr = SetStrings ();
if (FAILED (hr))
    return hr;

HRESULT hr2 = pPropBag->Read ();
if (SUCCEEDED(hr2))
{ // code
}
```

It is advisable to use these macros when testing a method’s return value instead of comparing `hr` against the raw values; however, it is allowable to use these values as return types in code, for example:

```
HRESULT FinalConstruct ()
{
    return S\_OK;
}
```
b) **VARIANT;** ATL supports many kinds of data types that can be passed as parameters in methods. Each specific data type has its own discriminator and this allows the marshaler to deal with the data. In C++, it is similar to a `void*` type. We shall have little use for this data type in this chapter but it is useful that you know what it is.

c) **SAFEARRAY;** this is a self-describing array type that holds the size and type of each element in it as well as the number of elements and the bounds of the array. This type was created in order to allow arrays of data to be passed between processes. You will see some examples of use type in generated code later but the good news is that you can ignore it. We note that a **SAFEARRAY** can be copied to a **VARIANT**.

d) String data types: passing strings between processes poses problems in general. The main problem is that strings do not have a predetermined size. The first string type that we are interested in is **BSTR**, known as basic string or binary string. This is a pointer to a wide character string used by Automation data manipulation functions. In fact, **BSTR** is a typedef for **LPWSTR** Automation data. Finally, you may see some example of strings of type **LPSTR** in generated code.

### 27.3.2 The Structure of an ATL Project and its Classes

When developing Automation and COM addins we need to create two main classes. The first class is the embodiment of the ATL server and it has the same name as that of the ATL project that contains all relevant code. This class has functions for registering and unregistering the COM object in the Windows registry. The second class is the actual COM object where all the application code resides: in particular, it has functionality for:

- Communication with Excel: when connecting to Excel we add a menuitem to the Tools menu and when disconnecting from Excel we remove the menuitem from the Tools menu.

- ATL generates much of the code that is needed by the COM object.

We now discuss the main code blocks in the above two classes. In a later section we assemble the code in one place. We concentrate on code that we, as developers modify while we ignore unmodified generated code. The code for registering and unregistering the COM addin in the Windows registry uses two static member functions developed by the authors:

```c
STDAPI DllRegisterServer(void)
{
    // Register this COM addin for Excel
    COMUtils::RegisterCOMAddin("Excel", pszProgID,
                                pszFriendlyName, 3);

    // registers object, typelib and all interfaces in typelib
    HRESULT hr = _AtlModule.DllRegisterServer();
    return hr;
}
```

We remark that the addin can be loaded in different ways (in the above case we load in the most usual way, namely it is loaded at application startup time, hence the parameter '3'). Other possibilities are:

- **Disconnect** (value 0), addin is not loaded.
• *Connected* (value 1), addin is loaded.

• *Bootload* (value 2), load addin on application startup.

• *DemandLoad* (value 8), load addin only when requested by user.

• *ConnectFirstTime* (value 16), load addin only once (on next startup).

The code for unregistering the addin is:

```c++
// DllUnregisterServer - Removes entries from the system registry
STDAPI DllUnregisterServer(void)
{
    // Unregister this COM addin
    COMUtils::UnRegisterCOMAddin("Excel", pszProgID);

    HRESULT hr = _AtlModule.DllUnregisterServer();
    return hr;
}
```

In these cases we define strings for the ProgID (programmable ID) that the registry needs and for a user-friendly name for the addin that can be seen as one of the entries in the list of addins when Excel is running:

```c++
LPSTR pszProgID = "MyProj.MyComp";
LPSTR pszFriendlyName = "ATL for Chapter 27";
```

We mention the notation used in the two above member functions, in particular the use of the shorthand notation: `#define STDAPI EXTERN_C HRESULT STDAPICALLTYPE` 

In other words, this defines an API function that returns a `HRESULT`.

We now discuss the application code. The class for this implements the `IDTExtensibility2` interface. The project environment generates the stubs for the methods in this interface. We must modify these methods to let them return the value and we modify the code for connecting to and disconnecting from Excel. To this end, we show the definition of this class and its members:

```c++
#include "ExcelImports.cpp"    // All Excel-related information
#include "COMUtils.hpp"       // Utilities (e.g. registration, conversion, menus and buttons)

class ATL_NO_VTABLE CMyComp :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyComp, &CLSID_MyComp>,
    public IDispatchImpl<IMyComp, &IID_IMyComp, &LIBID_MyProjLib,
                                /*wMajor=*/ 1, /*wMinor=*/ 0>,
    public IDispatchImpl<_IDTExtensibility2,
                                &__uuidof(_IDTExtensibility2),
                                &LIBID_AddInDesignerObjects, /* wMajor = */ 1>
private:
    // The Excel instance the addin communicates with
    Excel::_ApplicationPtr m_xl;

    // The menu item added by the addin
    Office::_CommandBarButtonPtr m_menuItem;

public:
};
```
Most of this code is generated with the exception of the private members, that are defined by the developer. The macro ATL_NO_VTABLE wraps some specific code that optimises the C++ code in the application. It is not relevant to our discussion, as long as you have an idea of what is going on.

The next step is to define how the COM interfaces in the object can be exposed to clients. To this end, we define the so-called COM interface map that is automatically generated and is defined by the macros:

```
BEGIN_COM_MAP(CMyComp)
  COM_INTERFACE_ENTRY(IMyComp)
  COM_INTERFACE_ENTRY2(IDispatch, IMyComp)
  COM_INTERFACE_ENTRY(_IDTExtensibility2)
END_COM_MAP()
```

We must also be able to couple external events (in this application this will be the act of clicking a button in the Tools menu) and in this case we define a sink map:

```
BEGIN_SINK_MAP(CmyComponent)
  SINK_ENTRY_EX(/*nID =*/ 1, __uuidof(Office::_CommandBarButtonEvents), /*dispid*/ 1, OnButtonClick) // Callback function
END_SINK_MAP()
```

We then need to register the event sink with the menu item. As we shall see, this takes place on connection with Excel by calling the function DispEventAdvise() while we unregister the event handler when the addin is unloaded by calling the function DispEventUnadvise(). An example of use is found in section 27.6.2.

The actual code for the event handler is represented as a method in the COM object:

```
// Handle click event
void __stdcall CmyComponent::OnButtonClick(Office::_CommandBarButtonPtr Ctrl, VARIANT_BOOL * CancelDefault)
{
  m_xl->GetRange("a1:b2")->Value2="Add-In called";
}
```

In this case the body of the code is very simple but it can be replaced by any application code that you wish, for example code for Monte Carlo simulation or for the finite difference method. It is advisable to create separate header and code files in the interest of maintainability.

### 27.3.3 Implementing the Methods in IDTExtensibility2

This is arguably the most important interface in the application and it is by means of its methods that we can communicate with Excel. It hosts the notifications that happen to addins. The main events are:

- **OnConnection()** occurs whenever an addin is loaded.
- **OnDisconnection()** occurs whenever an addin is unloaded.
- **OnAddInsUpdate()** occurs whenever an addin is loaded or unloaded.
- **OnBeginShutdown()** occurs whenever the host application shuts down while an addin is running.
• OnStartupComplete() occurs whenever an addin loads, which is set to load when the host application (that is, Excel) starts.

We are interested in the first two methods. First, the method for connection sets the reference to Excel and creates a menu item in the Tools menu. It also registers the event sink with the menu item:

```cpp
STDMETHOD(OnConnection)(LPDISPATCH Application, ext_ConnectMode ConnectMode,
                        LPDISPATCH AddInInst, SAFEARRAY * * custom)
{
    // Store reference to the Excel host application
    // Exit if host application is not Excel
    m_xl = Application;
    if (m_xl==NULL)
    {
        return S_OK;
    }

    Office::COMAddInPtr cai=AddInInst;
    if(cai!=NULL)
    {
        void* id;
        this->QueryInterface(IID_IDispatch, &id);
        cai->put_Object((IDispatch*)id);

        m_menuItem = COMUtils::AddMenuItem(m_xl, cai,
                                           CComBSTR("Tools"),CComBSTR("My MC ATL"), CComBSTR("MCComp"));

        return DispEventAdvise(m_menuItem);
    }

    return S_OK;
}
```

The menu for disconnecting contains code to unregister the event handler when the addin is unloaded and it removes the menu item from the Tools menu:

```cpp
STDMETHOD(OnDisconnection)(ext_DisconnectMode RemoveMode,
                         SAFEARRAY * * custom)
{
    if (m_menuItem!=NULL)
    {
        DispEventUnadvise(m_menuItem);

        COMUtils::RemoveMenuItem(m_xl, RemoveMode,
                             CComBSTR("Tools"),CComBSTR("My MC ATL"));
    }
}
```
The other methods have an empty body and return the standard success result:

```c
HRESULT STDMETHODCALLTYPE OnAddInsUpdate(SAFEARRAY ** custom)
{
    return S_OK;
}

HRESULT STDMETHODCALLTYPE OnStartupComplete(SAFEARRAY ** custom)
{
    return S_OK;
}

HRESULT STDMETHODCALLTYPE OnBeginShutdown(SAFEARRAY ** custom)
{
    return S_OK;
}
```

A final remark; STDMETHODCALLTYPE is a macro that wraps the _stdcall calling convention, the virtual specifier and the HRESULT return type.

## 27.4 Creating my First ATL Project: the Steps

In this section we enumerate the steps that you take in order to create a COM addin for Excel. We have already discussed this issue in Duffy 2004. The same steps are used for the creation of an Automation addin but in the latter case that we do not need to implement IDTExtensibility2 interface. We use the wizards that the Microsoft IDE delivers and the main challenge is knowing which buttons to press and where to insert your own code in the generated source files (it is also important to know why you are doing it):

1. Create an ATL DLL project and give it a name (in the settings dialog this project should be not attributed). The name of the project determines the names of created source files and the ProgID.
2. Select the project and add an ATL simple object to it (you select the project by navigating on the Solution Explorer, right mouse clicking the project and selecting the Add menuitem). Give the ATL simple object a name.
3. (this step for COM addin only) In Class View you add an Implementation Interface called IDTExtensibility2. Code is generated in the header file corresponding to the ATL object; in particular, you should change the return type of the four methods in IDTExtensibility2 to S_OK.
4. We now edit the project header file and insert code to register and unregister the addin:

```c
// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
{
    // Register this COM addin for Excel
    COMUtils::RegisterCOMAddin("Excel", pszProgID,
```
pszFriendlyName, 2);

    // registers object, typelib and all interfaces in typelib
    HRESULT hr = _AtlModule.DllRegisterServer();
    return hr;
}

// DllUnregisterServer - Removes entries from the system registry
STDAPi DllUnregisterServer(void)
{
    // Unregister this COM addin
    COMUtils::UnRegisterCOMAddin("Excel", pszProgID);
    HRESULT hr = _AtlModule.DllUnregisterServer();
    return hr;
}

Furthermore, you need to define two strings in the project file as follows:

LPSTR pszProgID = "MyProj.MyComp";
LPSTR pszFriendlyName = "ATL for Chapter 27";

5. At this stage you can build the project to check for syntax errors. You can start Excel and you will see your addin in the list but you cannot run it because 1) it is not connected to Excel and 2) we have not yet defined event sinks and event handlers

6. We now modify the code for the ATL simple object. In particular, you need to carry out the following steps:
   – Define how your object interacts with Excel:
     
     private:
     // The Excel instance the addin communicates with
     Excel::ApplicationPtr m_xl;

     // The menu item added by the sdd-in
     Office::CommandBarButtonPtr m_menuItem;

     – Code the connect and disconnect functions. Create a sink map. **Insert your application-specific code in the event handler.**

7. Build the project and you are finished if you have no errors. Start Excel and test your addin.

We now give some tips and guidelines in relation to this topic in order to help you discover errors and resolve those irritating error messages:

a) Make sure you include COMUtil.hpp and COMUtil.cpp in the relevant files (for the project and the ATL simple object) in the Visual Studio project. Make sure that you resolve compiler and linker errors at each stage of the above steps in the project.

b) You must import the Excel type libraries for your current version of Excel (see the file ExcelImports.cpp). In particular software operates with the English version of Excel. If you wish to use it with another language (for example, German) you need to change the following, admittedly hard-coded name Sheet1 to the appropriately-named name in your version of Excel (in the file ExcelDriver.cpp). The sourcecode that uses this hard-coded name is:
try {
    // Initialize COM Runtime Libraries.
    CoInitialize(NULL);

    // Start excel application.
    xl.CreateInstance(L"Excel.Application");
    xl->Workbooks->Add((long) Excel::xlWorksheet);

    // Rename "Sheet1" to "Chart Data".
    Excel::_WorkbookPtr pWorkbook = xl->ActiveWorkbook;
    Excel::_WorksheetPtr pSheet = 
        pWorkbook->Worksheets->GetItem("Sheet1");
    pSheet->Name = "Chart Data";
}

catch (_com_error & error )
{
    ThrowAsString(error);
}

27.5 Creating Automation Addins in Excel

Section 3 of this chapter discussed how to create a COM addin in Excel. In this section we discuss how to create an Automation addin for Excel.

Automation allows a client to communicate with a component. Instead of a component providing several custom interfaces specific to the services that it offers we shall see that the component offers those services through the single standard interface called IDispatch. Automation makes it possible for interpretive languages to access COM components that implement the IDispatch interface. Automation builds on COM. An Automation server is a COM component that implements the IDispatch interface and an Automation controller is a COM client that communicates with this server through this interface. In other words, the IDispatch interface is the proxy or intermediate between the controller and server. The requests from the controller are dispatched indirectly to the server.

Almost all services that COM provides can also be achieved using IDispatch. Furthermore, Automation addins are easier to create than COM addins.

27.5.1 Automation Interfaces

The IDispatch interface allows a component to expose any number of methods and properties. It is an ordinary COM interface and it uses a vtable that contains pointers to its methods. The distinguishing feature, however is that it has a method called Invoke and this method is used to call other methods. The developer also specify a dispatch interface (or dispinterface) and this specifies those methods that may be invoked. Each method in this interface has a dispatch identifier (DISPID). This identifier is usually an integer and functions as an index to some function in the function table. We represent the situation graphically in Figure 27.2. In fact, this is an array of function pointers. We shall see how to create an addin based on the IDispatch interface.

c) Clean the Registry on a regular basis, especially in a development environment.

d) In your Project Properties settings, do not use precompiled headers and make sure that Unicode settings are off.
interface. In particular, we shall use Automation to create worksheet functions in Excel. For example, here is an example of generated component code with two simple methods (notice the two DISPID values):

interface IComp777 : IDispatch
{
    [id(1), helpstring("method myFunc")]
    HRESULT myFunc(
        [out,retval] DOUBLE* myArg);
    [id(2), helpstring("method myFunc2")]
    HRESULT myFunc2(
        [out,retval] LONG* arg);
};

There are four methods in IDispatch:

- **GetTypeInfo**: if the object has a type library, then this method returns a pointer to the typeinfo object describing the dispinterface. In this way, a client can learn everything about a server, including its methods and associated parameters.
- **GetTypeInfoCount**: this method indicates if an object returns type information at run time, in other words if the method GetTypeInfo will return useful information.
- **GetIDOfNames**: this method allows the client to pass the name of a method (in the form of a string) that is then converted to an integer ID that we subsequently use in Invoke.
- **Invoke**: we use this method to invoke all methods in all dispinterfaces.

We note that an object can support multiple IDispatch instances as well as ordinary COM interfaces as shown in Figure 27.3.

Using dispinterfaces introduces a number of disadvantages. The main ones are:

1) invoking a method using a dispinterface is slower than invoking the same method in a vtable,
2) the process of packing parameters into variants is time-consuming and slows down performance,
3) it takes more time in C++ to program a method using dispinterfaces than to code the equivalent vtable.
In order to avoid these drawbacks it is possible to create objects that implement dual interfaces. In other words, such objects’ methods can be called through IDispatch or using vtable calls.

27.6 Useful Utilities and Interoperability Projects

Excel is a very popular tool and we can use it in a variety of ways:

- As an input device (by cells and ranges).
- As an output device (cells, charts).
- As a computational engine.

In this case we are interested in selecting data from Excel, computing some quantities using that data and presenting those results. The kinds of applications that we are interested in are:

- Mathematical structures: We created a number of data structure that we use for numerical work, for example vectors, matrices and tensors. Furthermore, we wish to integrate these structures with Excel such as two-way data exchange with VARIANT and SAFEARRAY.

- Integration with Standard Template Library (STL): We wish to integrate the most important STL containers (such as vector, set and map) with Excel.


- Solvers (for example, matrix algebra, eigenvalue calculations): useful building blocks such as linear and nonlinear solvers.

- Applications (PDE, Monte Carlo and other financial applications): this is the user level where we work with integrated Monte Carlo applications.
27.6.1 27.7 Test Case: A COM Addin and complete Code

We give the most important source code for COM addins and Automation addins in this section. You can find the complete code on the accompanying CD.

These source files represent the *finished product* as it were and you can compare your code with these to check that you have not forgotten something.

27.6.2 27.7.1 COM Addin

There are two files, namely the `.hpp` file (which you can ignore because it contains generated code that need not/may not be modified) and the `.cpp` file that contains code for registration and unregistration:

```cpp
#include "stdafx.h"
#include "resource.h"
#include "MyATLProj.h"
#include "COMUtils.hpp"

// Developer-supplied strings
LPSTR pszProgID = "MyATLProj.MyATLComp";
LPSTR pszFriendlyName = "Chapter 27 MC book";

class CMyATLProjModule : public CAtlDllModuleT< CMyATLProjModule >
{
public :
    DECLARE_LIBID(LIBID_MyATLProjLib)
    DECLARE_REGISTRY_APPID_RESOURCEID(IDR_MYATLPROJ,
        "{C4CA01E7-957B-46E8-B71F-273228EB4015}"
    );
};
CMyATLProjModule _AtlModule;

#ifdef _MANAGED
#pragma managed(push, off)
#endif
// DLL Entry Point
extern "C" BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    hInstance;
    return _AtlModule.DllMain(dwReason, lpReserved);
}

#ifdef _MANAGED
#pragma managed(pop)
#endif

// Used to determine whether the DLL can be unloaded by OLE
STDAPI DllCanUnloadNow(void)
{

```
return _AtlModule.DllCanUnloadNow();
}

// Returns a class factory to create an object of the requested type
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    return _AtlModule.DllGetClassObject(rclsid, riid, ppv);
}

// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
{
    // Register this COM addin for Excel
    COMUtils::RegisterCOMAddin("Excel", pszProgID,
                              pszFriendlyName, 2);
    // registers object, typelib and all interfaces in typelib
    HRESULT hr = _AtlModule.DllRegisterServer();
    return hr;
}

// DllUnregisterServer - Removes entries from the system registry
STDAPI DllUnregisterServer(void)
{
    // Unregister this COM addin
    COMUtils::UnRegisterCOMAddin("Excel", pszProgID);
    HRESULT hr = _AtlModule.DllUnregisterServer();
    return hr;
}

For the ATL simple object, there are two files, namely the .cpp file (which you can ignore because it contains generated code and is empty) and the .hpp file that contains code for connecting to and disconnecting from Excel (we shall use default inline code for convenience):

// MyATLComp.h : Declaration of the CMyATLComp
#pragma once
#include "resource.h" // main symbols
#include "MyATLProj.h"
#include "ExcelImports.cpp"
#include "COMUtils.hpp"
#if defined(_WIN32_WCE) && !defined(_CE_DCOM) && !defined(_CE_ALLOW_SINGLE_THREADED_OBJECTS_IN_MTA)
#endif

// CMyATLComp
class ATL_NO_VTABLE CMyATLComp :
    public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CMyATLComp, &CLSID_MyATLComp>,
public IDispatchImpl<IMyATLComp, &IID_IMyATLComp,
 &LIBID_MyATLProjLib, /*wMajor ==*/ 1, /*wMinor ==*/ 0>,
public IDispatchImpl<_IDTExtensibility2,
 &__uuidof(_IDTExtensibility2), &LIBID_AddInDesignerObjects,1>
{
 public:
 CMyATLComp()
 {
 }

 DECLARE_REGISTRY_RESOURCEID(IDR_MYATLCOMP)

 BEGIN_COM_MAP(CMyATLComp)
  COM_INTERFACE_ENTRY(IMyATLComp)
  COM_INTERFACE_ENTRY2(IDispatch, IMyATLComp)
  COM_INTERFACE_ENTRY(_IDTExtensibility2)
 END_COM_MAP()

 DECLARE_PROTECT_FINAL_CONSTRUCT()

 HRESULT FinalConstruct()
 {
  return S_OK;
 }

 void FinalRelease()
 {
 }

 private:
 // The Excel instance the addin communicates with
 Excel::_ApplicationPtr m_xl;

 // The menu item added by the addin
 Office::_CommandBarButtonPtr m_menuItem;

 public:
 STDMETHOD(OnConnection)(LPDISPATCH Application,
 ext_ConnectMode ConnectMode,
 LPDISPATCH AddInInst, SAFEARRAY ** custom)
 {

 // Store reference to the Excel host application
 // Exit if host application is not excel
 m_xl = Application;
 if (m_xl==NULL)
  {
   return S_OK;
  }
Office::COMAddInPtr cai=AddInInst;

if(cai!=NULL)
{
    void* id;
    this->QueryInterface(IID_IDispatch, &id);
    cai->put_Object((IDispatch*)id);

    m_menuItem = COMUtils::AddMenuItem(m_xl, cai,
            CComBSTR("Tools"), CComBSTR("My MC ATL2"),
            CComBSTR("MCComp2"));

    return DispEventAdvise(m_menuItem);
}

return S_OK;
}

STDMETHOD(OnDisconnection)(ext_DisconnectMode RemoveMode,
SAFEARRAY * * custom)
{
    if (m_menuItem!=NULL)
    {
        DispEventUnadvise(m_menuItem);

        COMUtils::RemoveMenuItem(m_xl, RemoveMode,
            CComBSTR("Tools"), CComBSTR("My MC ATL2"));

    }

    return S_OK;
}

STDMETHOD(OnAddInsUpdate)(SAFEARRAY * * custom)
{
    return S_OK;
}

STDMETHOD(OnStartupComplete)(SAFEARRAY * * custom)
{
    return S_OK;
}

STDMETHOD(OnBeginShutdown)(SAFEARRAY * * custom)
{
    return S_OK;
}

OBJECT_ENTRY_AUTO(__uuidof(MyATLComp), CMyATLComp)
27.6.3 Automation Addin

We have created an addin whose name contains text (in this case '999') to accentuate the differences between system-generated names and names thought up by the developer. The source file for the project is:

```cpp
// AUTO999.cpp : Implementation of DLL Exports.
#include "stdafx.h"
#include "resource.h"
#include "AUTO999.h"

class CAUTO999Module : public CAtlDllModuleT< CAUTO999Module >
{
public :
    DECLARE_LIBID(LIBID_AUTO999Lib)
    DECLARE_REGISTRY_APPID_RESOURCEID(IDR_AUTO999,
        "{11673BE6-472B-42D9-806D-9166EC516F44"
    );

    CAUTO999Module _AtlModule;

#ifdef _MANAGED
    #pragma managed(push, off)
#endif

    // DLL Entry Point
    extern "C" BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
    {
        hInstance;
        return _AtlModule.DllMain(dwReason, lpReserved);
    }

#ifdef _MANAGED
    #pragma managed(pop)
#endif

    // Used to determine whether the DLL can be unloaded by OLE
    STDAPI DllCanUnloadNow(void)
    {
        return _AtlModule.DllCanUnloadNow();
    }

    // Returns a class factory to create an object of the requested type
    STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
    {
        return _AtlModule.DllGetClassObject(rclsid, riid, ppv);
    }
```
// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
{
    // registers object, typelib and all interfaces in typelib
    HRESULT hr = _AtlModule.DllRegisterServer();
    return hr;
}

// DllUnregisterServer - Removes entries from the system registry
STDAPI DllUnregisterServer(void)
{
    HRESULT hr = _AtlModule.DllUnregisterServer();
    return hr;
}

The source code for the header file for the simple ATL object contains one method in this
case (you may define any number of methods in this object). In this case we just return a number
but in your case you can add code that is relevant to your application:

// Comp999.h : Declaration of the CComp999
#pragma once
#include "resource.h"  // main symbols
#include "AUTO999.h"

#ifndef WIN32_WCE || defined(_CE_DCOM) || !defined(_CE_ALLOW_SINGLE_THREADED_OBJECTS_IN_MTA)
#endif

// CComp999
class ATL_NO_VTABLE CComp999 :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CComp999, &CLSID_Comp999>,
    public IDispatchImpl<IComp999, &IID_IComp999, &LIBID_AUTO999Lib, /*wMajor=*/ 1, /*wMinor=*/ 0>
{
    public:
        CComp999()
        {
            DECLARE_REGISTRY_RESOURCEID(IDR_COMP999)

            BEGIN_COM_MAP(CComp999)
                COM_INTERFACE_ENTRY(IComp999)
                COM_INTERFACE_ENTRY(IDispatch)
            END_COM_MAP()

            BEGIN_COM_MAP(CComp999)
                COM_INTERFACE_ENTRY(IComp999)
            END_COM_MAP()
        }
    }
CHAPTER 27. EXCEL, C++ AND MONTE CARLO INTEGRATION

DECLARE_PROTECT_FINAL_CONSTRUCT()

HRESULT FinalConstruct()
{
    return S_OK;
}

void FinalRelease()
{
}

public:
    STDMETHOD(myPI999)(void);
    STDMETHOD(func999)(DOUBLE* myarg);
};

OBJECT_ENTRY_AUTO(__uuidof(Comp999), CComp999)

The code file for the ATL simple object is given by:

// Comp999.cpp : Implementation of CComp999
#include "stdafx.h"
#include "Comp999.h"

// CComp999
STDMETHODIMP CComp999::myPI999(void)
{
    // TODO: Add your implementation code here

    return S_OK;
}

STDMETHODIMP CComp999::func999(DOUBLE* myarg)
{
    // TODO: Add your implementation code here

    *myarg = 1.42;
    return S_OK;
}

27.6.4 The Utilities Class

We have provided a number of useful functions to ease the pain of writing Excel applications in C++. We include the following functionality (that was included in Duffy 2004 and Duffy 2006) for registration/unregistration, interfacing to and from Excel and creating buttons and menu items in the Excel user interface:

#ifdef COMUtils_hpp
#define COMUtils_hpp

#include "stdafx.h"

#endif
#include "atsafe.h"
#include "excelimports.cpp"
#include "Vector.cpp"

class COMUtils
{
public:
    // Register a COM addin
    static HRESULT RegisterCOMAddin(LPCSTR pszOfficeApp,
        LPCSTR pszProgID, LPCSTR pszFriendlyName,
        DWORD dwStartupContext);

    // Unregister a COM addin
    static HRESULT UnRegisterCOMAddin(LPCSTR pszOfficeApp,
        LPCSTR pszProgID);

    // Set a key in the registry
    static BOOL SetKeyAndValue(HKEY hKeyRoot, LPCSTR pszPath,
        LPCSTR pszSubkey, LPCSTR pszValue);

    // Recursively delete a key in the registry
    static LONG RecursiveDeleteKey(HKEY hKeyParent,
        LPCSTR pszKeyChild);

    // Convert variant to vector with doubles
    static Vector<DOUBLE> ExcelRangeToVector(VARIANT* range);

    // Convert Excel range to vector with doubles
    static Vector<DOUBLE> ExcelRangeToVector(Excel::RangePtr pRange);

    // Add menu item to Excel
    static Office::_CommandBarButtonPtr
        AddMenuItem(Excel::_ApplicationPtr xl,
            Office::COMAddInPtr addin,
            CComBSTR menuName,
            CComBSTR menuItemCaption,
            CComBSTR menuItemKey);

    // Remove menu item from Excel
    static void RemoveMenuItem(Excel::_ApplicationPtr xl,
        ext_DisconnectMode removeMode,
        CComBSTR menuName,
        CComBSTR menuItemCaption);

};

#endif COMUtils_hpp

The complete sourcecode is to be found on the CD.
27.7 Summary and Conclusions

We have given a detailed account of how to create C++ applications that interface with Excel and we paid particular attention to COM addins and Automation addins. The latter addins are used for worksheet functions where response time is important while the former addins are more suitable for compute-intensive applications.

We described the steps that you need to execute to create an addin and we gave extensive examples of the C++ code that implements them. We have provided several non-trivial examples and applications on the CD.

27.8 Exercises and Projects

1. (Curve Fitting and Cubic Splines)
   The objective of this exercise is to integrate C++ code for cubic spline interpolation (as developed in Duffy 2006) with Excel. We wish to write an Automation addin to calculate the value of the spline for a specific value of the independent variable. Design the program in Excel based on the following input:

   - An array of x values.
   - An array of y values (these two arrays x,y must have the same size).
   - An x value whose spline value we wish to calculate.
   - An array of x values whose spline values we wish to calculate.
   - The kind of boundary conditions used (Duffy 2006, Press 2002).

   The class interface that we need is given by:

   ```cpp
   enum CubicSplineBC {SecondDeriv, FirstDeriv};

   class CubicSplineInterpolator
   {
   private:
   // ...
   private:
   // Private member functions

   public:
   CubicSplineInterpolator(const Vector<double, long> xarr,
                          const Vector<double, long> yarr,
                          CubicSplineBC BCTYPE, double alpha = 0.0,
                          double beta = 0.0);

   // Find the interpolated valued at a value x
   double Solve(double xvar) const;

   // Create the interpolated curve, MEMBER DATA AS ABSCISSAE
   Vector<double, long> Curve() const;
   };
   ```

   Test your addin using examples from your own work.
2. (My first Automation addin)
In this exercise we use the exact solution of the Black Scholes formula for put and call options, including the calculation of call and put prices and the sensitivities such as delta, gamma and theta. The full C++ code is on the CD.

Answer the following questions:

a) design the user interface for this problem, in particular the input and output aspects; you will need to define option parameters (such as expiry and strike price), give them values and determine in which cells they are defined. Furthermore, the output cells contain the put and call option price.

b) Create an Automation addin for the design in part a). Use STL to hold your data structures.

c) We now extend the addin to support calculation of the option sensitivities: delta, gamma and vega. Print these values in Excel.

d) Compare the solution with the solution if you had to do the exercise in VBA. Do you notice performance improvements and what can you say about code maintainability in these two languages?

3. (COM addin), ****
In chapter 0 we developed a simple Monte Carlo framework to price one-factor plain options. We now wish to integrate this code with Excel using a COM addin. In this case we wish to use the same data as in exercise 4 (namely, option parameters). The C++ code when using the console for input and output was:

```c++
int main()
{
    // Create the basic SDE (Context class)
    double T = 30.0;
    Range<double> range (0.0, T);
    double initialCondition = 100.0;

    // Discrete stuff
    long N = 100;
    cout << "Number of subintervals: ";
    cin >> N;

    // Tell the Builder what kinds of SDE and FDM Types you want
    //SDEType {A, B, C, D};
    //FDMType {Euler, PC, CN, MIL, SIE, IE, DerivFree, FRKito, Fit};
    FDMTypeDBuilder fdmBuilder(FDMTypeDBuilder::D, FDMTypeDBuilder::PC, N, range,
                                initialCondition, drift, diffusion,
                                diffusionDerivative);

    // V2 mediator stuff
    long NSimulations = 50000;
    cout << "Number of simulations: ";
    cin >> NSimulations;
```
double r = 0.08;
cout << "Starting up the mediator...\n";

try
{
    MCTypeDMediator mediator(fdmBuilder, NSimulations, r, T,
                             myPayOffFunction);

    cout << "Final Price: " << mediator.price() << endl;
}

catch(string& exception)
{
    // CN, IE or IE cannot be used with NL/NL SDEs

    cout << exception << endl;
    cout << "Press any key to stop\n";
    int yy; cin >> yy;
    exit(1);
}

return 0;

The objective of this exercise is to get the same program running while using Excel as the input/output medium. Use the steps in section 27.3 to guide you.