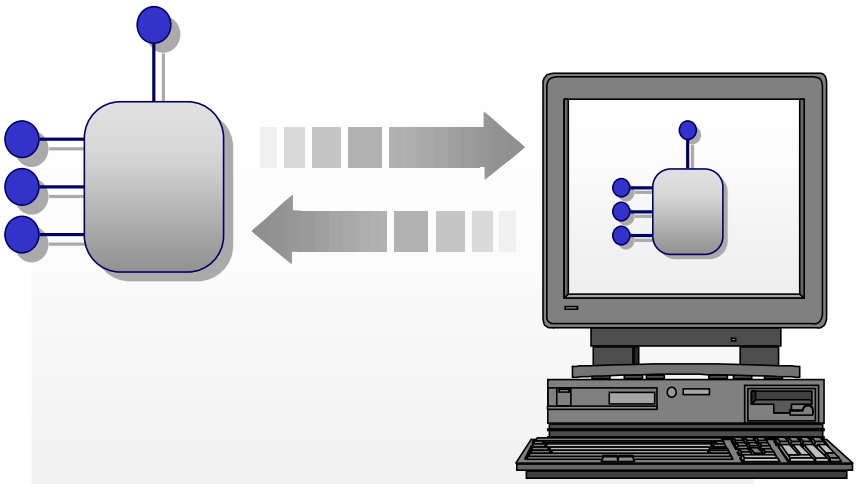
COM EXE Servers

QA-IQ



COM Programming

191

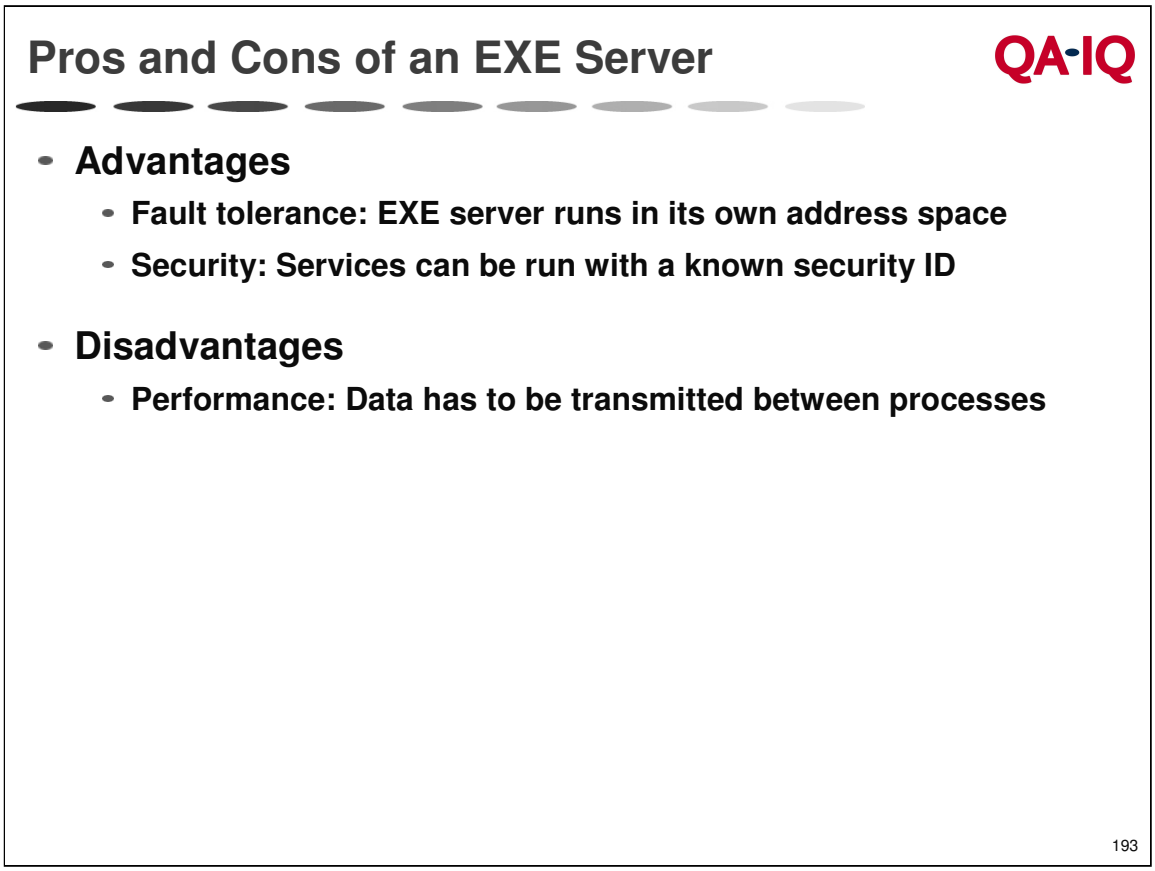
Chapter Overview

QA-IQ

- Objectives
  - Explain the need for marshalling when crossing the process boundary
  - Examine a COM EXE server
- Chapter content
  - Proxies and stubs
  - Standard marshaling
  - Type-library marshalling
  - Marshalling issues and IDL
  - EXE servers
  - Accessing a server remotely
- Practical content
  - Build a simple EXE server
- Summary

192

This chapter is our first look at writing out of process COM servers. Up to now, we have focused on implementing DLL servers, but there are times when we need the additional protection, scalability and fault tolerance that we gain from running our COM objects in a different process or on another machine.



COM EXE Servers run in their own address space. This means that a failure with the client will not affect the server, and a failure of the server will not affect the client. This vastly improves the fault tolerance levels of the system.

Additionally, processes (especially services) can be launched with known security credentials, offering improved security options over in-process servers.

However, these benefits come at a cost - namely performance, because data has to be transmitted (or *marshalled*) between the processes. Obviously, crossing a process boundary will also necessitate thread switching, and may lead to an increase in code swapping from disk, both of which will again decrease performance.

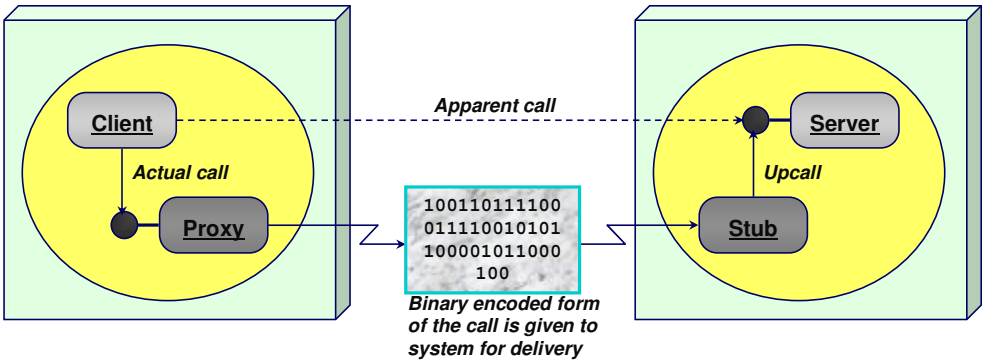
Whilst we will look at how to write EXE servers in this chapter, you should think long and hard before writing an EXE server. One of the most common reasons for writing EXE servers is to manage state within an application. Managing state in this way might be acceptable for simple applications, but it is simply not scalable or robust enough in most situations. So how can you get the robustness of out of process servers, plus scalability as well?

The answer to this is to write DLL servers, but to host them using COM+ on Windows 2000. We will take a look at COM+ later in the course, but suffice to say that the current opinion on writing COM servers is to always write DLL servers, and use surrogate hosting techniques to run them out of process when necessary. It should be noted that EXE servers cannot be configured to run within COM+.

Crossing the Process Boundary

QA-IQ

- COM object can reside in another process
- On the client side, a proxy is responsible for representing the server object
- On the server side, a stub is responsible for delivering a call to the object



194

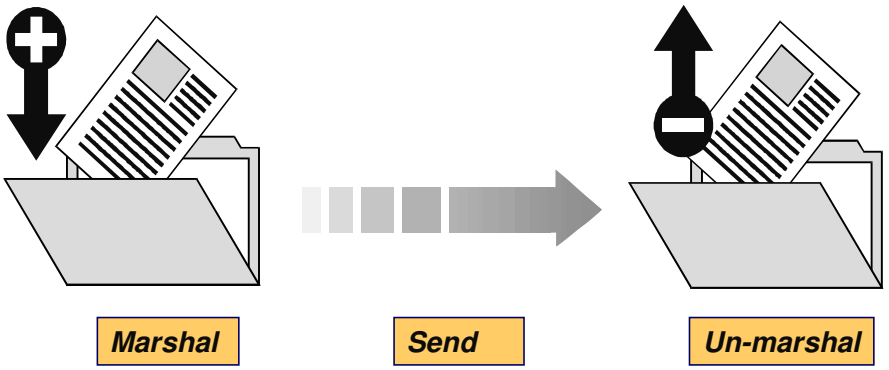
The *proxy* on the client side is responsible for speaking to the runtime system. It translates the high level call into an appropriate representation – Network Data Representation (NDR) in the case of DCOM – that is then transmitted across the transport, e.g. a TCP/IP network. This process of translation is known as *marshalling*.

On the receiving supplier or server side, the runtime library reconstructs the high level call from the marshalled binary stream. The operation call is then delivered to the destination object via a *stub* (effectively the inverse of the proxy on the caller side). This process is known as *unmarshalling*.

Marshalling

QA-IQ

- **Marshalling is the translation of call parameters into an intermediate binary representation for transmission**
- **Un-marshalling is the translation back from the on-the-wire representation to call parameters**



195

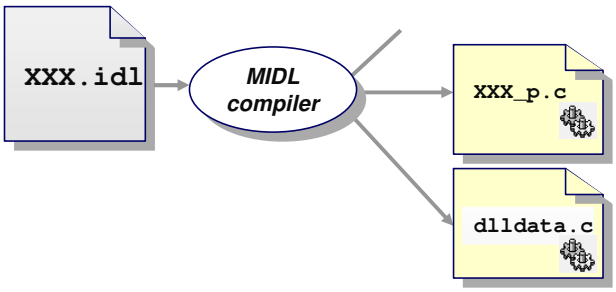
Operation calls are synchronous, which means that the caller waits for the callee to complete the call and return its results. This is handled by the runtime components linked in with the application which reconstruct the call on the server against the server object.

For data transfer: *in* parameters must be marshalled across the network and unmarshalled on the receiving side; *in, out* parameters, *out* parameters and return values must be marshalled back and unmarshalled for the benefit of the caller.

Using Standard Marshalling

QA-IQ

- Define object interface(s) in IDL
- Compile using MIDL compiler
  - Generates proxy/stub DLL source code
- Compile and link proxy/stub DLL
  - Define REGISTER_PROXY_DLL
- Register this DLL under HKCR\Interface\<IID>
  - Call DllRegisterServer() on the proxy/stub DLL, e.g. using regsvr32.exe



Interface proxy file: contains proxy and stub code for interfaces in IDL file

Proxy DLL file: contains code required to put proxy and stub code into a DLL (e.g. DllMain)

196

Standard marshalling is implemented by MIDL-generated proxy/stub code, and is therefore designed to marshal a custom interface that is defined in an IDL file. This should not be confused with the marshalling of *standard* COM interfaces (such as `IErrorInfo`), which is implemented by `ole32.dll`.

As explained in the *Introduction to IDL* chapter, the MIDL compiler generates a number of files. These include the interface proxy file (`XXX_p.c`), which contains the proxies and stubs for all the interfaces defined in the IDL file, and the proxy DLL file (`dlldata.c`), which contains the necessary functions and other information to build and register a proxy/stub DLL.

Before you can build the proxy/stub DLL, you need to provide a DLL definition file that lists the exported functions, e.g.

```
LIBRARY      "SimpleAccountPS"

DESCRIPTION  'Proxy/Stub DLL'

EXPORTS
    DllGetClassObject       @1 PRIVATE
    DllCanUnloadNow         @2 PRIVATE
    GetProxyDllInfo         @3 PRIVATE
    DllRegisterServer       @4 PRIVATE
    DllUnregisterServer     @5 PRIVATE
```

To make the proxy/stub DLL self-registering, make sure that you define `REGISTER_PROXY_DLL` when compiling the source modules. Also, when linking, you'll need to specify the following libraries: `kernel32.lib`, `rpcndr.lib`, `rpcns4.lib`, `rpcrt4.lib` and `uuid.lib`.

Note that if you use the *ATL COM App Wizard* in Visual C++, it automatically generates the definition file and makefile necessary to build the proxy/stub DLL.

Using Type-Library Marshalling

QA-IQ

- Define interface in IDL
  - Must use Automation-compatible types
  - Tag with [oleautomation] or [dual] attribute
- Generate type library using MIDL compiler
- Register type library
  - Under HKCR\TypeLib\<LIBID>\1.0\0\Win32
- Associate type library with interface
  - LIBID under HKCR\Interface\<IID>\TypeLib
- Set proxy/stub DLL for interface
  - Set HKCR\Interface\IID\ProxyStubClsid32 to [00020424-0000-0000-C0000-000000000046]
  - This GUID corresponds to the PSOAInterface class, which points to oleaut32.DLL
- No need to install MIDL-generated proxy/stub DLL

197

Whereas standard marshaling is specific to a custom interface, *type-library* marshaling is implemented by a generic proxy/stub DLL (`oleaut.dll`) known as the *Automation marshaller*. As its name suggests, this proxy/sub DLL implements the marshaller used by Automation; in other words, it is designed to marshal the standard interface `IDispatch`. However, its use is not limited to Automation, because it can marshal *any* custom interface that uses only Automation-compatible types, and for which a type library is available. (For a list of Automation-compatible types, please refer to the *Automation* chapter.) Advantages of type-library marshalling over standard marshalling are that a proxy/stub DLL does not have to be built, and does not have to be installed and registered on client and server machines; the Automation marshaller, as implemented in `oleaut.dll`, is automatically available on all 32-bit Windows machines. On the other hand, type-library marshalling is less efficient than standard marshalling due to the extra step required to look up the type-library information.

To use type-library marshalling with a particular non-standard interface, you must make sure that the interface is either a dual interface (i.e. it derives from `IDispatch`), or is a custom interface that is marked as Automation-compatible. The latter is achieved using the `[oleautomation]` attribute, as shown in the following example:

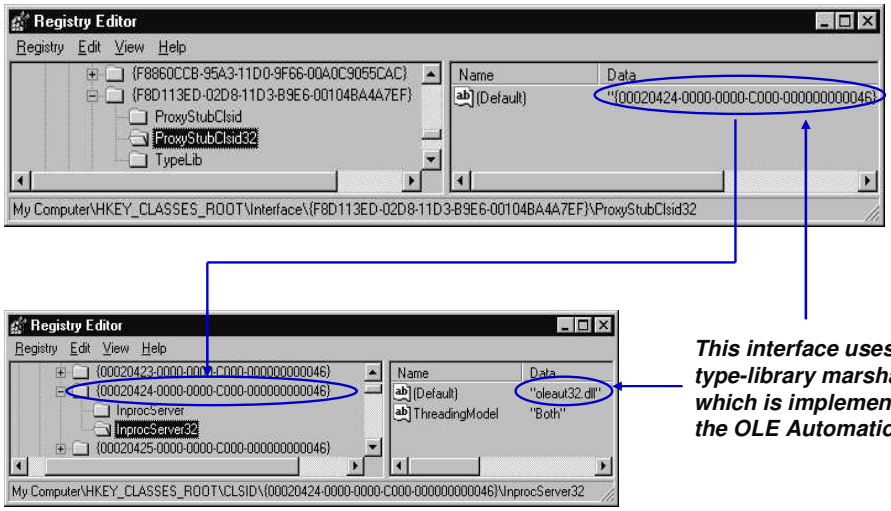
```
[
  object, uuid(80FE63ED-056A-11D3-9C99-000000000000),
  oleautomation,
]
interface IAccount : IUnknown {
  ...
};
```

You'll also need to generate a type library, which must be registered on both client and server machines. As shown on the next slide, `oleaut.dll` must also be registered as the interface's proxy/stub DLL on both client and server machines.

Registering a Proxy/Stub DLL

QA-IQ

- **COM locates the interface proxy/stub DLL via the HKCR/Interface/<IID> registry key**
  - **Will contain a ProxyStubClsid32 entry**



198

Whichever method of marshalling is used, the appropriate proxy/stub DLL must be registered on both client and server machines. As shown on the slide, several entries are added to the Registry.

First an IID key is added under `HKEY_CLASSES_ROOT\Interface`. Under this key, we can see several entries, but the most important is the `ProxyStubClsid32` key, which links the interface to a CLSID. When we look at this CLSID under `HKEY_CLASSES_ROOT\CLSID`, we find it has an `InprocServer32` entry that specifies the name of the proxy/stub DLL. In the example on the slide, the specified proxy/stub DLL is `oleaut.dll`, which implements the Automation marshaller.

Marshalling Issues and IDL

QA-IQ

- Ensure data is passed in most efficient manner
  - Use [in] and [out] attributes *
- Only support NULL and alias pointers where necessary
  - Be aware of the [pointer_default()] attribute
- Support varying length strings
  - Marshaller needs to know how much data to copy
  - Use [string] attribute or BSTRs *
- Support varying size arrays
  - Marshaller needs to know how much data to copy
  - Use [size_is] and [length_is] attributes
- Support interface pointers
  - Use [iid_is] attribute

* See *Introduction to IDL* chapter

199

One of the motivations for defining an RPC or COM interface in IDL is to marshal function calls in the most efficient manner. This is particularly important with remote objects in order to reduce network traffic. Therefore, when defining a COM interface in IDL, various attributes must be used to allow the MIDL compiler to generate efficient proxies and stubs.

In the *Introduction to IDL* chapter, we covered some of the basic attributes such as `[in]`, `[out]` and `[string]`. In the next few slides, we will discuss some more advanced attributes such as `[pointer_default]`, `[size_is]`, `[length_is]` and `[iid_is]`. We will also discuss the use of the COM task memory allocator.

Pointer Types and pointer_default()

QA-IQ

- **pointer_default() for use with embedded pointers**

Pointer Type	Reference	Unique	Full
IDL attribute	`ref`	`unique`	`ptr`
Allows `NULL` pointers	*No*	*Yes*	*Yes*
Allows *alias* pointers	*No*	*No*	*Yes*

```
// SimpleAccount.idl ----------------
import "unknwn.idl";
[
  object, uuid(7902EAB1-E763-11d2-9C56-000000000000),
  pointer_default(unique)
]
interface IAccount : IUnknown
{
  HRESULT GetBalance([out, retval] float *pBalance);
  ...
};
```

200

A function parameter of a basic data type, such as a `float`, is marshaled by simply copying its value from the address space of the client process to the address space of the server process. However, if a function parameter is a pointer, the data pointed to by this pointer must be copied from one address space to another. Therefore, it is fairly obvious that additional information must be provided to tell the marshaller how many bytes to copy. In addition, to optimise the proxy/stub code, the MIDL compiler needs to know whether `NULL` is a valid value for a pointer (if not, there is no point in marshalling it), and whether alias pointers are allowed.

There are three types of pointer in COM:

A *reference* pointer must always contain a valid address, i.e., it cannot be `NULL`. This means that the marshaller can check for a `NULL` value before it's sent across the wire. Furthermore, two reference pointers that are passed in the same call must not contain the same address, i.e., they can't be duplicated or *aliased*. This means that the marshaller won't check for alias pointers in one address space, and the unmarshaller doesn't have to ensure that the alias pointers point to the same memory address in the destination address space.

A *unique* pointer can be `NULL`, but cannot be aliased with another pointer in the same call.

A *full* pointer is equivalent to a 'C' pointer. The pointer can be `NULL`, and it can be aliased with another pointer in the same call.

By default, all top level pointers are *ref* pointers. There is a `pointer_default()` attribute that can be set for the interface, but this only affects embedded pointers or pointers that are returned by functions (although remember that all functions should return `HRESULT`). An *embedded* pointer might be found as members of `structs`, including, for example, node pointers within linked lists.

Arrays

QA-IQ

- If array size is fixed at design time, specify its length

```
short rgs[7] = {8, 13, 27, 34, 35, 41, 24};
HRESULT SetNumbers([in] short rgs[7]);
```

- Otherwise, use [size_is] attribute
  - Specifies maximum capacity of array

```
HRESULT SetNumbers2([in] long cMax,
                    [in, size_is(cMax)] short *rgs);
```

- With out parameters, also use [length_is] attribute
  - Specifies valid length of array

```
HRESULT GetWinningNumbers([in] long cMax,
                          [out] long *pcActual,
                          [out, size_is(cMax),
                          length_is(*pcActual)] short *rgs);
```

201

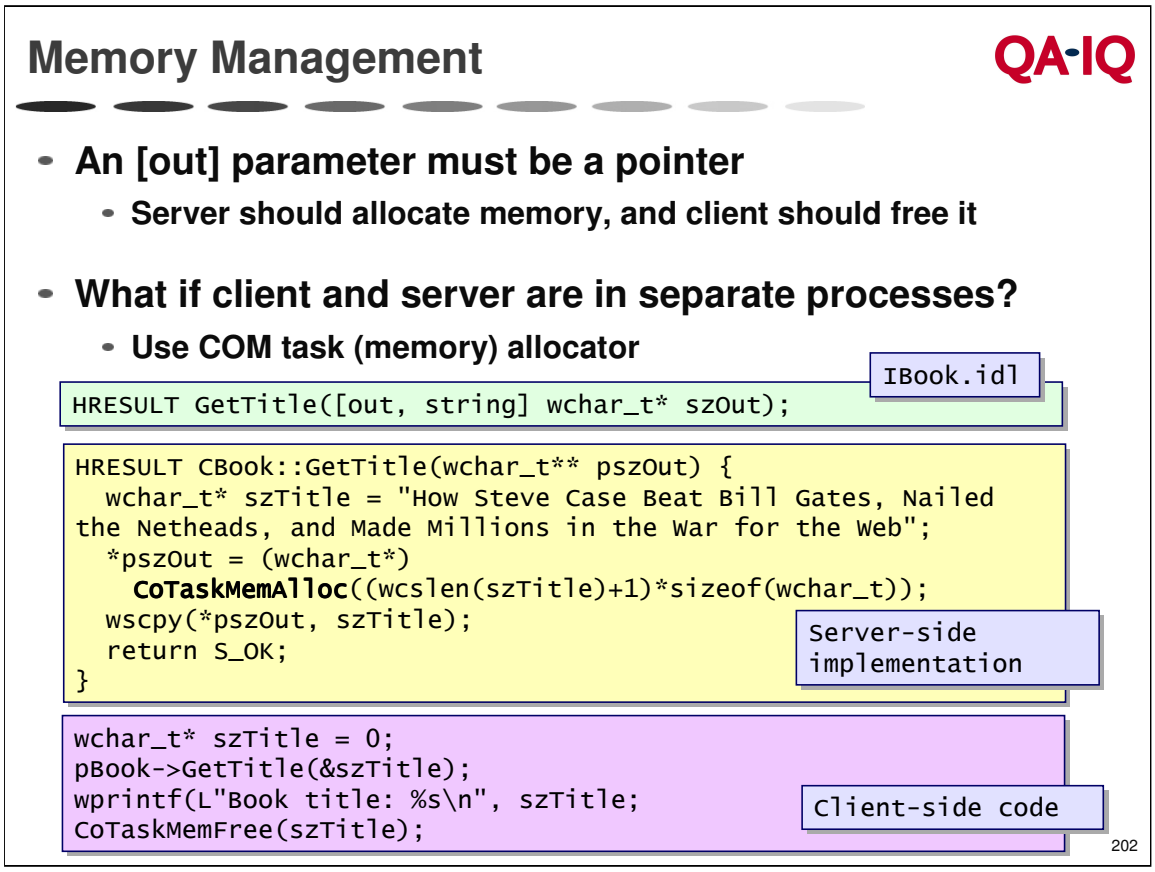
Without an attribute or qualifier of some kind, an IDL pointer is assumed to point to a single byte. In the case of arrays, there are several ways in which the size of the array can be specified.

If the size of an array is known at design time, the size of array can be hard-coded using the C array syntax as shown in the first example on the slide. This type of array is known as a *fixed array*.

If the size of an array is unknown at design time, the `[size_is()]` attribute can be used to specify the maximum capacity of the array at run time. This type of array is known as a *conformant array*. Note that the parameter specified to `size_is()` can be a single value or an arithmetic expression. Typically, as shown in the second example on the slide, the size of the array will also need to be passed as a function parameter, and this parameter will also be specified to `size_is()`.

Conformant arrays work well when used as `[in]` parameters, but they are not always efficient when used as `[out]` parameters. The problem is that the callee may not use the full capacity of the array, in which case unused bytes will be returned to the caller. If this is the case, the `[first_is()]` and `[length_is()]` attributes can be used to mark the valid length of the array, which will be the actual number of bytes sent across the wire. Such an array is called a *varying array*. The third example on the slide shows a typical use of the `[length_is()]` attribute. Note that the `[size_is()]` attribute is still required so that the marshaller knows what buffer size to allocate.

Note that `size_is()` has a special syntax to support arrays of arrays - check the MSDN documentation for more information.

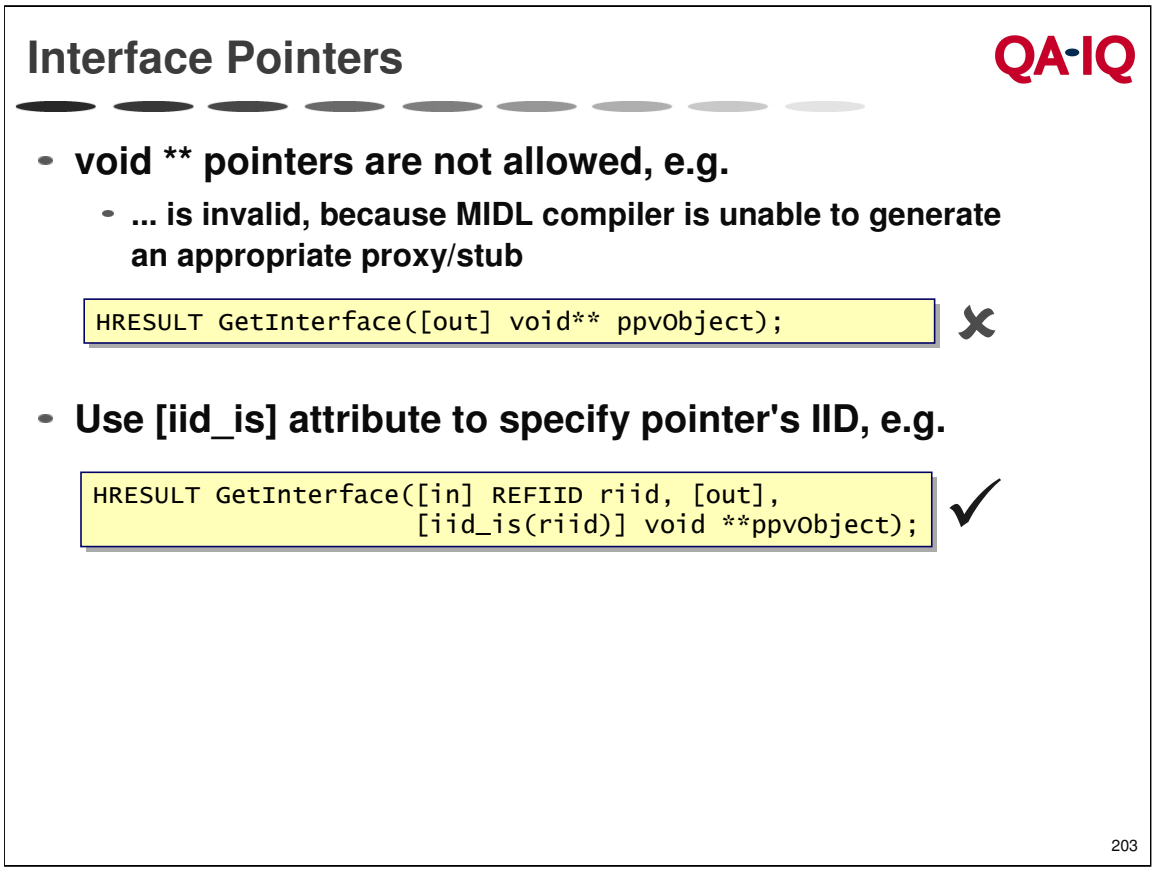


As previously mentioned, an `[out]` parameter must be a pointer. If a unique or full pointer is specified, this pointer could be `NULL`, in which case the callee must allocate the memory. But, if the client and the server are in different processes, how can the caller free this memory?

To solve this problem, COM provides a standard *task (memory) allocator* that allocates memory from a memory region shared by processes on either side of a COM interface. The task allocator is an implementation of the `IMalloc` interface, but it is usually accessed via three convenience functions provided by the COM runtime:

```
LPVOID CoTaskMemAlloc(ULONG cb);
LPVOID CoTaskMemRealloc(LPVOID *pv, ULONG cb);
VOID   CoTaskMemFree(void *pv);
```

An example is given on the slide. (Yes, a book actually exists with the title "How Steve Case Beat Bill Gates …"! It recounts the growth of AOL.)



If a function of a custom interface needs to return a pointer to another interface, you might be tempted to specify the pointer type as `void**` in the IDL definition, e.g.:

```
HRESULT GetInterfacePointer([out]void **ppv);  // wrong!
```

However, this won't be accepted by the MIDL compiler because it can't generate any marshalling code for a `void**` pointer.

One solution would be to specify a pointer to an `IUnknown` interface, as follows:

```
HRESULT GetInterfacePointer([out]IUnknown **ppv);
```

Having called this function, a client could then call `QueryInterface()` to obtain a pointer to a specified interface. However, the problem with this approach is that it would take two calls (i.e. two round trips) to obtain a pointer to a particular interface (and not only that, how does `QueryInterface()` get defined in IDL to avoid the same problem?).

The solution is to use the `[iid_is()]` attribute to specify the IID of the desired interface, e.g.:

```
HRESULT GetInterfacePointer([in] REFIID riid,
                            [out, iid_is(riid)]void **pv);
```

This is similar to the way in which `IUnknown::QueryInterface()` is defined (see `unknwn.idl`).

EXE Servers

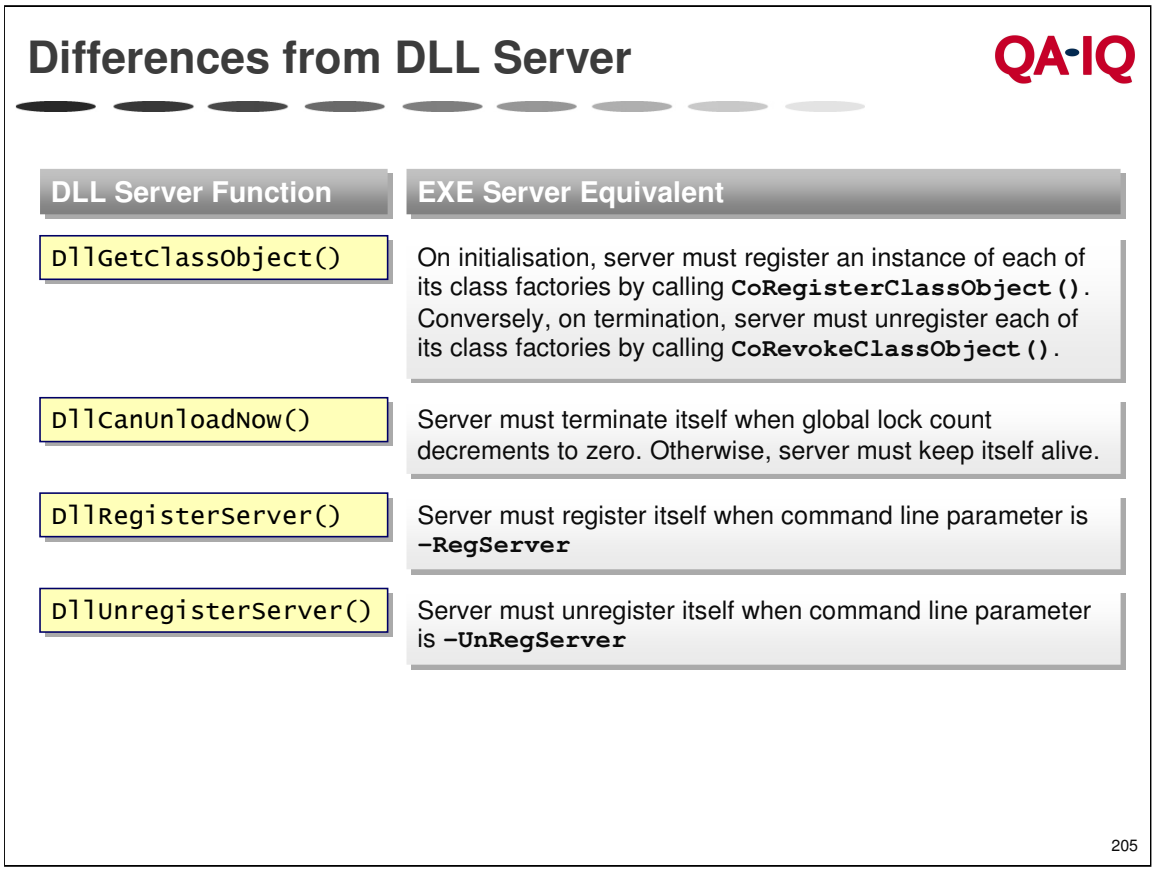
QA-IQ

- No changes required to component class
- Some changes required to class factory
- Server must call CoInitialize() and CoUninitialize() for itself
- Server cannot export any functions such as DllGetClassObject(), etc.
- Server must manage its own lifetime

204

Over the next few slides, we'll look at how to build a EXE server to serve the Account object that we developed in earlier chapters. An EXE server (sometimes called a *local* or *out-of-process* server) has several advantages over a DLL server. For example, because an EXE server has its own address space, it cannot corrupt the address space of a client (or vice versa).

Although there are many differences between an EXE server and a DLL server, as listed on this slide and the next, the key thing is that no changes are required to the component class itself.



As mentioned on the previous slide, an EXE server cannot export any functions, so the same functionality must be provided in other ways.

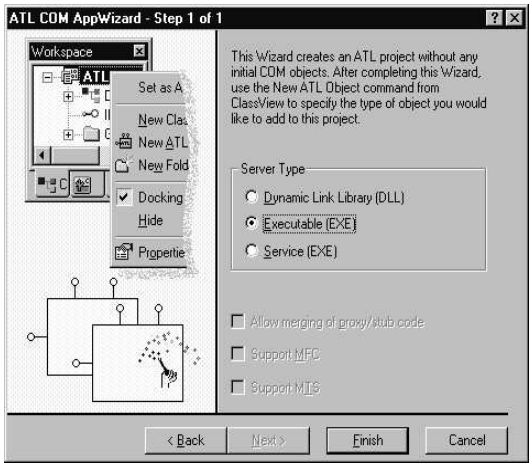
Note that `CoRegisterClassObject()` takes the CLSID of the class factory and a pointer to an instance of the corresponding class factory. It returns a magic *cookie* (like a handle), which is subsequently used in `CoRevokeClassObject()`.

Also note that when a command line parameter such as `-RegServer` or `-UnregServer` is specified, the EXE server should immediately terminate after performing the requested operation.

Building an EXE Server Using ATL

QA-IQ

- **In Visual Studio , create new ATL COM APP Wizard project**
  - **Select Executable (EXE) Server Type, then click Finish**
  - **Wizard generates skeleton IDL, C++ source and header files**



206

As we will see, if you use the *ATL COM App Wizard* provided with Visual Studio 5.0 and 6.0, it is very easy to create a EXE server. As in the case of a DLL server, this wizard generates a number of files, including a skeleton source file for the EXE server itself.

EXE Source File

QA-IQ

- **Declares a global CExeModule object called _Module**
  - **Extends functionality of CComModule to provide lifetime management**

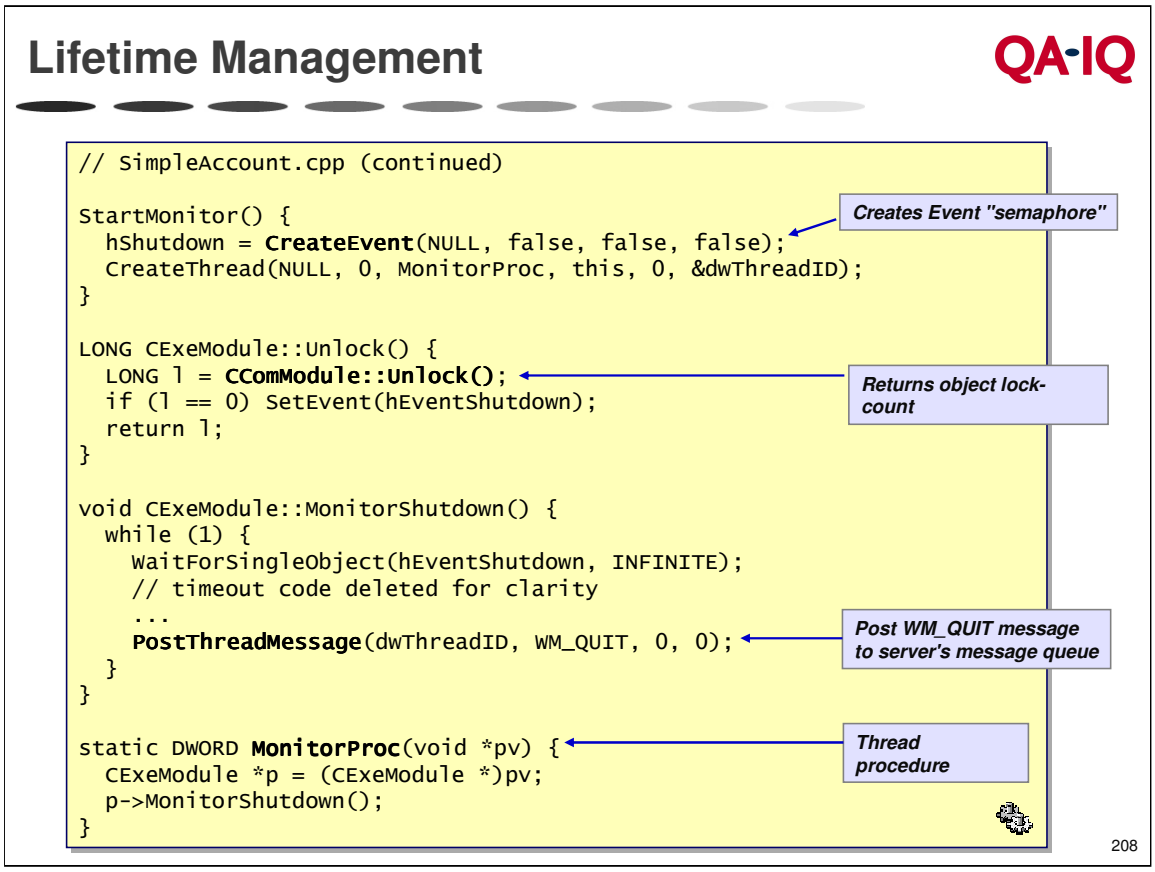
```
// SimpleAccount.cpp : Implementation of WinMain

#include "stdafx.h"
...
CExeModule _Module;

...
```

```
// stdafx.h : include file ...

...
class CExeModule : public CComModule {
public:
  LONG Unlock();
  HANDLE hEventShutdown;
  void MonitorShutdown();
  bool StartMonitor();
  ...
};
```

207

Unlike a DLL server, an EXE server declares a global object of type `CExeModule` rather than `CComModule`. However, as shown on the slide, `CExeModule` is actually derived from `CComModule`.

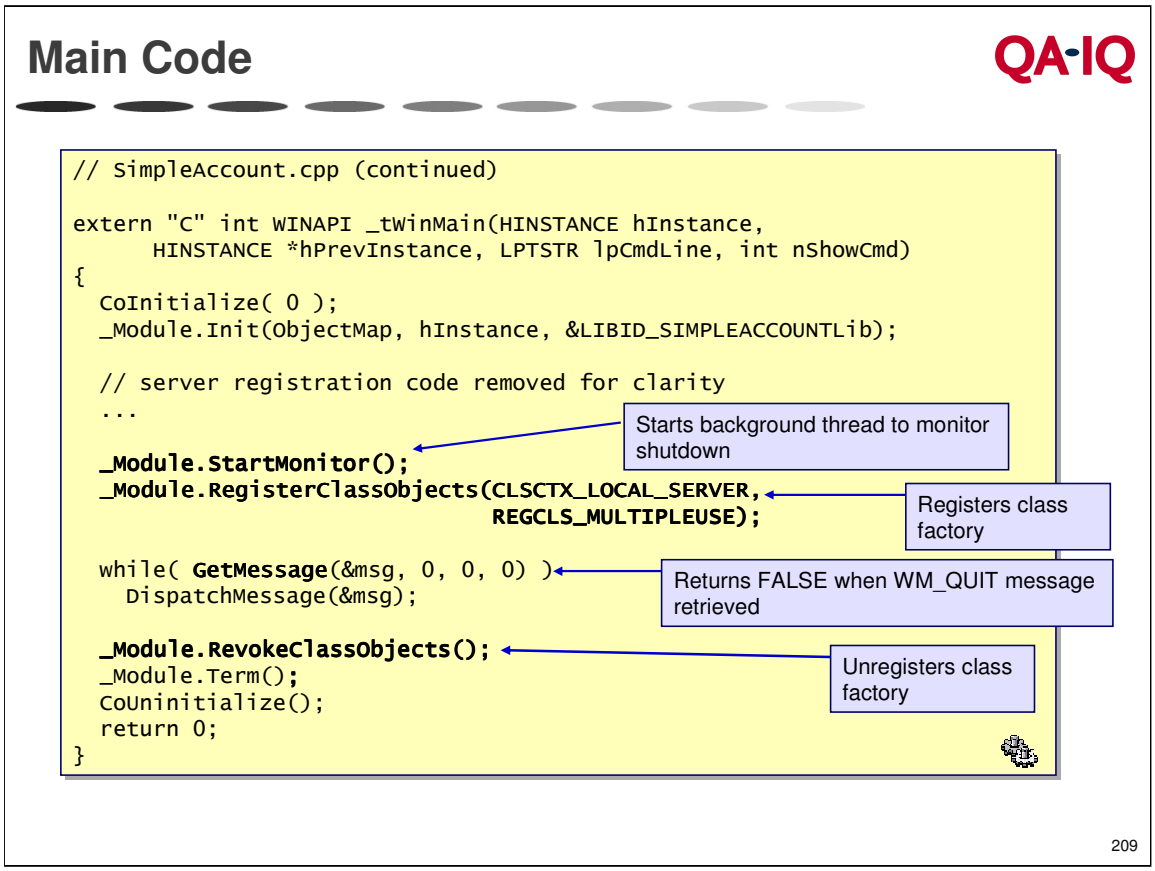
`CExeModule` provides the additional functionality that is required to manage the EXE server's lifetime, and to ensure that the correct registry entries are created and removed as used by EXE servers.



An EXE server must manage its own lifetime, which means it should terminate itself when no client is using (or has locked) any of its COM objects. The `CComModule` class uses a global lock count (`m_nLocked`), which is not only incremented by `Lock()` and decremented by `Unlock()`, but is also incremented and decremented by `CComClassFactory::LockServer()`. As shown on the slide, the wizard-generated code for `CExeModule::Unlock()` overrides `CComModule::Unlock()` but immediately calls `CComModule:Unlock()`. The latter returns the new lock count, so if this has fallen to zero, the shutdown event (semaphore) is set in order to initiate the shutdown sequence.

The shutdown sequence involves two timeouts, which are implemented in a background thread (the details of which are not shown on the slide). When both timeouts expire, the background posts a `WM_QUIT` message to the server's message queue, which will force the server to terminate.

These additional timeouts are in place to minimise race conditions that might occur in earlier versions of COM.



This slide shows the `_tWinMain()` function (in a simplified form) in the EXE server code.

ATL implements a background worker thread to help with server lifetime management. Having started this thread, it then registers the class objects. The combination of flags passed to `CComModule::RegisterClassObjects()` means that a class factory will be visible both in-process and out-of-process, and that multiple client applications will be able to connect to a single instance of a class factory (i.e. a singleton).

The program then enters a standard Windows message pump. This pumps messages (which is a requirement for any thread that calls CoInitialize(0) and then creates COM objects - more on this later). The background worker thread will post the `WM_QUIT` message to this main thread's queue when there are no more objects and no locks on the server, which will cause the program to end.

Registration Code

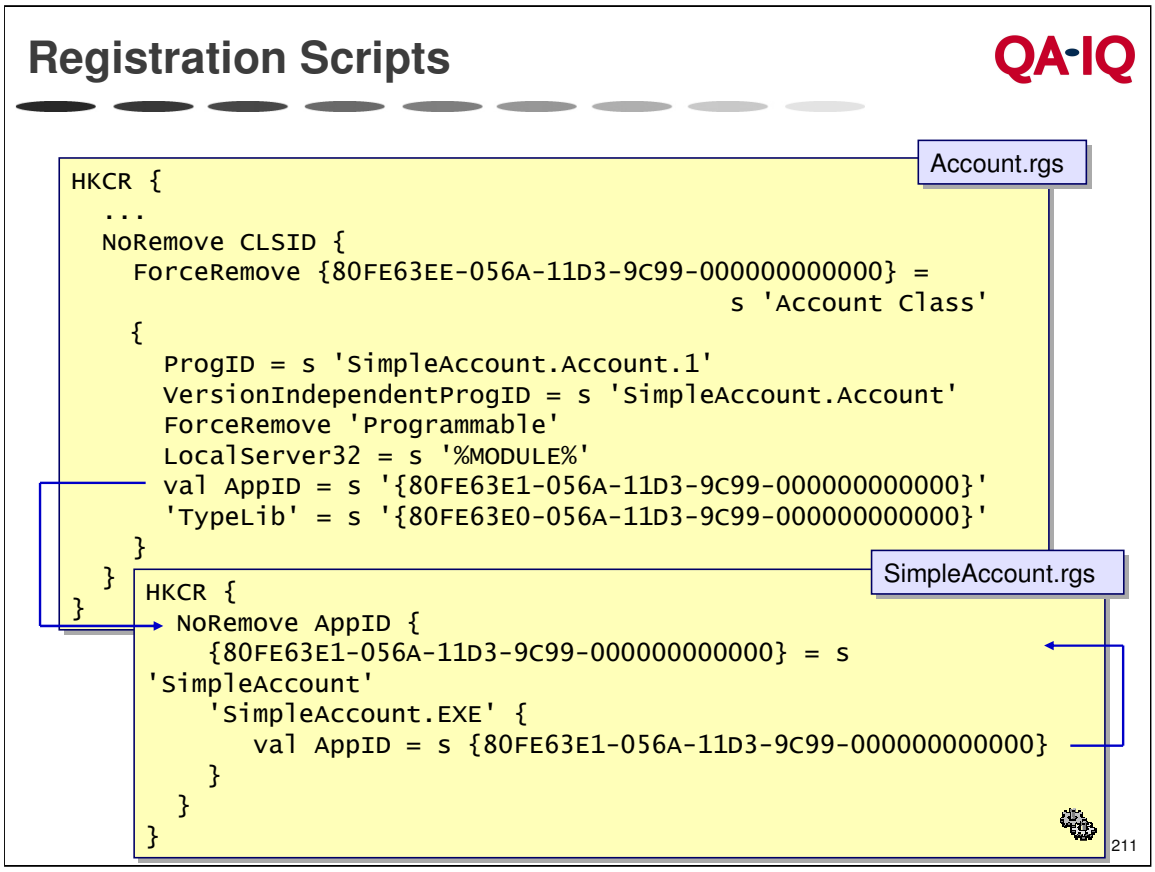
QA-IQ

```
// SimpleAccount.cpp (simplified for clarity)

extern "C" int WINAPI _tWinMain(HINSTANCE hInstance,
      HINSTANCE *hPrevInstance, LPTSTR lpCmdLine, int nShowCmd)
{
  ...
  // command-line tokenising code removed for clarity
  ...
  if (lstrcmpi(lpszToken, _T("UnregServer")) == 0)
  {
    _Module.UpdateRegistryFromResource(IDR_SimpleAccount, FALSE);
    _Module.UnregisterServer(TRUE);
  }
  else if (lstrcmpi(lpszToken, _T("RegServer")) == 0)
  {
    _Module.UpdateRegistryFromResource(IDR_SimpleAccount, TRUE);
    _Module.RegisterServer(TRUE);
  }
  ...
  ...
}
```

210

Here is the registration code that was removed from the previous slide for reasons of clarity. Note that, unlike a DLL server, an EXE server built using ATL automatically registers an *AppID*, in addition to a coclass and a type library. This allows the served component(s) to be accessed remotely, as we will see later.

Not shown in the code on the slide is the `bRun` flag, which is set to `FALSE` whenever registration or unregistration is performed. This causes the program to exit after performing its registration code, as required by the COM specification.



With an EXE server, the *ATL Object Wizard* generates two registration scripts. The first is similar to that generated for a DLL server, except that the server is registered under the key *LocalServer32* instead of *InProcServer32*. An *AppID* named value is also added underneath the server's *CLSID*, the value of which is a GUID that refers to an *AppID* entry under `HKEY_CLASSES_ROOT\AppID` as specified by the second script.

As we will see later, *AppID*s are used by DCOM to allow remote access to a component. A fully-specified *AppID* will have at least two values, but the script generated by the *ATL Object Wizard* simply gives the *AppID* a single named value, which is a friendly name (in this example,. *SimpleAccount*).

The script also adds an entry underneath `HKEY_CLASSES_ROOT\AppID` for the server application (in this case, *SimpleAccount.EXE*). This key has a single value which points back to the *AppID*.

Building the Proxy-Stub DLL

QA-IQ

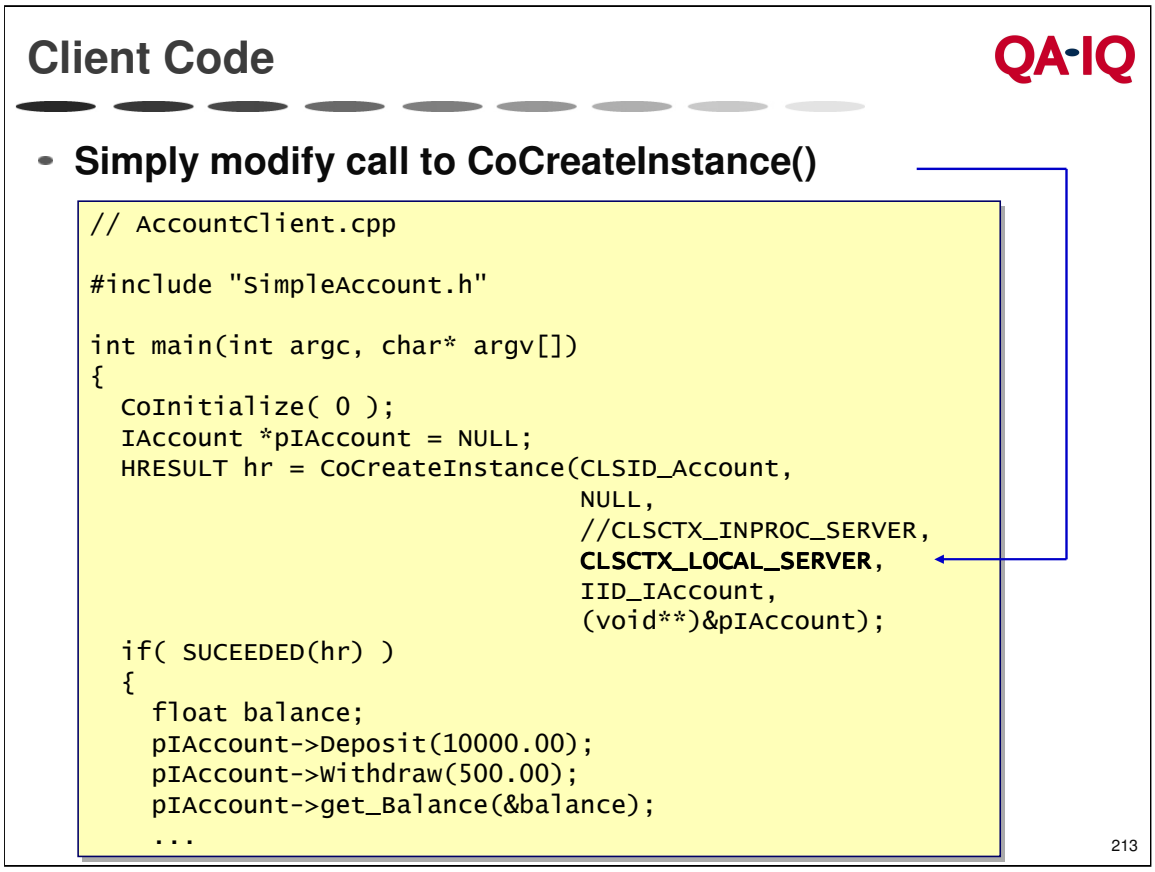
- **ATL COM App Wizard provides a definition file ...**
  ... and a makefile
- **Build proxy-stub DLL ...**
  - **nmake /f SimpleAccountps.mk**
- **... and register it**
  - **regsvr32 SimpleAccountps.dll**

```
LIBRARY         "SimpleAccountPS"

DESCRIPTION  'Proxy/Stub DLL'

EXPORTS
    DllGetClassObject       @1 PRIVATE
    DllCanUnloadNow         @2 PRIVATE
    GetProxyDllInfo         @3 PRIVATE
    DllRegisterServer       @4 PRIVATE
    DllUnregisterServer     @5 PRIVATE
```

212

The *ATL COM App Wizard* provides a proxy/stub DLL definition file and a makefile, so building the MIDL-generated proxy/stub DLL is dead easy. However, if you want to use this proxy/stub DLL, you must remember to install and register it on both client and server machines.

Note that an EXE server built using the *ATL COM App Wizard* will automatically register the type library when the server registers itself. Therefore, if the type library describes a dual interface, or one or more Automation-compatible interfaces, `oleaut.dll`, which implements the Automation marshaller, will be registered as the proxy/stub DLL. In this case, there is no need to build, install or register the MIDL-generated proxy/stub DLL. However, if any of the interfaces are not Automation-compatible, you will need to use standard marshalling as implemented by the MIDL-generated proxy/stub DLL.



The only change required to the client is to specify `CLSCTX_LOCAL_SERVER` instead of `CLSCTX_INPROC_SERVER` in the call to `CoCreateInstance()`. Now we can test that our EXE server works on a single machine.

Note that if your client doesn't mind whether it uses either an inproc server or a local server, it can OR the two flags together, as shown below:

```
CLSCTX_LOCAL_SERVER|CLSCTX_INPROC_SERVER
```

Additionally, if your client doesn't mind what type of server it talks to at all, you can use `CLSCTX_ALL` or `CLSCTX_SERVER`.

Summary

QA-IQ

- A custom interface can use either standard marshalling or type-library marshalling
  - Standard marshalling is implemented by MIDL-generated stub/proxy code
  - Type-library marshalling is implemented by the Automation marshaller
- When an interface is defined in IDL, attributes allow proxy/stub code to work out how many bytes to marshal
  - Very important with arrays and strings
- The ATL COM App Wizard provides an option for building an EXE server

214