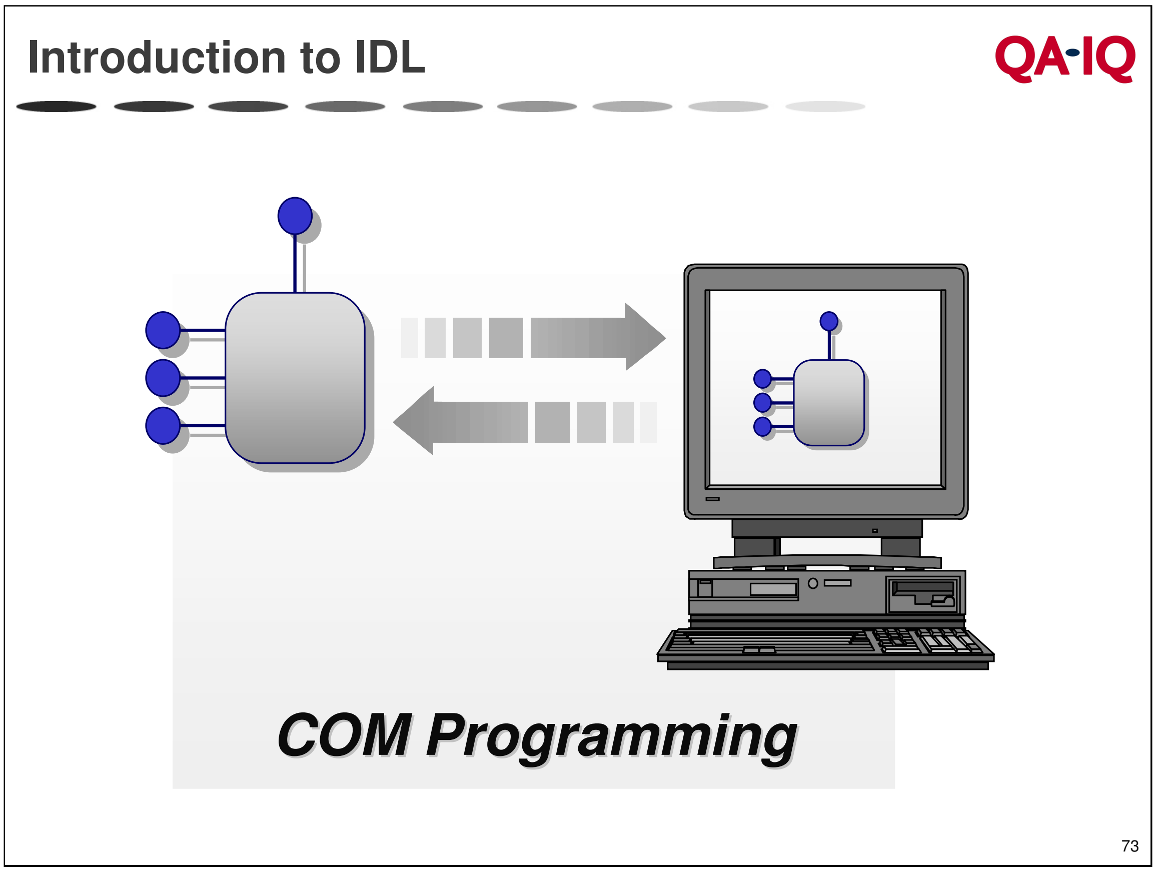
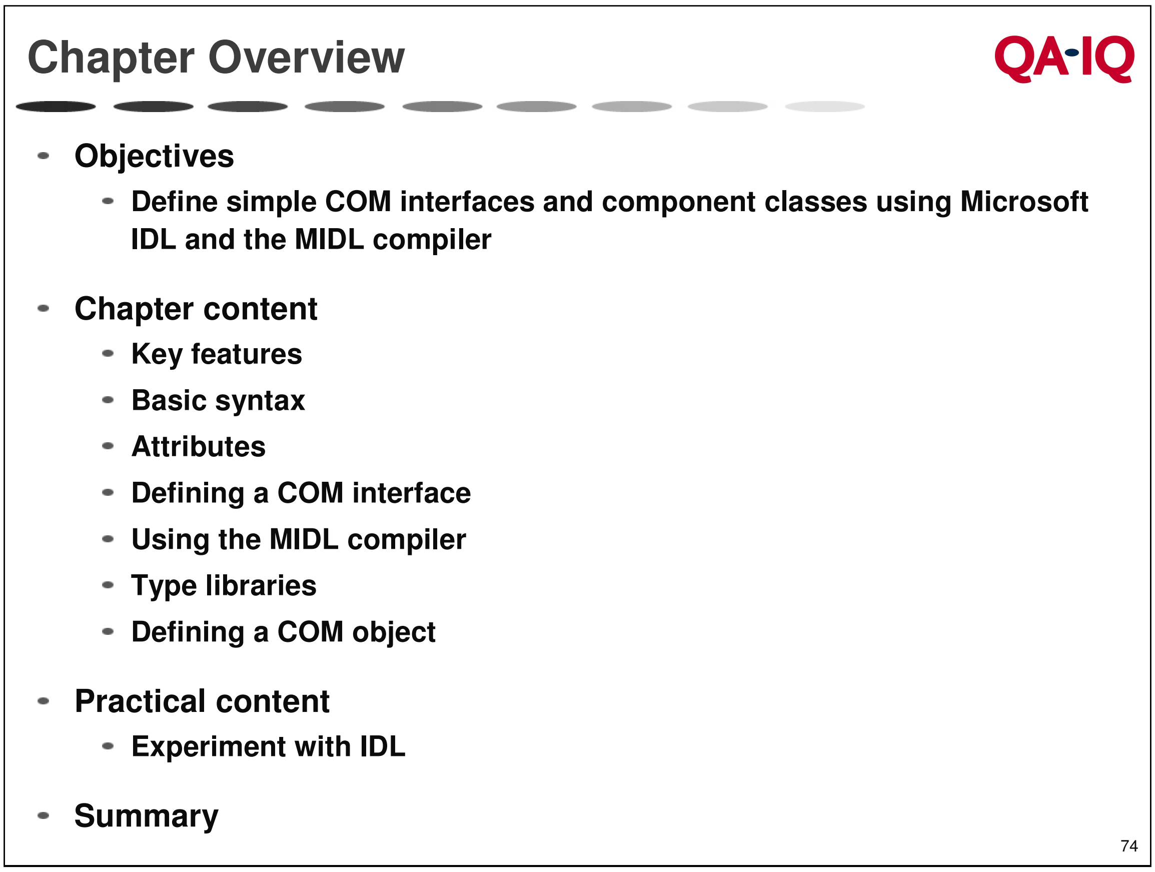


Introduction to IDL

QA-IQ

73



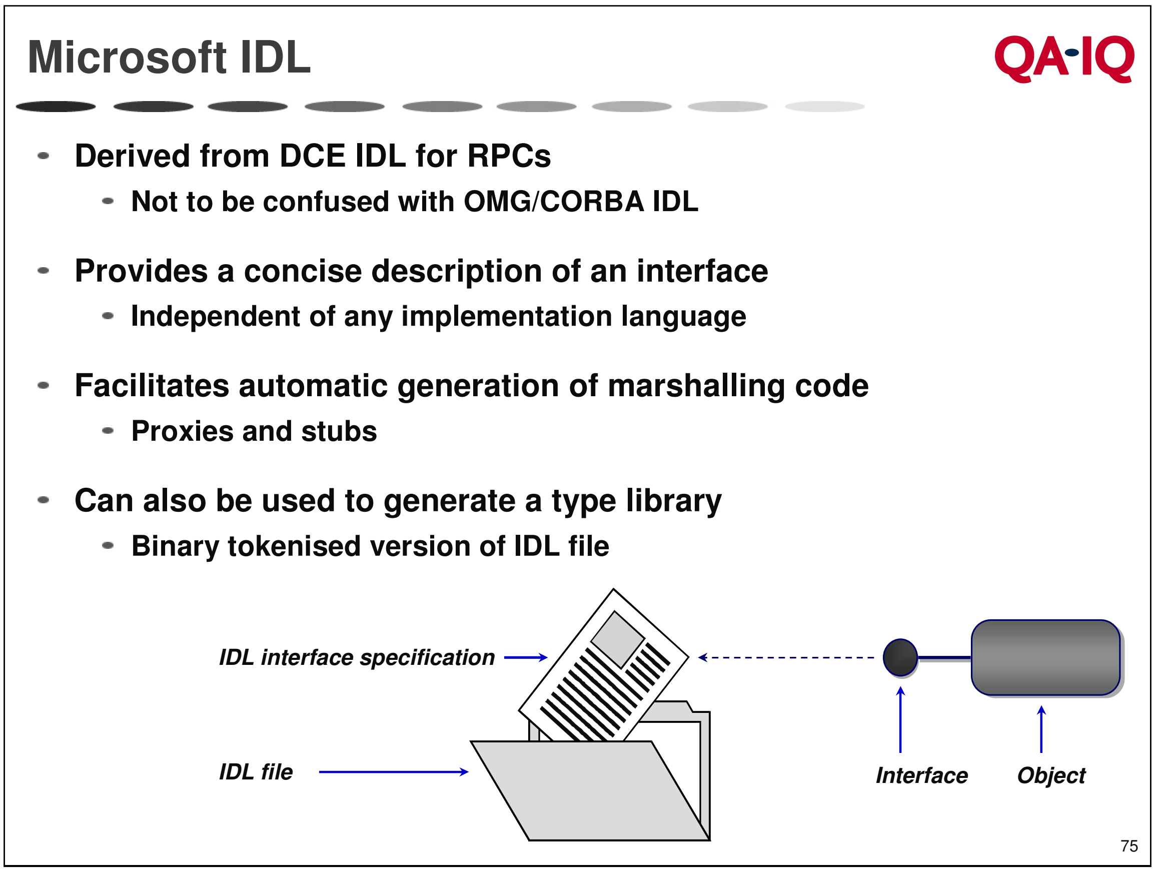
Chapter Overview

QA-IQ

- Objectives
  - Define simple COM interfaces and component classes using Microsoft IDL and the MIDL compiler
- Chapter content
  - Key features
  - Basic syntax
  - Attributes
  - Defining a COM interface
  - Using the MIDL compiler
  - Type libraries
  - Defining a COM object
- Practical content
  - Experiment with IDL
- Summary

74

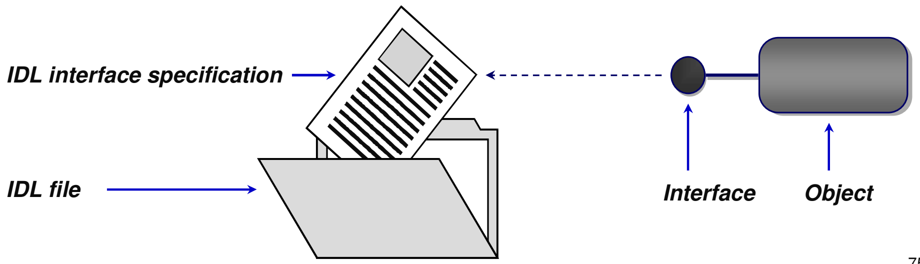
In this chapter, we look at Microsoft Interface Definition Language (IDL), which is the preferred way to describe COM interfaces and component classes.



Microsoft IDL

QA-IQ

- Derived from DCE IDL for RPCs
  - Not to be confused with OMG/CORBA IDL
- Provides a concise description of an interface
  - Independent of any implementation language
- Facilitates automatic generation of marshalling code
  - Proxies and stubs
- Can also be used to generate a type library
  - Binary tokenised version of IDL file

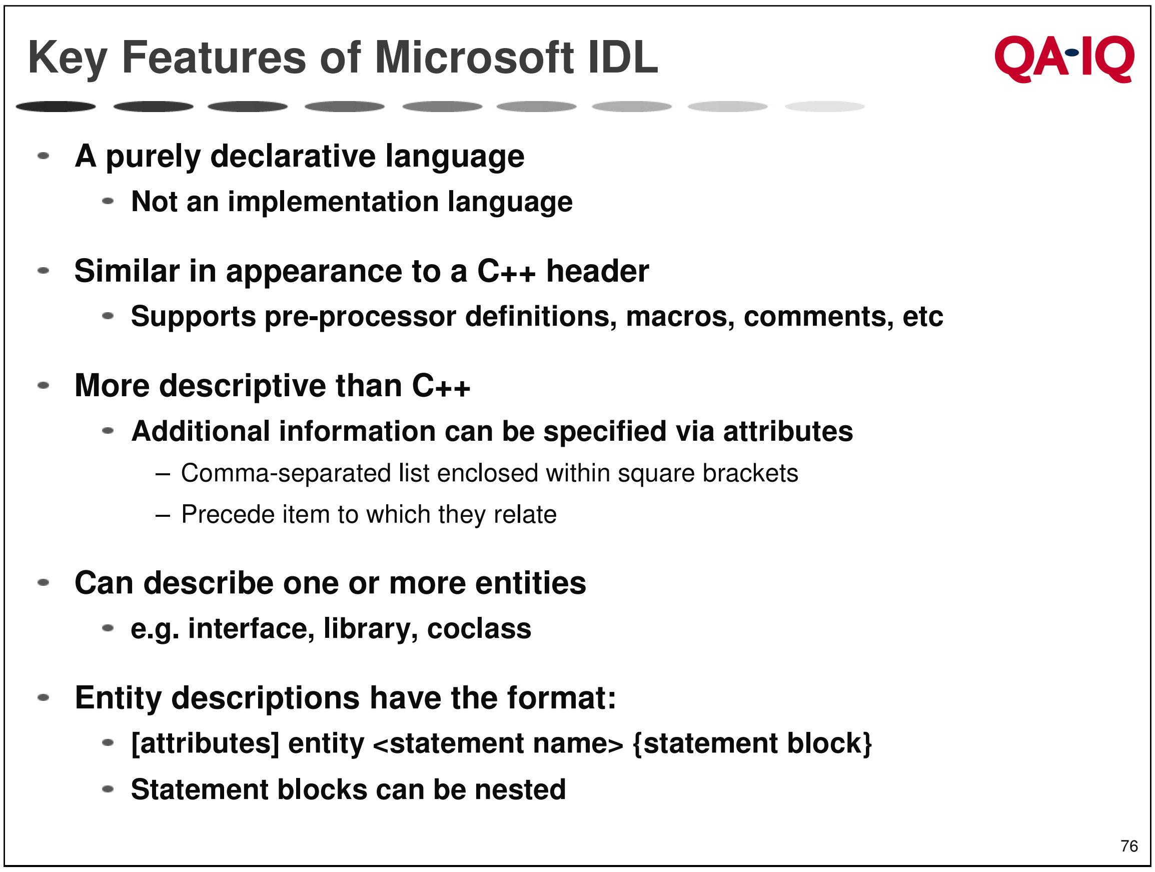


75

IDL was originally developed for the Open Software Foundation's Distributed Computing Environment (DCE) in order to describe Remote Procedure Calls (RPCs). Special attributes were designed so that an IDL compiler could generate the necessary proxy and stub code to marshal parameters between the caller and the called procedure in the most efficient manner. Later, a similar, but different, IDL was specified by the Object Management Group (OMG) to describe the interfaces of Common Object Request Broker Architecture (CORBA) objects.

Rather than re-invent the wheel, Microsoft based its implementation of RPC on DCE RPC, so Microsoft's first version of IDL (generally known as MIDL) was more-or-less identical to DCE IDL. When Microsoft developed the foundation of OLE (now known as COM), it was natural to use IDL to describe COM interfaces. In addition, Microsoft developed a new Object Definition Language (ODL) to describe Automation objects and their dispatch interfaces. A tool called MKTYPLIB was used to take a ODL file and generate a type library, which could be used to support early binding with Visual Basic clients.

With Windows NT 4.0, Microsoft released MIDL 3.0, which extended IDL to support all of functionality of ODL. A single file could then be used to describe all of the features of a COM class, and the new MIDL 3.0 compiler could generate a type library as well as C++ headers, proxies and stubs.



Key Features of Microsoft IDL

- **A purely declarative language**
  - **Not an implementation language**
- **Similar in appearance to a C++ header**
  - **Supports pre-processor definitions, macros, comments, etc**
- **More descriptive than C++**
  - **Additional information can be specified via attributes**
    - Comma-separated list enclosed within square brackets
    - Precede item to which they relate
- **Can describe one or more entities**
  - **e.g. interface, library, coclass**
- **Entity descriptions have the format:**
  - **[attributes] entity <statement name> {statement block}**
  - **Statement blocks can be nested**

76

It is important to note that although Microsoft IDL looks similar in appearance to a C++ header, it is used purely to describe COM objects and/or COM interfaces. An IDL file does not contain any implementation code whatsoever.

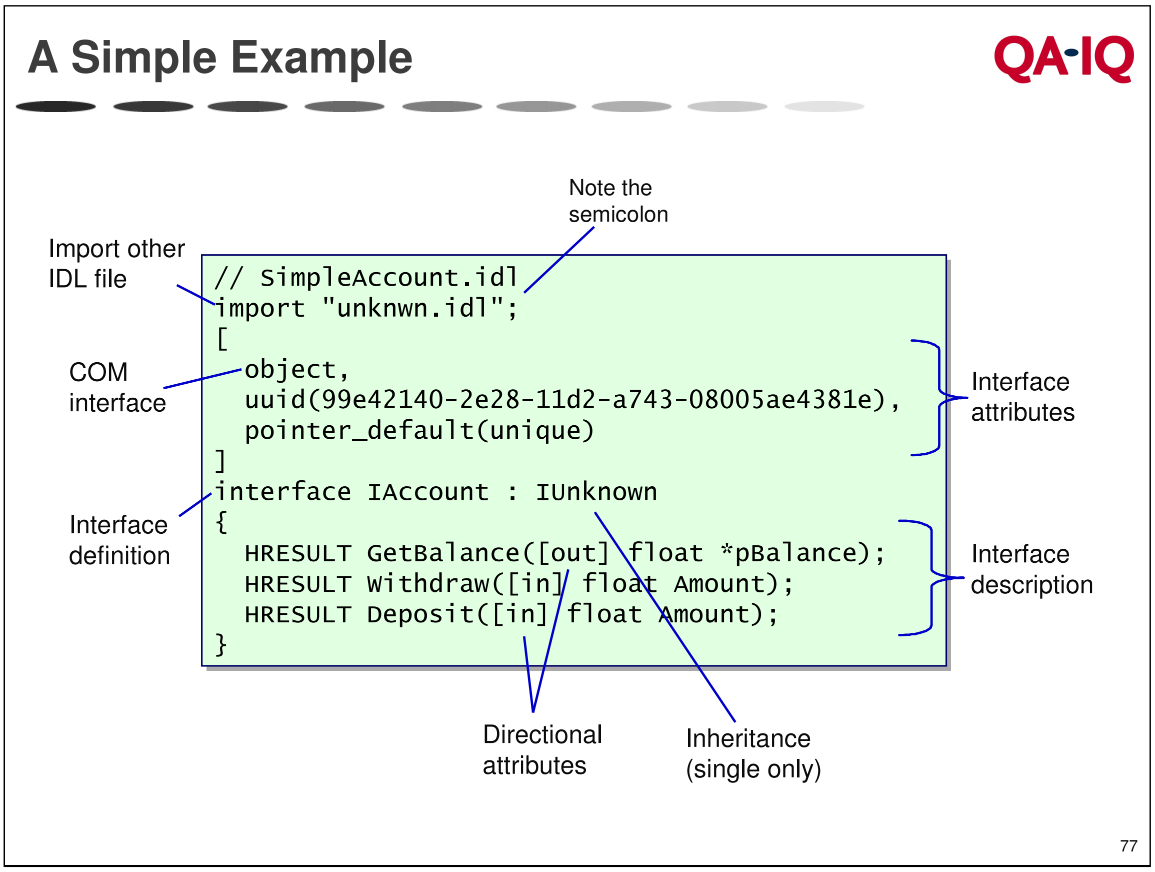
However, you can include #defines, macros and comments in an IDL file, as you would in a C or C++ file. IDL supports all of the primitive data types of C++, but unlike C++ (and with the exception of the `int` type), the representation of these types is independent of the implementation language and the target platform. IDL also supports many of the data-definition keywords of C++, such as `const`, `enum`, `typedef`, `struct` and `union`.

Another difference between IDL and C++ is that an IDL declaration can be annotated using *attributes*. These attributes must be placed within square brackets and must immediately precede the item, e.g. function parameter, to which they relate. Multiple attributes can be specified using a comma-separated list within the square brackets.

As mentioned on the previous slide, Microsoft IDL can now describe more than COM interfaces. Strictly speaking, it can describe one or more *entities*, where an entity is an `interface`, `dispinterface`, `library`, `coclass` or `module`. All entity-description statements in an IDL file have following format:

```
[attribute(s)] entity <statement name> {statement block};
```

where *`entity`* is a keyword such `interface`, `library` or `coclass`, etc., *`statement name`* is a programmatic name, and `statement block` can contain data definitions, `typedefs` (and other statements) inside a pair of curly braces. Each statement can be preceded by a list of comma-separated attributes inside square brackets.



A Simple Example

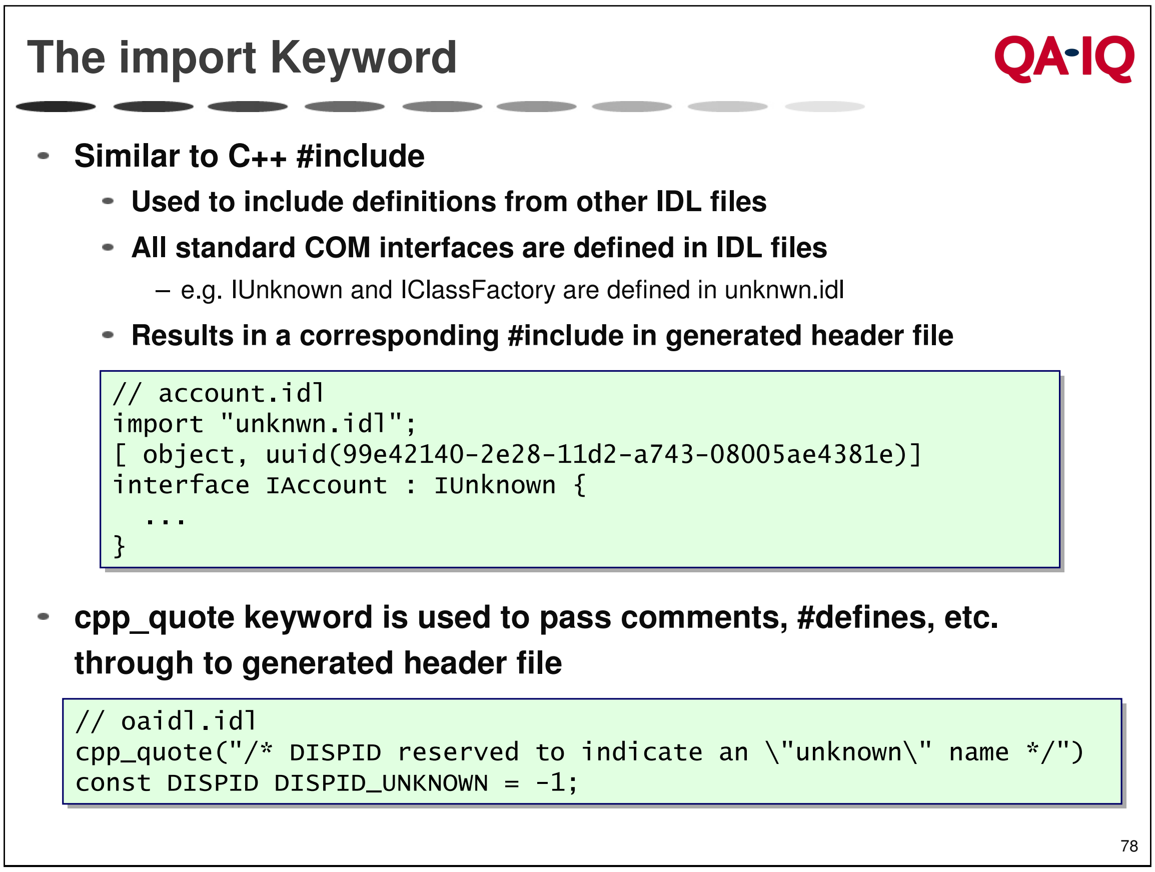
```
// SimpleAccount.idl
import "unknwn.idl";
[
  object,
  uuid(99e42140-2e28-11d2-a743-08005ae4381e),
  pointer_default(unique)
]
interface IAccount : IUnknown
{
  HRESULT GetBalance([out] float *pBalance);
  HRESULT Withdraw([in] float Amount);
  HRESULT Deposit([in] float Amount);
}
```

QA-IQ
Note the semicolon
Import other IDL file
COM interface
Interface definition
Interface attributes
Interface description
Directional attributes
Inheritance (single only)
77

The slide shows the IDL description of a simple interface called `IAccount`. At this point, we are interested only in the general structure; we will discuss the details as we progress through the chapter.

You will notice that the structure conforms to the format of an entity description that we explained on the previous slide. Here, we have a single entity (an `interface`) called `IAccount`. The statement block, which describes this interface, contains signatures for its functions, `GetBalance()`, `Withdraw()` and `Deposit()`. Note that each function parameter is preceded by a directional attribute, such as `in` and `out`, which specifies whether the parameter is passed *in* to the object or *out* of the object, respectively.

The entire interface description is also preceded by an attribute block that includes the `object` keyword, which indicates that this is the description of a *COM* interface and not an RPC interface.



The import Keyword

- Similar to C++ #include
  - Used to include definitions from other IDL files
  - All standard COM interfaces are defined in IDL files
    - e.g. IUnknown and IClassFactory are defined in unknwn.idl
  - Results in a corresponding #include in generated header file

```
// account.idl
import "unknwn.idl";
[ object, uuid(99e42140-2e28-11d2-a743-08005ae4381e)]
interface IAccount : IUnknown {
  ...
}
```

- cpp_quote keyword is used to pass comments, #defines, etc. through to generated header file

```
// oaidl.idl
cpp_quote("/* DISPID reserved to indicate an \"unknown\" name */")
const DISPID DISPID_UNKNOWN = -1;
```

78

The `import` keyword specifies the names of one or more IDL files to import. It is used in a similar way to the C `#include` directive, and results in a corresponding `#include` in the generated header file. Incidentally, note that the `import` statement is terminated with a semi-colon.

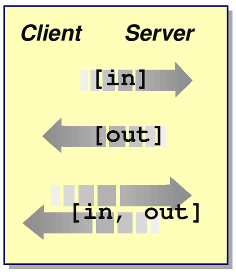
All of the standard COM interfaces are defined in IDL files that are included with Visual C++; these include `wtypes.idl`, `unknwn.idl`, `objidl.idl` and `oaidl.idl`. For example, `IUnknown` and `IClassFactory` are defined in `unknwn.idl`.

The MIDL compiler expands `#define` statements and ignores comments so these will not appear in the generated headers. If you need to pass `#defines` and comments through to the generated header, use the IDL `cpp_quote` keyword as shown on the slide.

Function Declarations

QA-IQ

- Similar to C++ except for parameter attributes
  - Enclosed in square brackets
  - Precede formal parameter
- Use directional attributes to specify passing mode
  - [in], [out] and [in, out]
  - [out] and [in, out] parameters must be pointers
- Use [retval] attribute on primary out parameter
  - Maps parameter to return value as seen by VB and Java clients

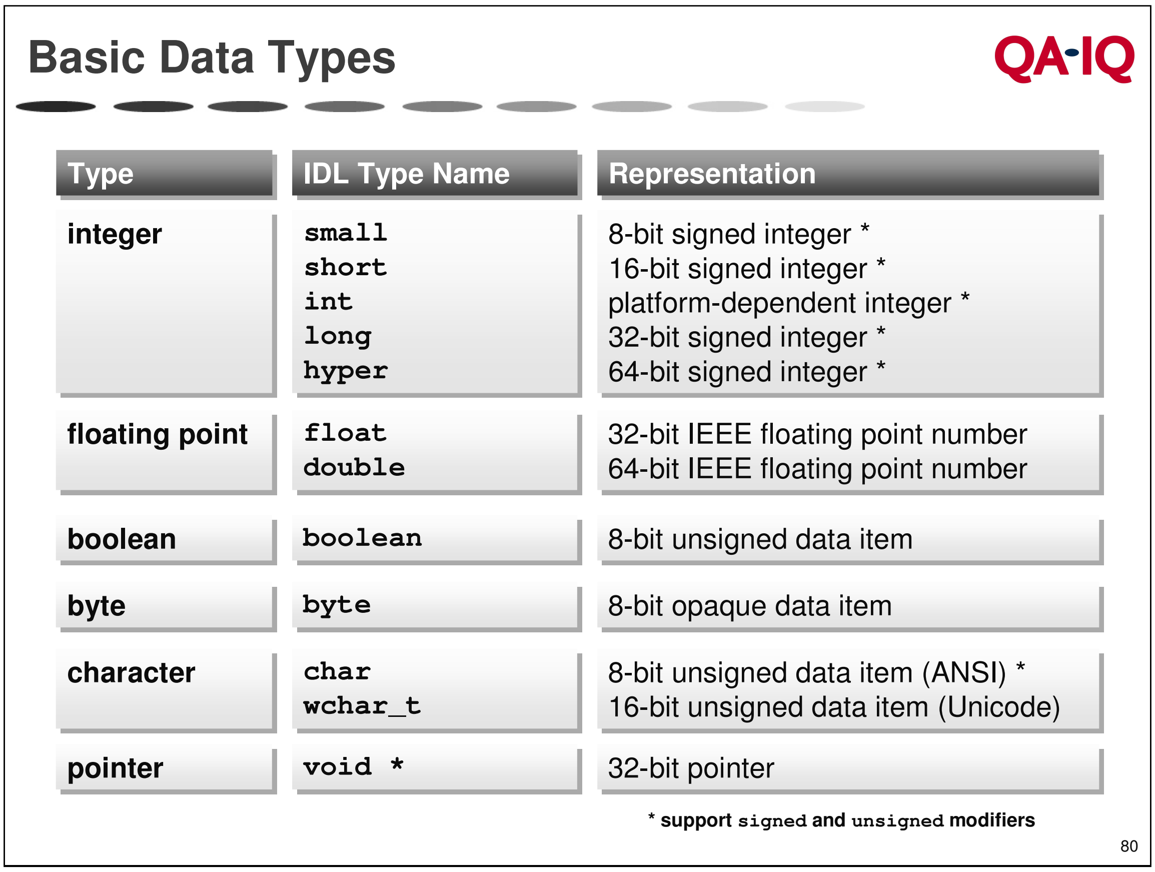


79

Directional attributes, namely `[in]` and `[out]`, are used to specify whether a function parameter is passed *in* from caller to callee (i.e. from a client to a server) or *out* from callee to caller (i.e. from a server to a client). If neither attribute is specified, the parameter is assumed to be an *in* parameter by default. A single parameter can also be used to pass data in both directions (i.e. to update a value), in which case the `[in, out]` attribute must be used.

Note that both `[out]` and `[in,out]` attributes are meaningful only when used with pointer parameters, since, by default, parameters are passed by value (as in C and C++).

Functions can have more than one *out* parameter, which means that a function can return several values. The primary *out* parameter should also be qualified with the `[retval]` attribute, which means that this parameter will become the return value when called by Visual Basic, Java and scripting clients. (The HRESULT returned by most COM interface functions is hidden from clients written in these languages.)



Basic Data Types

Type	IDL Type Name	Representation

integer	`small`
	`short`
	`int`
	`long`
	`hyper`
		8-bit signed integer *
		16-bit signed integer *
		platform-dependent integer *
		32-bit signed integer *
		64-bit signed integer *

floating point	`float` `double`	32-bit IEEE floating point number 64-bit IEEE floating point number
boolean	`boolean`	8-bit unsigned data item
byte	`byte`	8-bit opaque data item
character	`char` `wchar_t`	8-bit unsigned data item (ANSI) * 16-bit unsigned data item (Unicode)
pointer	`void *`	32-bit pointer

*** support `signed` and `unsigned` modifiers**

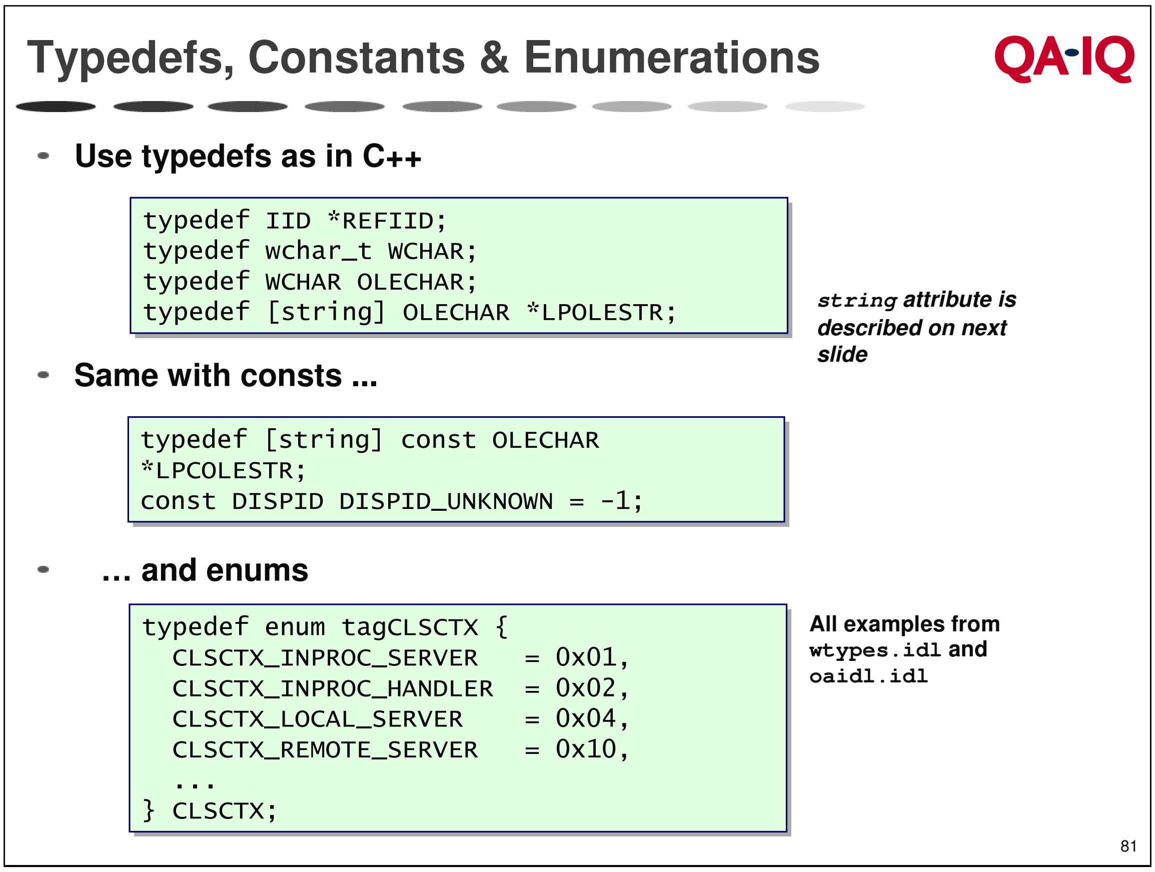
80

The above table shows the basic data types supported by IDL. Note that, with the exception of `int`, the representation of these types is independent of the implementation language and target platform.

It should also be noted that the default char type for IDL is *unsigned char*, whereas the default for the Microsoft C/C++ environment is *signed char*.

However, independence from implementation languages only goes so far. Visual Basic, for example, is unable to cope with unsigned integer values, so for compatibility with VB, only signed `short`s and `long`s should be used.

IDL also supports the passing of interface pointers (for example IUnknown *).



Typedefs, Constants & Enumerations

QA-IQ

- Use typedefs as in C++

```
typedef IID *REFIID;
typedef wchar_t WCHAR;
typedef WCHAR OLECHAR;
typedef [string] OLECHAR *LPOLESTR;
```

`string` attribute is described on next slide

- Same with consts ...

```
typedef [string] const OLECHAR
*LPCOLESTR;
const DISPID DISPID_UNKNOWN = -1;
```

- ... and enums

```
typedef enum tagCLSCTX {
  CLSCTX_INPROC_SERVER    = 0x01,
  CLSCTX_INPROC_HANDLER   = 0x02,
  CLSCTX_LOCAL_SERVER     = 0x04,
  CLSCTX_REMOTE_SERVER    = 0x10,
  ...
} CLSCTX;
```

All examples from `wtypes.idl` and `oaidl.idl`

81

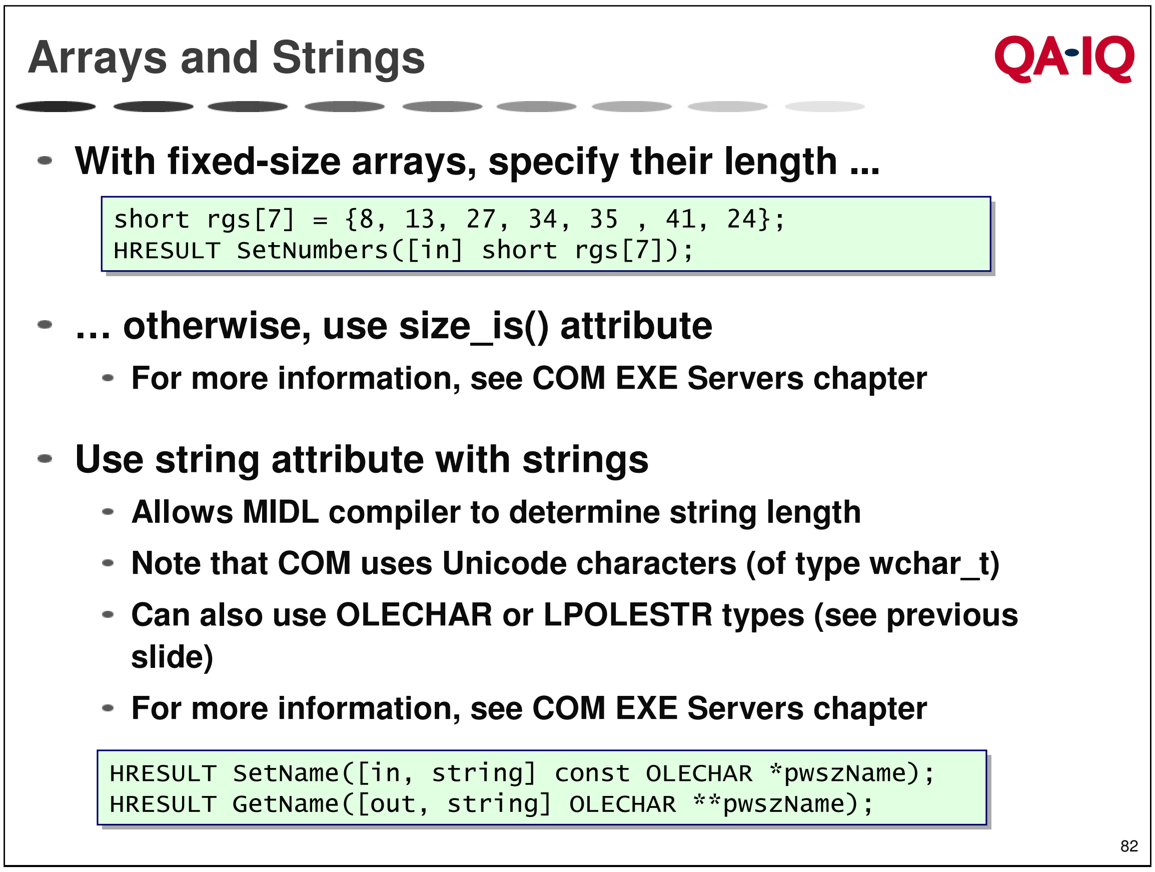
As shown above, you can use `typedef`s, `const`s and `enum`s in the same way as you would with C++. All of these examples come from one of the standard IDL files, either `wtypes.h` and `oaidl.idl`. It's worth having a look at these to get an idea of what you can do with IDL - you will find them in the `include` subdirectory of Visual C++.

You can also declare user defined types (`struct`s) in IDL. However, it should be noted that, unlike C++, you either have to use the `struct` keyword in front of every use of the new type, or you must use the `typedef` keyword when you declare the type.

For example, you can declare a Date structure using the following syntax:

```
typedef struct tagDate
{
  short day;
  short month;
  short year;
} Date;
```

Bear in mind that UDTs might not be supported in all implementation languages, such as VB Script.



Arrays and Strings

QA-IQ

- With fixed-size arrays, specify their length ...

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

- ... otherwise, use size_is() attribute
  - For more information, see COM EXE Servers chapter
- Use string attribute with strings
  - Allows MIDL compiler to determine string length
  - Note that COM uses Unicode characters (of type wchar_t)
  - Can also use OLECHAR or LPOLESTR types (see previous slide)
  - For more information, see COM EXE Servers chapter

```
HRESULT SetName([in, string] const OLECHAR *pwszName);
HRESULT GetName([out, string] OLECHAR **pwszName);
```

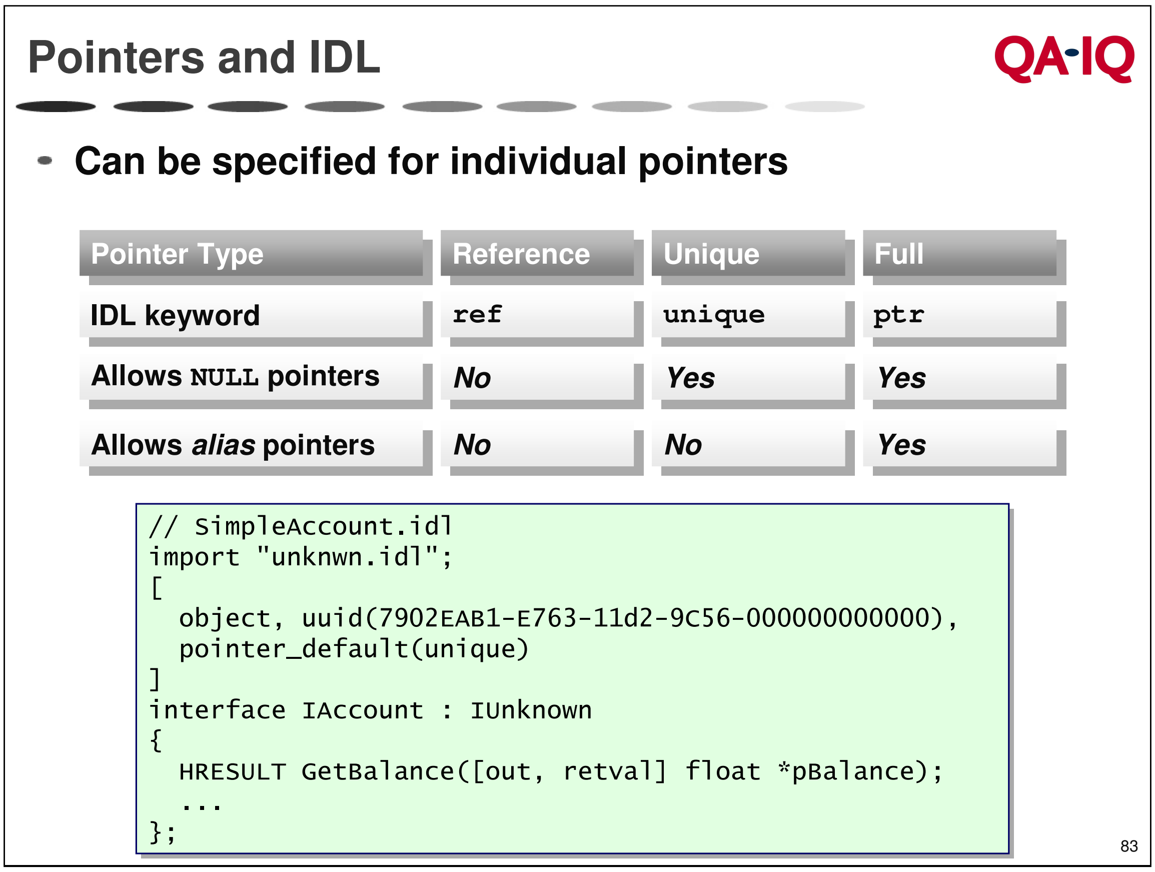
82

As a C++ programmer, you will know that C/C++ pointers and arrays are interchangeable, which means that if you call a function that takes a pointer to an array, the called function has no idea of the array's size. To get round this problem, the function would typically provide another parameter to take the size of the array.

With RPC and COM, the problem is more complicated, because if the caller and the called function reside in different address spaces, the contents of the array will need to be marshalled in some way. In other words, the MIDL compiler must generate appropriate proxy and stub code to pass a copy of the array contents from one address space to the other. To do this, the MIDL compiler clearly needs to know the size of the array. If the size of the array is known at compile time, this is easy as shown in the first example on the slide. The `[in]` attribute specifies that the array is being passed in from the caller to the called function, which means that it needs to be marshalled in one direction only. If the size of the array is not known at compile time, the `size_is()` attribute must be used as will be described in the *COM EXE Servers* chapter.

In IDL, strings are arrays of Unicode characters (i.e. `wchar_t`s), but unlike arrays of other types, IDL strings are assumed to be NULL-terminated (as in C and C++), so if you precede string parameters with the `[string]` attribute, the MIDL compiler can work out the length of the string. You also need to specify whether a string is an in or out parameter, as shown in the second example on the slide. If a string is an `[out]` parameter, the convention is that the callee is responsible for allocating the string buffer, and the caller is responsible for freeing it. COM provides a special memory allocator for this purpose, which we will discuss in more detail in the *COM EXE Servers* chapter.

Incidentally, COM function declarations typically use the `OLECHAR` and `LPOLESTR` types, which are derived from `wchar_t`, as shown on the previous slide.



Pointers and IDL

QA-IQ

- Can be specified for individual pointers

Pointer Type	Reference	Unique	Full
IDL keyword	`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);
  ...
};
```

83

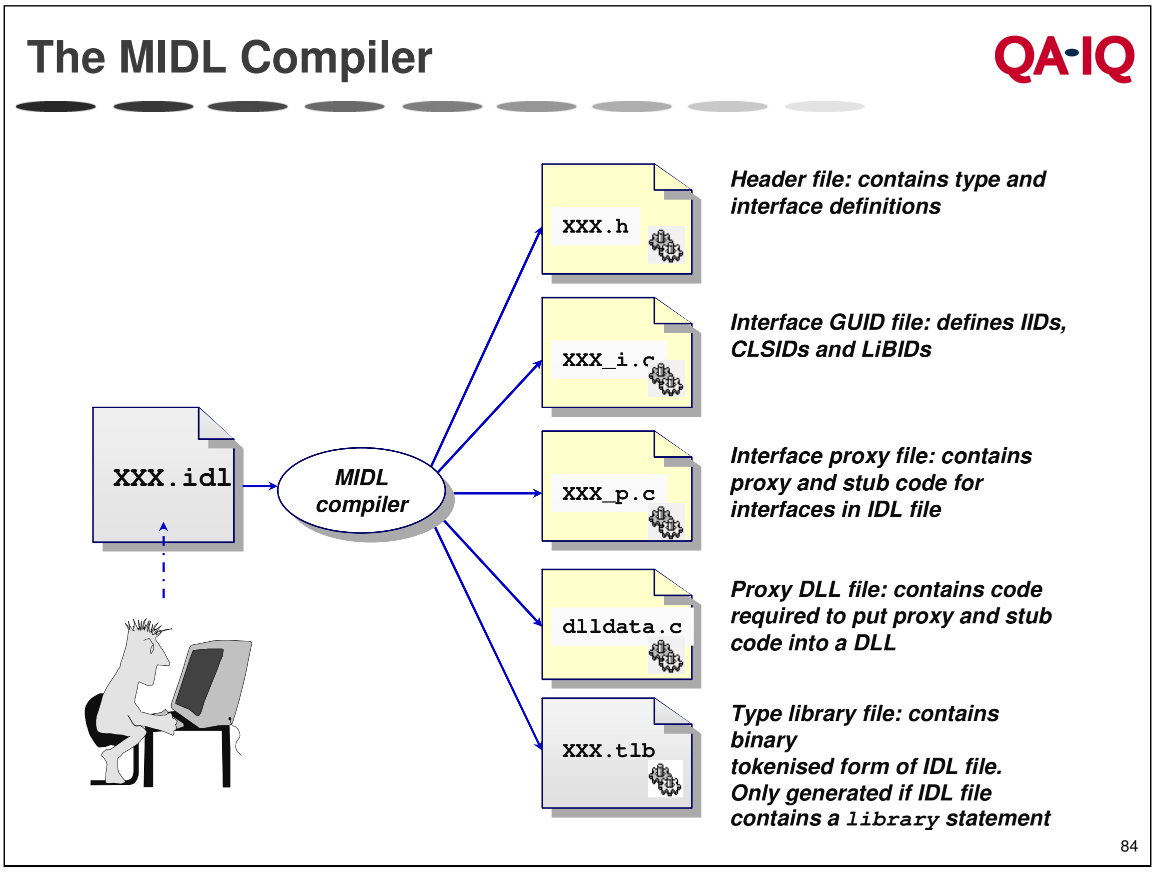
A function parameter of a basic data type, such as a `float`, is marshaled by simply copying its value from the address space of the client process to the address space of the server process. However, if a function parameter is a pointer, the data pointed to by this pointer must be copied from one address space to another. Therefore, it is fairly obvious that additional information must be provided to tell the marshaler 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 marshaler can check for a `NULL` value before it's sent across the wire. Furthermore, two reference pointers that are passed in the same call cannot contain the same address, i.e., they can't be duplicated or *aliased*. This means that the marshaler doesn't have to check for alias pointers in one address space, and the unmarshaler 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).



The MIDL Compiler

Header file: contains type and interface definitions

Interface GUID file: defines IIDs, CLSIDs and LiBIDs

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

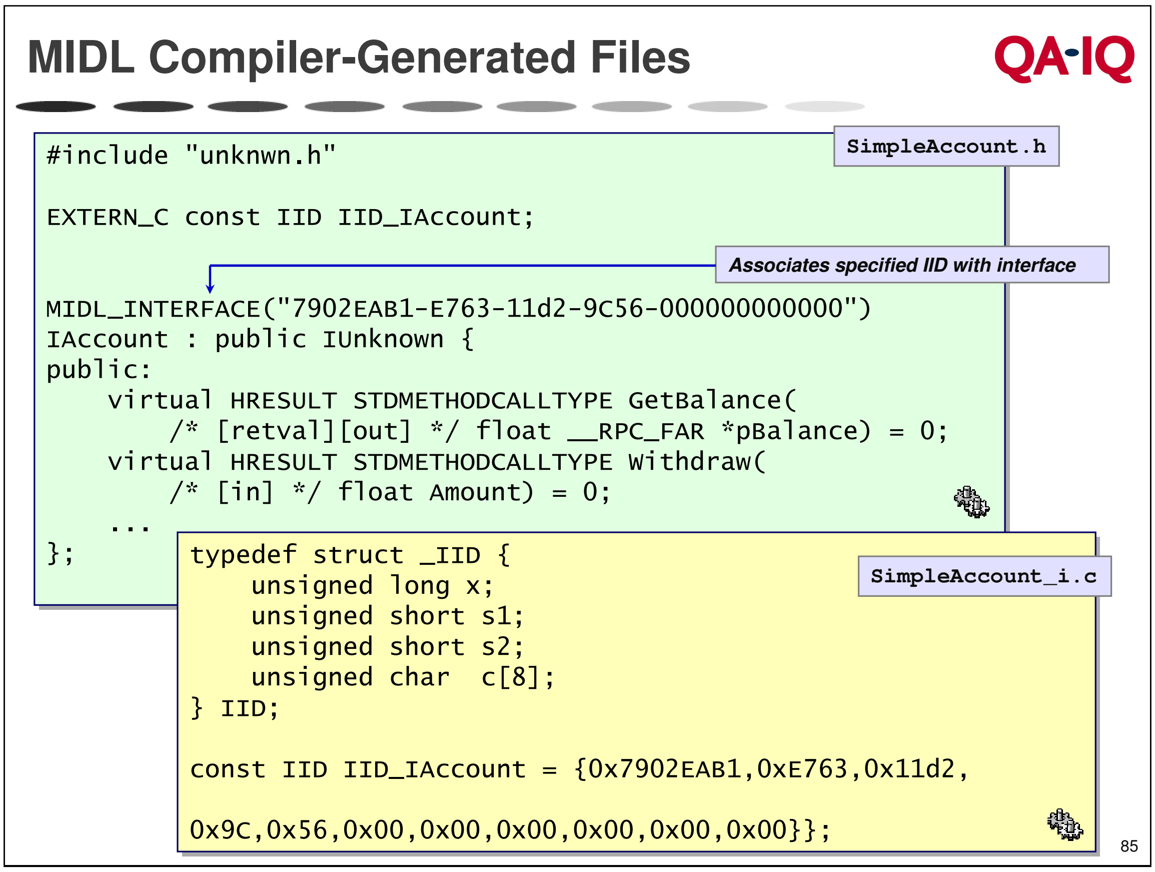
Proxy DLL file: contains code required to put proxy and stub code into a DLL

Type library file: contains binary tokenised form of IDL file. Only generated if IDL file contains a `library` statement

84

The MIDL compiler generates a number of files as shown on the slide. We will discuss the header (`XXX.h`), the interface GUID file (`XXX_i.c`) and the type-library file (`XXX.tlb`) later in the chapter. The interface proxy file (`XXX_p.c`) and the proxy DLL file (`dlldata.c`) are required only to support marshalling, so we will delay any further discussion of these files until the *COM EXE Servers* chapter.

It should be pointed out that the MIDL compiler produces very impressive code, but it is a little shy with its output diagnostics (if something's wrong, it might just tell you that there is one error, but provide no other information).



MIDL Compiler-Generated Files

SimpleAccount.h

```
#include "unknwn.h"

EXTERN_C const IID IID_IAccount;

MIDL_INTERFACE("7902EAB1-E763-11d2-9C56-000000000000")
IAccount : public IUnknown {
public:
    virtual HRESULT STDMETHODCALLTYPE GetBalance(
        /* [retval][out] */ float __RPC_FAR *pBalance) = 0;
    virtual HRESULT STDMETHODCALLTYPE Withdraw(
        /* [in] */ float Amount) = 0;
    ...
};
```

Associates specified IID with interface

SimpleAccount_i.c

```
typedef struct _IID {
    unsigned long x;
    unsigned short s1;
    unsigned short s2;
    unsigned char  c[8];
} IID;

const IID IID_IAccount = {0x7902EAB1,0xE763,0x11d2,

0x9C,0x56,0x00,0x00,0x00,0x00,0x00,0x00}};
```

85

When the file `SimpleAccount.idl` (shown on page13) is passed to the MIDL compiler, it generates a header file `SimpleAccount.h` and an interface GUID file `SimpleAccount_i.c` as shown above (after some editing for length).

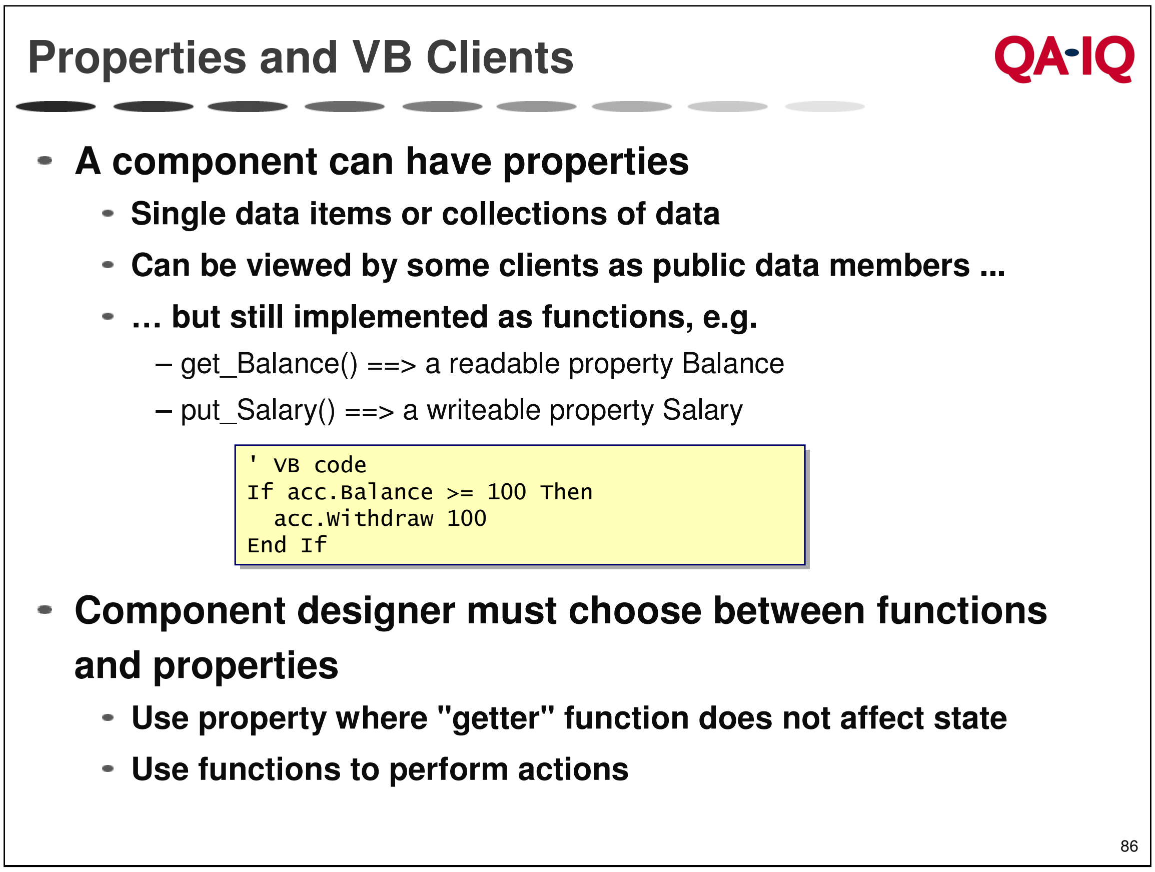
`SimpleAccount.h` contains a `#include "unknwn.h"`, which results from the `import unknwn.idl` statement in the IDL file. Next, comes an EXTERN_C declaration for the IID of the `IAccount` interface,which is defined in `SimpleAccount_i.c`. Note that the name of this IID is of the form `IID_Ixxx`, where `xxx` is the name of the interface.

The MIDL compiler supplied with Visual C++ 6.0 uses the MIDL_INTERFACE() macro to define an interface. This macro is defined in `rpcndr.h` as:

`struct __declspec(uuid(`*`<uuid>`*`)) __declspec(novtable)`

Basically, this uses the `__declspec(uuid(`*`<uuid>`*`))` declarator to attach a GUID to the structure that defines an RPC or COM interface. In this example, it attaches the IID specified in the IDL file to the structure that defines the `IAccount` interface. As we will see in the *Language Integration* chapter, the attached IID can subsequently be retrieved from an interface using the `__uuidof()` keyword.

The generated definition of the `IAccount` interface looks pretty much as you would expect. It simply defines the functions specified in the IDL as pure virtual functions. Notice how the parameter attributes are included inside comments as a reminder of their usage.



Properties and VB Clients

QA·IQ

- **A component can have properties**
  - **Single data items or collections of data**
  - **Can be viewed by some clients as public data members ...**
  - **... but still implemented as functions, e.g.**
    - get_Balance() ==> a readable property Balance
    - put_Salary() ==> a writeable property Salary

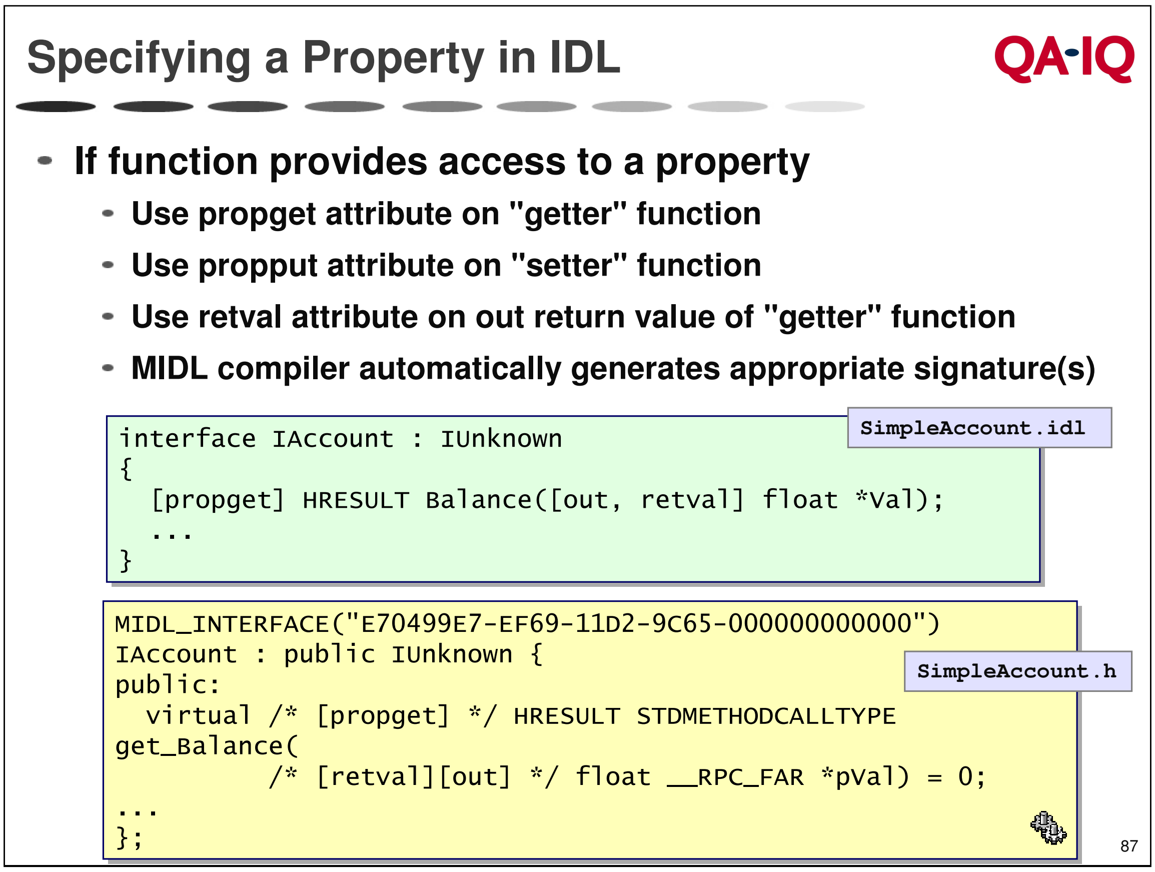
```
' VB code
If acc.Balance >= 100 Then
  acc.Withdraw 100
End If
```

- **Component designer must choose between functions and properties**
  - **Use property where "getter" function does not affect state**
  - **Use functions to perform actions**

86

A component can have properties, which can be single data items or collections of data items. A property `XXX` is implied by the existence of interface functions of the form `get_XXX()` and `put_XXX()`. If both functions are specified, the property is both readable and writeable, but if only a "getter" functions is specified, the property is read-only (as viewed by the client).

The motivation for specifying component properties is that Visual Basic and scripting clients can view them as public data members of the component. However, when designing an interface, it can sometimes be difficult to decide when to use properties and when to use methods. The basic rule is that a property is appropriate where a "getter" function would not affect the state of the component, whereas functions should be used to perform some action on the component.



Specifying a Property in IDL

QA-IQ

- If function provides access to a property
  - Use propget attribute on "getter" function
  - Use propput attribute on "setter" function
  - Use retval attribute on out return value of "getter" function
  - MIDL compiler automatically generates appropriate signature(s)

SimpleAccount.idl

```
interface IAccount : IUnknown
{
  [propget] HRESULT Balance([out, retval] float *Val);
  ...
}
```

SimpleAccount.h

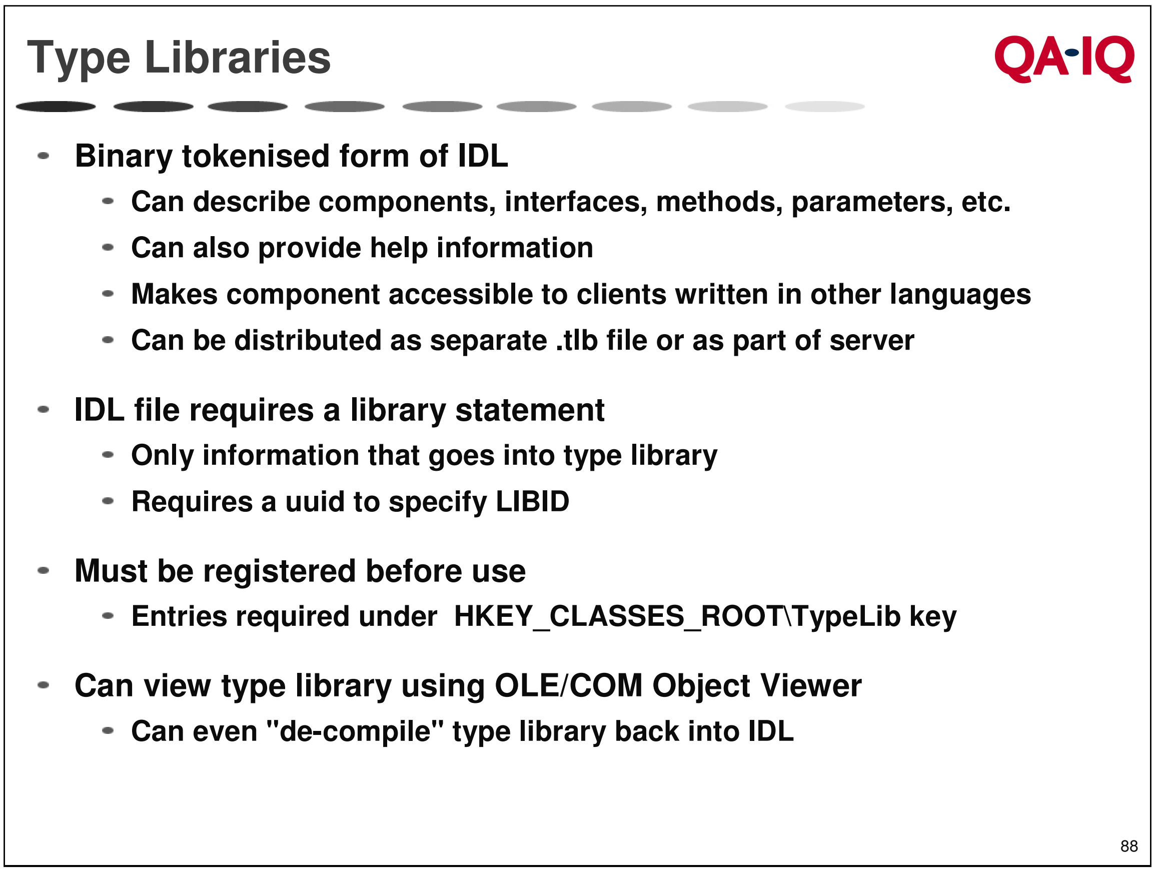
```
MIDL_INTERFACE("E70499E7-EF69-11D2-9C65-000000000000")
IAccount : public IUnknown {
public:
  virtual /* [propget] */ HRESULT STDMETHODCALLTYPE
get_Balance(
          /* [retval][out] */ float __RPC_FAR *pVal) = 0;
...
};
```

87

Properties can be specified in IDL using the [propget] and [propput] attributes.

As shown on the slide, a read-only property is declared in a similar way to a function except that the property name is substituted for the function name, the function **must** have a single [out, retval] parameter of the appropriate type, and the declaration must be preceded by the [propget] attribute. If the property is read/write, a second property declaration is required, which should be identical to the first except that it should have an [in] parameter, and a [propput] instead of a [propget] attribute. Properties can be made write only by solely providing a [propput] property.

The MIDL compiler automatically generates C++ setter and/or getter functions with the appropriate signatures. For example, if the IDL specifies a [propget] property called Balance, the generated interface-definition will contain a get_Balance() function, as shown on the slide. Conversely, if there was a [propput] property Balance, MIDL would generate a put_Balance() function in the C++ header file.



Type Libraries

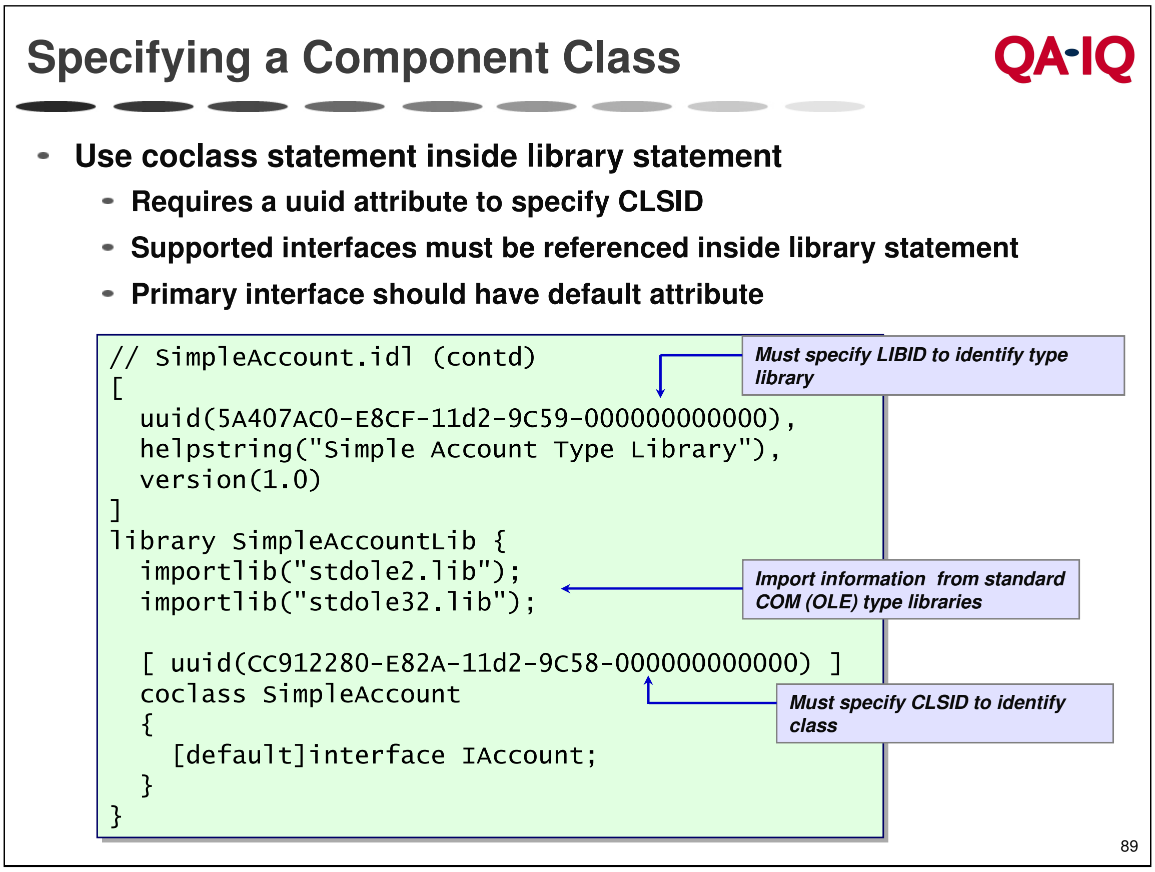
- Binary tokenised form of IDL
  - Can describe components, interfaces, methods, parameters, etc.
  - Can also provide help information
  - Makes component accessible to clients written in other languages
  - Can be distributed as separate .tlb file or as part of server
- IDL file requires a library statement
  - Only information that goes into type library
  - Requires a uuid to specify LIBID
- Must be registered before use
  - Entries required under HKEY_CLASSES_ROOT\TypeLib key
- Can view type library using OLE/COM Object Viewer
  - Can even "de-compile" type library back into IDL

88

Type libraries were originally introduced to support Automation. A type library is a binary file that provides type information about components, interfaces, methods, properties, parameters, return types and structures. Basically, it is a binary tokenised form of an IDL file that can accessed programmatically via standard interfaces, such as `ITypeLib` and `ITypeInfo`.

As we will see in the *Automation* and *Language Integration* chapters, without access to a type library, a Visual Basic client is limited to using a component through a standard interface known as `IDispatch`. On the other hand, provided it has access to a component's type library, a Visual Basic 5.0 or 6.0 client can access that component through any of its custom or standard interfaces (i.e. though a vtable).

The only information in an IDL file that goes into a type library is that specified in a `library` statement. Typically, a `library` statement contains a `coclass` statement, which specifies the interface(s) implemented by a particular component class. So even though we are not really interested in type libraries yet, we must provide a `library` statement in an IDL file in order to specify a component class.



Specifying a Component Class

QA-IQ

- Use coclass statement inside library statement
  - Requires a uuid attribute to specify CLSID
  - Supported interfaces must be referenced inside library statement
  - Primary interface should have default attribute

```
// SimpleAccount.idl (contd)
[
  uuid(5A407AC0-E8CF-11d2-9C59-000000000000),
  helpstring("Simple Account Type Library"),
  version(1.0)
]
library SimpleAccountLib {
  importlib("stdole2.lib");
  importlib("stdole32.lib");

  [ uuid(CC912280-E82A-11d2-9C58-000000000000) ]
  coclass SimpleAccount
  {
    [default]interface IAccount;
  }
}
```

Must specify LIBID to identify type library

Import information from standard COM (OLE) type libraries

Must specify CLSID to identify class

89

Like an `interface` statement, a `library` statement requires a `uuid()` attribute to specify the LIBID of the type library. A `helpstring()` attribute can also be provided; the specified string can be displayed by object browsers. By convention, a `version` attribute should also be provided, but note that this specifies the version of the type library, not the version of the component.

The library statement requires a name. In the example shown on the slide, we've used the same convention as ATL, which is to append "Lib" to the `coclass` name.

Inside the `library` statement block, you can use the `importlib()` statement to import information from another type library. Typically, the standard COM (OLE) type libraries, `stdole2.lib` and `stdole32.lib`, are imported as shown in the example on the slide. A component class is specified using a `coclass` statement. Again, this requires a `uuid()` attribute to specify the CLSID of the component.

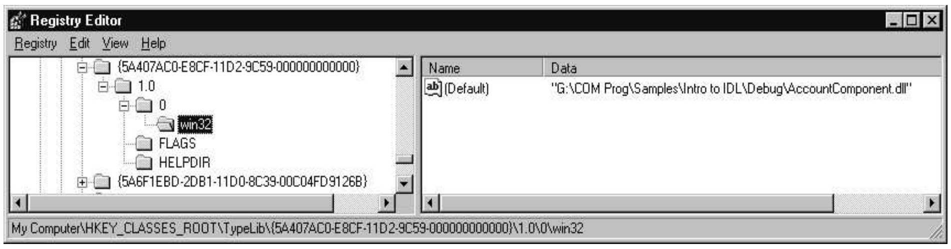
Inside the `coclass` statement block, you must specify which interfaces are implemented by the component. If you specify more than one interface, the primary interface should be tagged with the `[default]` attribute. This will be the primary interface as viewed by Visual Basic clients.

When passed to the MIDL compiler, all items declared or *referenced* in a library statement will be included in the generated type library. So, in the case of our simple example, information about both the interface `IAccount` and the component class `SimpleAccount` will be included in the type library `SimpleAccount.tlb`.

Registering a Type Library

QA-IQ

- In DLL server, modify DllRegisterServer()...
  - Call LoadTypeLibEx() to register type library
- ... and DllUnregisterServer()
  - Call UnRegisterTypeLib()
- Alternatively, prepare a .reg file for use with RegEdit
  - Several entries required under HKEY_CLASSES_ROOT\TypeLib key



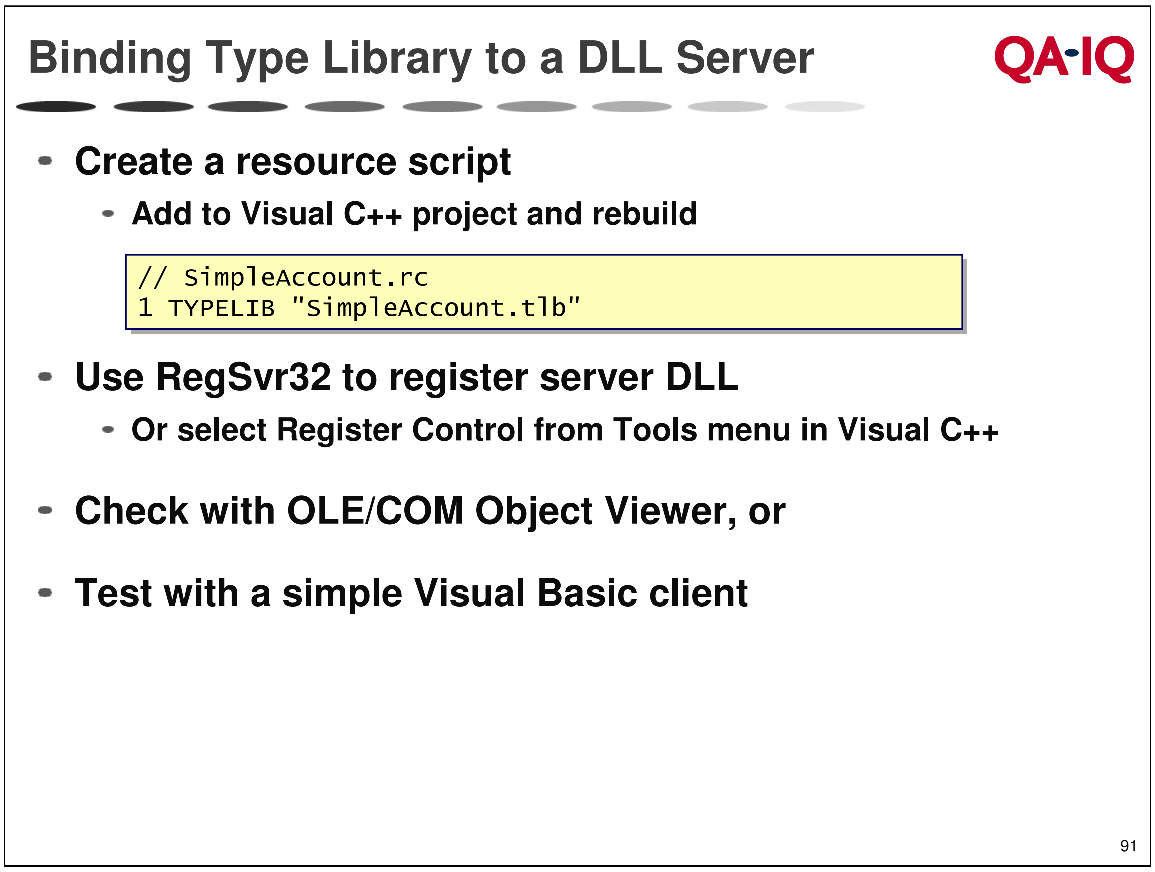
```
{5A407AC0-E8CF-11D2-9C59-000000000000}
  1.0
    0
      win32
    FLAGS
    HELPDIR
{5A6F1EBD-2DB1-11D0-8C39-00C04FD9126B}
```

Name	Data
(Default)	"G:\COM Prog\Samples\Intro to IDL\Debug\AccountComponent.dll"

My Computer\HKEY_CLASSES_ROOT\TypeLib\{5A407AC0-E8CF-11D2-9C59-000000000000}\1.0\0\win32

90

Before you can use a type library, it must be registered in the Registry. This means that several entries must be inserted under the `HKEY_CLASSES_ROOT\TypeLib` key. Programmatically, this is not as difficult as it sounds, because it can be achieved by the addition of a single call to `LoadTypeLibEx()` inside `DllRegisterServer()`.



Binding Type Library to a DLL Server

QA-IQ

- Create a resource script
  - Add to Visual C++ project and rebuild

```
// SimpleAccount.rc
1 TYPELIB "SimpleAccount.tlb"
```

- Use RegSvr32 to register server DLL
  - Or select Register Control from Tools menu in Visual C++
- Check with OLE/COM Object Viewer, or
- Test with a simple Visual Basic client

91

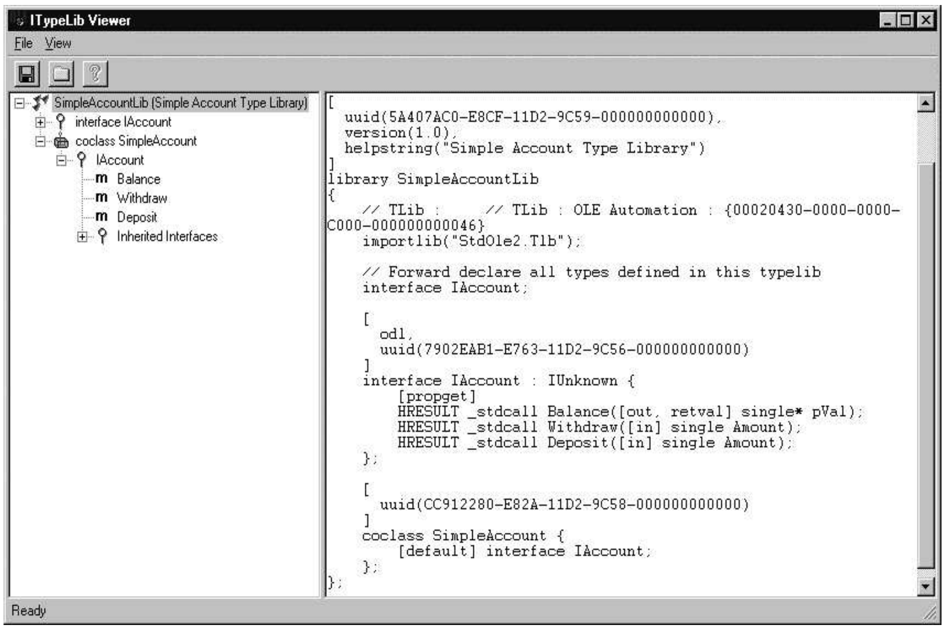
Once generated, a type library can be maintained either as a separate `.tlb` file, or it can be bound to the server DLL as a resource. The latter is much cleaner as only one file needs to be distributed to users of your component.

As shown on the slide, defining a type-library resource is very easy. If you add this file to your project, the resource compiler will automatically bind the type-library resource to the server DLL.

Using the OLE/COM Object Viewer

QA-IQ

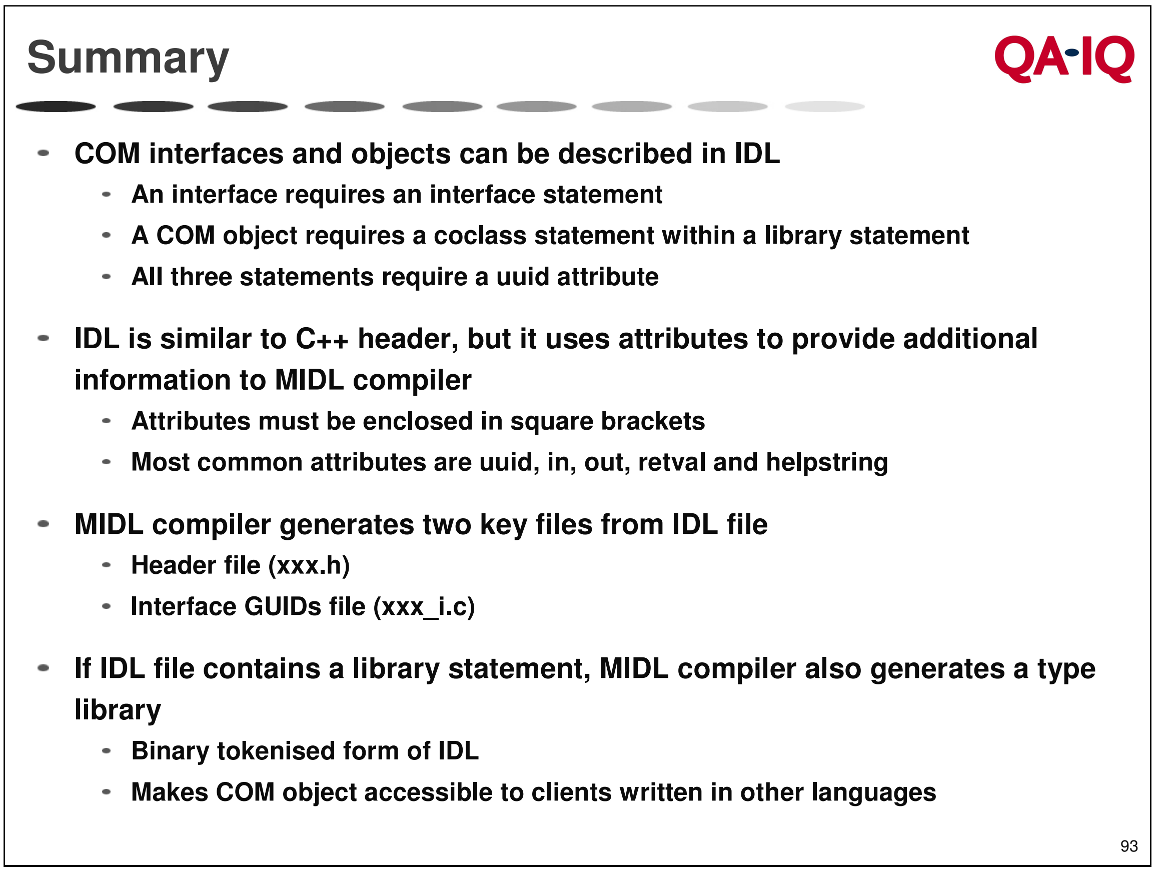
- **Can view type library and corresponding IDL**



```
[
  uuid(5A407AC0-E8CF-11D2-9C59-000000000000),
  version(1.0),
  helpstring("Simple Account Type Library")
]
library SimpleAccountLib
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-
C000-000000000046}
    importlib("StdOle2.Tlb");

    // Forward declare all types defined in this typelib
    interface IAccount;

    [
      odl,
      uuid(7902EAB1-E763-11D2-9C56-000000000000)
    ]
    interface IAccount : IUnknown {
        [propget]
        HRESULT _stdcall Balance([out, retval] single* pVal);
        HRESULT _stdcall Withdraw([in] single Amount);
        HRESULT _stdcall Deposit([in] single Amount);
    };

    [
      uuid(CC912280-E82A-11D2-9C58-000000000000)
    ]
    coclass SimpleAccount {
        [default] interface IAccount;
    };
};
```

92

A simple way to verify the information in a type library is to use the OLE/COM Object Viewer. If you expand the *Type Libraries* folder in the left-hand pane, you should find an entry for your type library. Clicking on this entry causes the type-library information to be displayed in the right-hand pane. Double-clicking on this entry brings up the *ITypeLib Viewer* as shown on the slide. Notice that the right-hand pane shows the IDL that the *OLE/COM Object Viewer* has generated by "de-compiling" the type library.



Summary

QA-IQ

- COM interfaces and objects can be described in IDL
  - An interface requires an interface statement
  - A COM object requires a coclass statement within a library statement
  - All three statements require a uuid attribute
- IDL is similar to C++ header, but it uses attributes to provide additional information to MIDL compiler
  - Attributes must be enclosed in square brackets
  - Most common attributes are uuid, in, out, retval and helpstring
- MIDL compiler generates two key files from IDL file
  - Header file (xxx.h)
  - Interface GUIDs file (xxx_i.c)
- If IDL file contains a library statement, MIDL compiler also generates a type library
  - Binary tokenised form of IDL
  - Makes COM object accessible to clients written in other languages

93