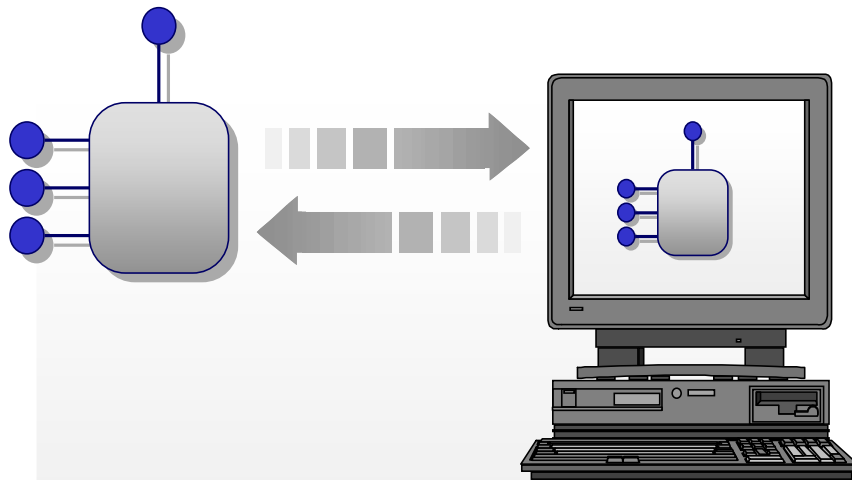


## From C++ to COM

QA-IQ



9

## Chapter Overview



- **Objective**
  - Develop a simple model for implementing reusable components in C++
- **Chapter content**
  - Component based development
  - Problems with traditional C++ class libraries
  - Separating a component's interface from its implementation
  - Using an abstract base class
  - Use of virtual function tables
  - Run-time discovery of a component's interfaces
  - Eliminating compiler dependencies
  - Managing the lifetime of a component
  - Reference counting
- **Practical content**
  - Develop and test a simple reusable component using C++
- **Summary**

10

---

In this chapter, we will review the limitations of traditional C++ class libraries and develop a model for developing reusable components.

## Component Based Development



- The motives...
- We want to build dynamically composable systems
  - Not monolithic statically linked applications
- We want to minimise coupling within the system
  - One change propagates to entire source code tree
- We want plug-and-play replaceability and extensibility
  - New pieces should be indistinguishable from old, known parts
- We want implementation language neutrality
- We want freedom from file/path dependencies
- We want components with different runtime requirements to live peaceably together
  - Need to mix heterogeneous objects in a single process
  - Language independence

11

One of the great motivating factors for object oriented languages, such as C++, was their ability to create reusable code bases for building large applications. In many ways, C++ has proven very successful at this, with libraries such as MFC enabling us to write great Windows applications both quickly and reliably.

However, C++ and other OO languages are not enough in and of themselves to allow us to write great component based systems, as opposed to monolithic applications.

Typically, we now want to be able to develop reusable components that can be brought together dynamically at runtime. These components can then be used from many different programming environments, completely independent of the implementation language that is being used. To do this, we have to minimise the coupling between components; otherwise they cannot be reused, deployed or modified without rebuilding the entire code base.

Obvious examples of this are units of business logic that are used both in Intranet/Internet applications as well as from rich Windows programs, and might therefore be called from scripting languages, Visual Basic or C++.

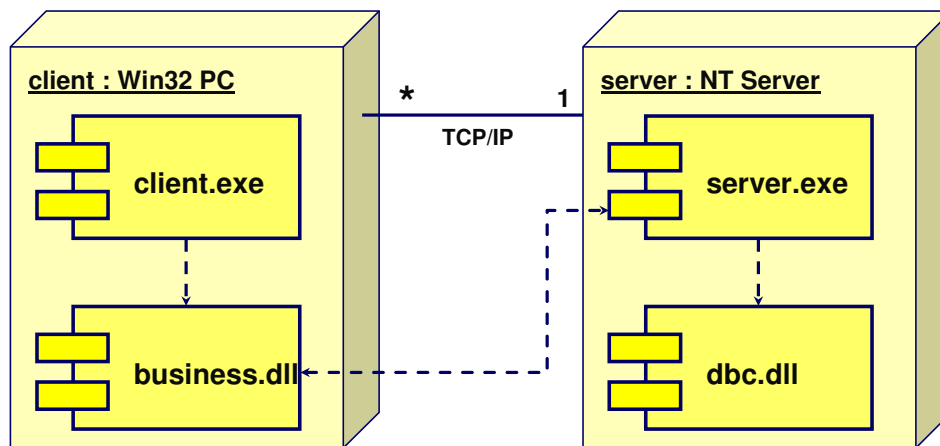
We also have to think about how we might want to deploy our components. One of the aspects of a component system, such as COM or CORBA, is the runtime environment that provides a mechanism for locating and running components. This overcomes the need to explicitly bind to a known DLL, offering both dynamic composition as well as location transparency.

Finally, we need to consider the typical consumer of our components. Will they support multiple threads of execution? Will they be programming in English, French or any other language? Are they writing in VB or Java? Our components should be made accessible to them without them having to know about the internal implementation details.

## Component Deployment



- **Components can provide both location and implementation transparency**
  - Can be located in the same process as the component user, in different processes on the same host, or on different hosts



12

Components conform to a component model, such as COM or CORBA, which defines the way that component interfaces are defined, constrains their implementation, and offers a mechanism for interconnection. In short, a component model is a framework on which systems architects build. Connectivity between components using the same middleware is assured; for connecting together different component models bridges are often used, e.g. the CORBA/COM bridge specified by the OMG.

Part of the intent of partitioning a system into components is to support location transparency: given an interface to functionality, having to understand and locate the component implementation should not be an issue to the component user.

As a piece of software, the user code must somehow link to a component. This may be done within the same program using static linking through traditional libraries, e.g. .lib and .a files. However, it is more common – and indeed, more in line with the principle of components – to bind to a component at runtime. Within the same process these are dynamically-linked libraries, e.g. .dll files. These are sometimes known as in-process servers. Alternatively a component may run separately as a local process. The final option is that a component may run as a process on a separate host.

## So how about C++?



- **C++ is a language and not a binary standard**
  - Name mangling, exception handling
  - All components must be developed with same compiler
- **Poor support for other languages**
  - Exposing 'C' functions!
- **Produces closely coupled systems**
  - class used for both definition and implementation
  - Changing private data items forces a rebuild
- **No location transparency / file path independence**
  - No implicit runtime support
- **Consequently, we need to think imaginatively to do components (and we need runtime support!)**

13

C++ itself is just a programming language (albeit a very fine language!).

If we take a closer look at C++, we see that it consists of a source code language standard, resulting in different C++ compiler vendors producing incompatible code for such items as name mangling (required for function overloading) and exception handling.

This would force all C++ code to be compiled with the same compiler! This doesn't exactly sound like the flexible, reusable mechanism that we are looking for when we are writing components.

C++ also suffers from the fact that the traditional way of distributing C++ classes involves shipping a library and a set of header files. The headers contain the class definitions, which makes them vulnerable to editing (a client programmer can mark private items public!).

It also means that the systems become closely coupled, as a change inside a class as simple as adding a member variable can break existing client code, forcing a rippling build through the entire code base.

## COM to the Rescue!

QA-IQ

- **So what is COM?**

**"A programming discipline, combined with a supporting runtime infrastructure, which together facilitate the development of component-based software systems"**

**In other words:**

### **Microsoft's component based technology**

- **Don't be confused by terms such as OLE or ActiveX**
  - **These are COM-based technologies directed at particular problem domains**
  - **It's all just COM!**

14

Microsoft understood the need for us to be able to write systems based on components. To this end, they provided us with COM, or to use its full name the Component Object Model.

The high-level answer to the simple question "So what is COM?" is given above.

It is a programming discipline in the sense that certain rules and procedures must be adopted and adhered to by developers - or it just won't work. The COM specification lays down these rules and procedures. Secondly, a supporting runtime infrastructure often referred to as the "COM plumbing" is required at run time to support COM's operations.

It should be noted that COM is Microsoft's implementation of a component based system. Others exist, such as OMG CORBA and Sun Microsystems' EJB. COM is available on all Windows versions, and has limited support on some other platforms.

A lot of the early literature mixed COM with OLE - originally object linking and embedding technology, which unfortunately led to the COM's reputation for being complex and incomprehensible. COM is the underlying technology upon which OLE and other related technologies such as ActiveX are based. Like most technologies, they are best learnt from the bottom up. With a solid understanding of COM, technologies such as OLE and ActiveX will fall into place.

## And COM+ ?

QA-IQ

- **Extends the basic component object interaction model**
  - **Builds on MTS**
  - **Declarative use of services where possible**
  - **True object pooling**
  - **Activators and interceptors**
- **Defines new services for COM components**

### *New COM+ Services*

- ➔ Dynamic load balancing
- ➔ Events (publish/subscribe)
- ➔ Queued components (Asynchronous method calls using MSMQ)
- ➔ Improved security
- ➔ Extended transaction support



15

COM+ builds on the existing COM framework in the same way that MTS does. COM+ is an evolution of MTS bringing the same form of declarative and assisted programming to other, non-transaction-related services. The slide lists the services made available to components by the COM+ runtime. To take advantage of these services, a component must submit itself to the COM+ runtime such that all calls into and out of the component can be monitored and "enhanced" with the requested services. This requires that the component is housed in a DLL rather than an EXE.

COM+ is delivered with Windows 2000. Other improvements have been made in Windows 2000 to COM in general. One of these is the provision of a new threading model, the neutral model, which allows for true object pooling in COM+ (this is not available in MTS, despite the presence of the CanBePooled() API function). Other improvements have allowed for extended distributed security and broader transaction support.

COM+ also provides a generalised event tracking and handling architecture in the form of interceptors, activators and policy makers. These provide hooks for handlers that can monitor or interact with method invocations. COM+ uses these to propagate "invisible" context information such as transaction context. Currently this mechanism is used by COM+ itself, and is not made available to developers in general, but this may change in the next release.

## Migrating from C++ to COM



- How do we migrate the following class?

```
// Account.h
class Account {
    float m_balance;
public:
    Account();
    ~Account();
    void Deposit(float amount);
    void Withdraw(float amount);
    float GetBalance() {return m_balance;}
};
```

16

Here is a typical class that we might have written as a traditional C++ programmer. Its implementation code would have been placed into a .cpp file, and built as a library (either static or dynamic). We would typically have shipped both the header and the library to the customer, and as long as they were working with exactly the same compiler, they could have used the class.

However, we want to support a broader client base, remove compiler independence and work as a proper component. After all, the user of this code might be working in Visual Basic!

So let's start the migration process to COM...

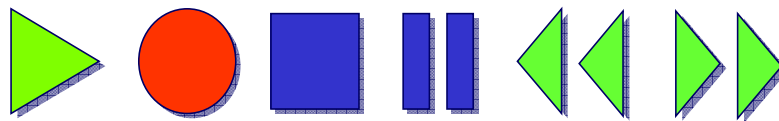


## Interface Based Programming

QA-IQ

- **Separating implementation from definition of behaviour**
- **What is an interface?**
  - **A collection of logically related functions**
  - **Immutable after it has been defined**

**The most recognisable interface on the planet?**



17

Our first step to working with components has to be separating the implementation code from how we define what an object actually does. To do this separation, we need to define the interfaces to the object. So what is an interface?

In reality, an interface is nothing more than a logically related group of functions. Looking at the function definitions enables us to see what the object will do, although we have no idea how the object does it. We can then merely treat any object as something that supports a set of interfaces.

For robustness, though, we must specify that an interface cannot be changed after it has been defined. This ensures that clients can continue to work with objects, even if something about the object changes internally.

As an example of an interface we have a series of simple shapes. These shapes might arguably represent the most recognisable interface on the planet, as they are used on virtually all tape recorders, videos, CD and DVD players. All the shapes combined represent the group of functions that can be performed on the relevant media.

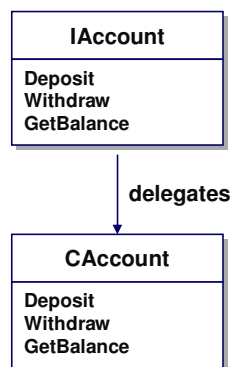
Each shape has well known and unchanging functionality. For example, we might have our prized video of a child's first steps, and we know that pressing the big green arrow (the play button) is not going to record Eastenders over the top of it.

But the important thing is that we do not need to know how the video recorder, CD or DVD player actually implements the interface. Therefore, we are capable of using newer models without changing our client perception of the interface.

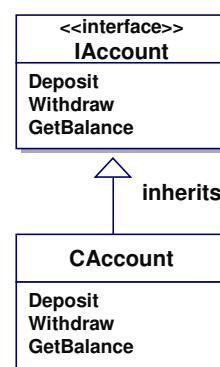
## Decoupling the Interface

- **Need two classes ...**
  - One to represent interface
  - Other to represent implementation
- ... plus some association between the two
- **Two ways to achieve this:**

*A. Use handle class for interface*



*B. Use abstract base class for interface*



18

To provide binary encapsulation with C++, we need to separate the public interface of a reusable class (the component) from its internal implementation. The interface should define the contract (a list of function signatures) between a client application and the component. This means that, provided the public interface isn't changed, client applications will be unaffected by changes to the implementation. Furthermore, once defined, a public interface can be implemented in many different ways, so providing clients with a high degree of polymorphism.

Clearly, we need two classes: one to represent the interface and another to represent an implementation of that interface. As shown on the slide, this could be achieved either by using a handle class or by using an abstract base class.

A *handle* class (c.f. a *tie* class in CORBA) is a trivial class that has no member variables; it simply delegates calls to its member functions to member functions of the same name in an implementation class. However, client applications must still link to symbolic function names, so this approach is limited because it doesn't solve the name-mangling problem.

The alternative is to use an abstract base class. This class simply contains one or more virtual functions that define the interface. An implementation class, which inherits from this abstract class, must therefore override (or *implement*) each of these virtual functions. As we will see later, this approach also solves the name-mangling problem.

## Using an Abstract Base Class

- Define interface in abstract base class

```
// IAccount.h
struct IAccount
{
    virtual void Deposit(float amount) = 0;
    virtual void Withdraw(float amount) = 0;
    virtual float GetBalance() = 0;
};
```

*IAccount*  
interface

- Implement interface in a derived class

```
// CAccount.h
class CAccount : public IAccount {
    float m_balance;
public:
    Account();
    ~Account();
    void Deposit(float amount);
    void Withdraw(float amount);
    float GetBalance() {return m_balance;}
};
```

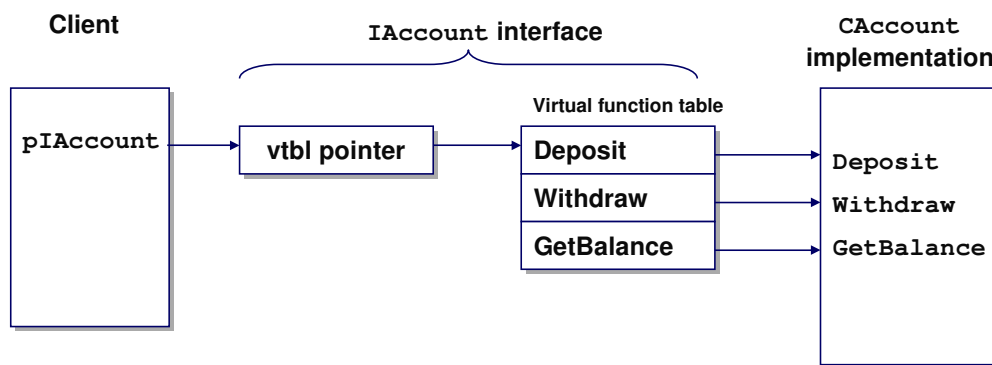
*CAccount*  
implementation

19

Here is an example of how we could use an abstract base class to separate the interface of an `Account` component from its implementation. Notice that the abstract base class, `IAccount`, contains nothing more than the three pure virtual functions that define the public interface of an `Account` component. The implementation class, `CAccount`, derives from `IAccount`, and will therefore need to override (or *implement*) each of these virtual functions.

## "Vtable" Mechanism

- Virtual-function table implemented by C++ compilers
- Key to binary compatibility
- Provides level of indirection
- Fixes name-mangling problem



20

The virtual function table (usually referred to as a *vtable* or *vtbl*) mechanism is the key to solving the C++ name-mangling problem and therefore to providing binary compatibility. A vtable is simply an array of pointers to virtual functions; it is used by most C++ compilers to implement polymorphic calls to virtual functions (i.e. late binding).

In essence, each instance of a class has a single hidden member-variable (a *vtbl pointer*) that is initialised by a constructor to point to the class' vtable. Furthermore, this vtable is inherited by all derived classes. For example, our `CAccount` class, which is derived from the `IAccount` class, has a vtable in which the first three entries are pointers to the `Deposit()`, `Withdraw()` and `GetBalance()` functions, respectively. So, if a client obtains a pointer `pIAccount` to the `IAccount` interface of *any* instance of a class derived from `IAccount`, it is a pointer to the class' vtable and not to an instance of an implementation class (all instances share the same vtable). If the client subsequently uses `pIAccount` to make a call to, for example, the `GetBalance()` function, this call is made indirectly through the vtable. The client doesn't need to link to the symbolic name of the `GetBalance()` or any other function of the `IAccount` interface, so name-mangling is no longer an issue.

One thing that is obviously important to binary compatibility is the layout of the vtable itself. Currently, there are two de facto standards: one is specified by CFRONT (as used by Solaris) and the other is specified by Microsoft (for Win32 compilers). Therefore, on any given platform we can assume that all compilers will implement vtables in the same way. Two other restrictions are (a) an abstract base class must not have any member variables, and (b) an abstract base class can have at most one base class. The second point will become more important when we look at extending interfaces.

## Client Code



- **Problem is that client uses new operator**
  - Only supports C++ clients
  - Client developer needs implementation-class definition
  - Gives client access to complete object, not just to its interface

```
// Client.cpp
#include "CAccount.h"    // implementation header!

int main() {
    CAccount *pCAccount = new CAccount(100.00);

    // get an IAccount pointer
    IAccount *pIAccount = pCAccount;

    // perform some operations on the Account object
    pIAccount->Deposit(50.00);
    float balance = pIAccount->GetBalance();
    cout << "Final balance is " << balance << endl;
    ...
}
```

21

Here is an example of how we might implement a client of our `Account` component. After instantiating a `CAccount` object, we simply obtain a pointer to its `IAccount` interface and then call a couple of functions through this interface. On closer inspection though, you can see that we've used the `new` operator to instantiate a `CAccount`, which means that to compile this code, we need the actual class definition of `CAccount`. This gives the client access to the implementation of `CAccount` through `pCAccount`, which is precisely what we're trying to avoid!

Therefore, what we need to do is abstract away the creation process from the client.

We'll see how to do that on the next page.

## Restricting Client Access

- Provide global CreateXXX() wrapper function
  - Calls new on client's behalf
  - Returns a pointer to object's interface, not its implementation

```
// IAccount.h
class IAccount {
public:
    virtual void Deposit(float amount) = 0;
    ...
};

extern "C" IAccount *CreateAccount();
```

IAccount interface

Allows linking  
to "C" clients

```
// CAccount.cpp
...

IAccount *CreateAccount() {
    return new CAccount;
}
```

CAccount implementation

22

To restrict client access to an implementation object's interface, the implementation code must provide a global wrapper function that instantiates the object on the client's behalf, and returns a pointer to the object's *interface*, not to the object itself.

For example, in the CAccount implementation code, we could provide a *global* CreateAccount() function that simply creates a CAccount object and returns a pointer to its IAccount interface. Client code must link symbolically to CreateAccount(), so to prevent name mangling, we've declared it as extern "C" in the IAccount header. This will also allow "C" clients to instantiate (and use) any object that implements the IAccount interface.

It should be noted that the abstraction away of the creation process allows the developer to perform interesting tricks behind the scenes.

For example, it would now be possible to implement a singleton object by simply returning a pointer to a global variable from the CreateAccount() function. Care has to be taken here though, because of the semantic implication that CreateAccount() makes a new account object.

## Object Destruction

- **Must also provide matching Delete() function**
  - Another virtual function in each interface
  - Calls delete on client's behalf

```
// IAccount.h
class IAccount {
public:
    virtual void Deposit(float amount) = 0;
    ...
    virtual void Delete() = 0;
};

extern "C" IAccount *CreateAccount();
```

*IAccount interface*

```
// CAccount.cpp
...

void CAccount::Delete() {
    delete this;
}
```

*CAccount  
implementation*

23

If the implementation class is going to provide a `CreateXXX()` function, it must also provide a matching `Delete()` function to delete the object on the client's behalf. Having created an object of the implementation class, the client has a pointer to the object's interface, so the `Delete()` function can be part of this interface (i.e. a pure virtual function).

The implementation of a `Delete()` method is very straightforward as shown on the slide.

## Revised Client Code

- Client can only access object through its interface
- Client developer doesn't need definition of implementation class
- Client could be written in another language

```
// Client.cpp
#include "IAccount.h"    // interface header only

int main() {

    // get an IAccount pointer
    IAccount *pIAccount = CreateAccount(100.00);

    if (pIAccount) {
        // perform some operations on the Account object
        pIAccount->Deposit(50.00);
        float balance = pIAccount->GetBalance();
        ...
        pIAccount->Delete(); // finished with IAccount
    }
    ...
}
```

24

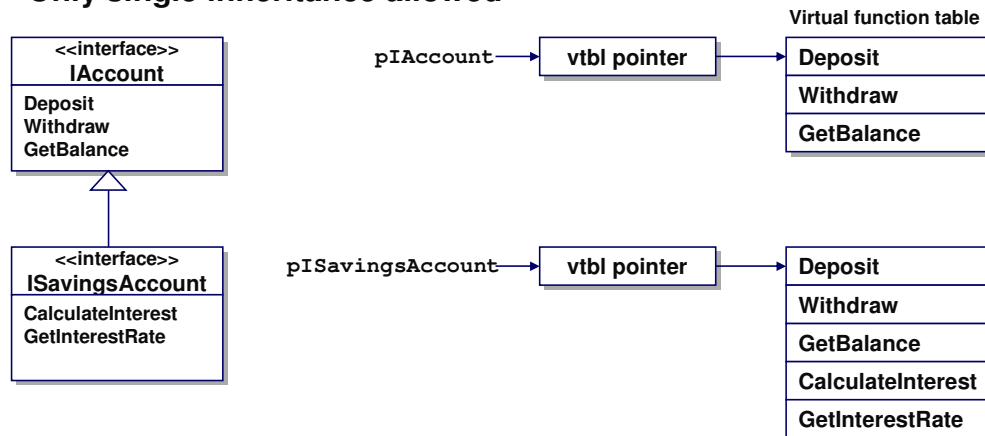
Having modified the `IAccount` interface and the `CAccount` implementation code, we can now revise the client code as shown on the slide. Notice that `CreateAccount()` returns an `IAccount` pointer so the client can only access a `CAccount` object only through this interface. Also, to compile this code, we no longer need the class definition of `CAccount`.



## Extending an Interface



- **An interface is immutable**
  - A binary and semantic contract
  - Once published, cannot change signatures or semantics
- **But, a new interface can inherit from another interface**
  - Only single inheritance allowed



25

Once defined, we must consider an interface to be immutable; otherwise, we risk breaking existing client applications that use that interface. After all, an interface defines a binary and semantic contract between implementations of that interface and client applications.

So, as component developers, we must resist the temptation to "tweak" a published interface. Instead, we need to define a new interface, which according to our model, means defining a new abstract class. However, because an interface is just a pure abstract class, we can use inheritance to extend an existing class. For example, to define a new **ISavingsAccount** interface, we can inherit from the existing **IAccount** interface, as shown on the slide.

To implement the **ISavingsAccount** interface, we simply inherit from the **ISavingsAccount** base class, which of course means that we'll need to override the virtual functions of both the **IAccount** and **ISavingsAccount** base classes.

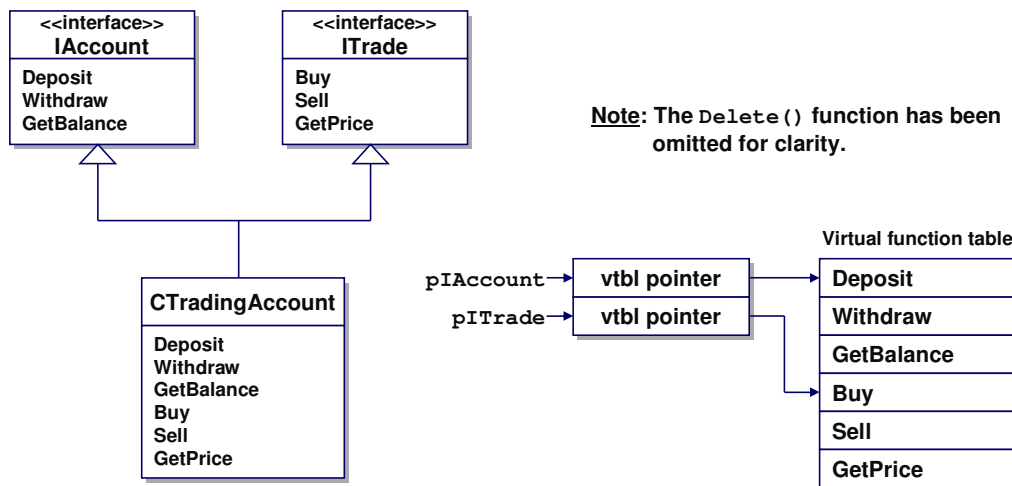
As previously mentioned, a derived class inherits the vtable layout of its base class, which means that the vtable of a class that implements the **ISavingsAccount** interface will contain five entries as shown on the slide.

Therefore, if a client obtains a pointer to an implementation's **ISavingsAccount** interface, it can call any of the functions of the **IAccount** or **ISavingsAccount** interfaces through that pointer. Conversely, if a client is only able to obtain a pointer to the same implementation's **IAccount** interface, it can only call the functions defined in the **IAccount** interface.

Finally, it is worth noting that although C++ supports multiple inheritance, to retain the standard layout of the vtable, we must not attempt to derive an interface from two or more interfaces.

## Extending a Component

- **Implementation can expose more than one interface**
  - Can use multiple inheritance
  - Must implement all the functions of each interface



26

As we saw on the previous slide, one way to extend the functionality of a component without breaking existing client applications is to implement an extended version of the original interface. Because the extended interface is polymorphic with the original interface, a client can treat an `ISavingsAccount` object (i.e an object that implements the `ISavingsAccount` interface) as either an `IAccount` object or an `ISavingsAccount` object.

However, extending an interface through inheritance only makes sense where the additional functions are semantically related to the functions in an existing interface. With our `ISavingsAccount` interface, this is obviously the case. But what if we want to provide a component with some additional capability that is either unrelated to an existing interface, or can stand alone in its own right? For example, say we wanted to implement some kind of `TradingAccount` which would allow a client to buy and sell some commodity using the funds of an account. At the very least, we'd need to provide clients with functions such as `Buy()`, `Sell()` and `GetPrice()`. The semantics of these methods are unrelated to the methods `Deposit()`, `Withdraw()` and `GetBalance()` of our existing `IAccount` interface. Also, we can envisage situations where we might want to implement these trading functions in isolation. Therefore, a better approach would be to define a separate `ITrade` interface with three methods `Buy()`, `Sell()` and `GetPrice()`. Then, as shown on the slide, we can use multiple inheritance to define a `CTradingComponent` class that implements both the `IAccount` and `ITrade` interfaces.

The slide also shows a feature of multiple inheritance from non-virtual base classes in that, although there is a single vtable, there is a separate vtbl pointer for each base class. Therefore, if (by some means) a client obtains pointers to both the `IAccount` and `ITrade` interfaces of a `CTradingAccount` object, these pointers will have *different* values.

## Discovering a Component's Interfaces



- Client could use RTTI (`dynamic_cast<>` operator)
- Problem is that RTTI is compiler dependent
  - Client and component developers must use same C++ compiler!
  - What about non-C++ clients?

```
// Client.cpp -----  
...  
// get an IAccount pointer  
IAccount *pIAccount = CreateAccount(100.00);  
  
if (pIAccount) {  
    // now try and get an ITrade pointer  
    ITrade *pITrade = dynamic_cast<ITrade*>(pIAccount);  
    if (pITrade) {  
        pITrade->GetPrice();  
        ...  
    }  
    pIAccount->Delete();  
}  
...
```

27

If a component implements more than one interface, and its `CreateXXX()` method returns a pointer to just one of these, how is a client to discover the component's other interface(s)? One possible solution with C++ is to use Run-Time Type Identification (RTTI) as shown in the example on the slide.

Having obtained a pointer, `pIAccount`, to the component's `IAccount` interface, we use the `dynamic_cast` operator in an attempt to cast `pIAccount` to a pointer to the component's `ITrade` interface. If `pIAccount` points to an object that is derived from the `ITrade` type (in other words, the component actually implements the `ITrade` interface), this `dynamic_cast` will succeed and `pITrade` will point to the component's `ITrade` interface. Otherwise, the cast will fail and `pITrade` will contain a null pointer.

Although this technique works with many compilers, RTTI is implemented in different ways by different compiler vendors. Even with Microsoft Visual C++ 6.0, RTTI support is not enabled by default (you must use the `/GR` switch). Besides, how are we to support clients that are implemented in non-polymorphic languages such as C?

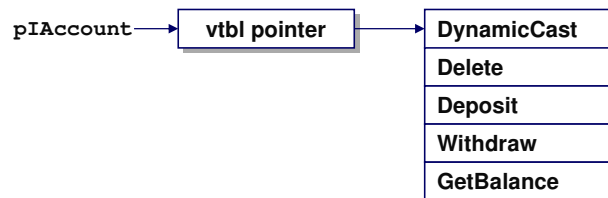
## Eliminating Compiler Dependencies

- **Provide a well-known DynamicCast() function**
  - Another virtual function in each interface
  - Same semantics as `dynamic_cast<>` operator
- **Could factor out common functions to a base interface**

```
class IBase {
public:
    virtual void *DynamicCast(const char *pszType) = 0;
    virtual void Delete() = 0;
};
```

*IBase interface*

- **All derived interfaces will have these functions as the first two entries in their vtables**



28

To eliminate C++ compiler dependencies and support non-C++ clients, the implementation class could provide a well-known function with the same semantics as the `dynamic_cast` operator. For this reason, we'll call this function `DynamicCast()`, although a better name might be `GetInterface()`. It will take a single parameter: the name of a requested "IAccount". If the requested interface is implemented, `DynamicCast()` should return a pointer to it; otherwise it should return null.

`DynamicCast()` must be a virtual function and together with `Delete()`, must be defined in every single interface. It therefore makes sense to define a base interface, which we'll call `IBase`, that defines just these two functions as shown on the slide.

All other interfaces must ultimately derive from `IBase`, so the first two entries of a their vtables will contain pointers to `DynamicCast()` and `Delete()` functions. This means that, having obtained a pointer to *any* interface of a component, a client can call `DynamicCast()` through that pointer to "discover" a pointer to another specified interface.

## Revised Implementation Code

```

class IAccount : public IBase {
public:
    virtual void Deposit(float amount) = 0;
    virtual void Withdraw(float amount) = 0;
};

class ITrade : public IBase {
public:
    virtual void Buy(long quantity) = 0;
    virtual void Sell(long quantity) = 0;
};

class CTradingAccount : public IAccount, public ITrade {
public:
    ...
    // IAccount functions
    void Deposit(float amount);
    ...
    // ITrade functions
    void Buy(long quantity);
    void Sell(long quantity);
    ...
    // IBase functions
    void *DynamicCast(const char *pszType);
    void Delete() {delete this; }
};

```

**IAccount interface**

**ITrade interface**

**CTradingAccount implementation:**  
Must implement all the functions of the IAccount, ITrade and the IBase interfaces

29

Now we're ready to revise the implementation code in order to define a CTradingAccount class that implements both the IAccount and ITrade interfaces. Note that we must also implement the methods of the IBase interface.

## Implementation of DynamicCast()

QA-IQ

- Allows client to request pointer to specified interface
- Use `static_cast<>` operator to simulate RTTI

```
// CTradingAccount.cpp
...
void *CTradingAccount::DynamicCast(const char *pszType) {
    if (strcmp(pszType, "IAccount") == 0)
        return static_cast<IAccount*>(this);
    else if (strcmp(pszType, "ITrade") == 0)
        return static_cast<ITrade*>(this);
    else if (strcmp(pszType, "IBase") == 0)
        return static_cast<IAccount*>(this);
    else
        return 0; // unsupported interface
}
```

Note: `static_cast<IBase*>(this)` would be ambiguous!

30

As we mentioned a couple of slides back, the `DynamicCast()` function allows a client to request a pointer to a specified interface. To implement this method, we simply compare the specified interface name with the names of the implemented interfaces, which in our example are "IAccount", "ITrade" and "IBase". If there's a match, we use the `static_cast` operator to cast the `this` pointer to the appropriate type and value; otherwise we return null.

Note that when a pointer to the implementation's `IBase` interface is requested, we must statically cast to either `IAccount` or `ITrade`. A static cast to `IBase` would be ambiguous because `IBase` is a common base class, which means that there will be multiple copies of `DynamicCast()` and `Delete()` in the implementation's vtable. You can deduce this by referring to the diagrams on slide 13. To prevent multiple copies of `IBase` occurring in derived interfaces, we could declare `IBase` as `virtual` when it is inherited, but this would change the layout of the vtable and introduce additional complications. For a more detailed explanation, see Section 10.3c, "Multiple Inheritance and Casting" in *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup.

## Instance Management



- **Replace CreateXXX() by a CreateObject() function that returns a pointer to component's IBase interface**

*CTradingAccount implementation*

```
IBase *CreateObject() {
    return reinterpret_cast<IBase *>(new CTradingAccount);
}
```

- **Client is responsible for managing component's lifetime**
  - Call CreateXXX() to get pointer to component's IBase
  - Call DynamicCast() to get pointer to a specified interface
  - Call Delete() (through any pointer) to destroy object
  - Must track which pointer refers to which interface and only call Delete() once per object!
- **Component should really manage its own life time**
  - Implement reference counting
  - Commit suicide when reference count falls to zero

31

It no longer makes sense to have a `CreateXXX()` method that returns a pointer to the component's XXX interface. Instead, it would be much better to provide clients with a well-known `CreateObject()` method that returns a pointer to the new component's `IBase` interface. As shown on the slide, we need to use a `reinterpret_cast`, because there is no other way to overcome a potential ambiguity (apart from casting first to a `void*`)

As things stand, the client is also responsible for deleting a component when it is no longer required. With the introduction of the `DynamicCast()` method, this is now much more complicated because the client can acquire several pointers to the different interfaces of the same component. It must therefore track which pointer refers to which interface and ensure that `Delete()` is called once (and only once) when none of the component's interfaces are required.

The component itself should really manage its own lifetime, and this could be achieved if the component maintained a "usage" or reference count. This count should be incremented when a client obtains another pointer to an interface and decremented when such a pointer is no longer required. A zero reference count indicates that none of the interfaces are in use, so the component can safely destroy itself. From the client's point of view, things are much simpler because it simply needs to inform the component when it is about to use or duplicate an interface pointer and when it has finished with that interface pointer. However, it is important to note that, in the absence of some form of smart pointer, the onus is still on the client to follow these rules.

## Reference Counting (1)



- In IBase, replace Delete() by IncrementRef() and DecrementRef() functions

```
class IBase {
public:
    virtual void *DynamicCast(const char *pszType) = 0;
    virtual void IncrementRef() = 0;
    virtual void DecrementRef() = 0;
};
```

IBase interface

- DynamicCast() should auto-increment count
- To avoid memory leaks, client must follow the rules:
  - Call IncrementRef() after duplicating an interface pointer
  - Call DecrementRef() when an interface pointer is no longer required

32

To implement reference counting, we need to modify our IBase interface and replace Delete() by two functions: IncrementRef() and DecrementRef(). As their names suggest, these functions allow a client to increment and decrement a component's reference count, respectively. It also makes sense for DynamicCast() to increment the reference count automatically, so we'll modify this function as follows:

```
void *CAccount::DynamicCast(const char *pszType) {
    void *pvResult = 0;
    if (strcmp(pszType, "IAccount") == 0)
        pvResult = static_cast<IAccount*>(this);
    ...
    ...
    reinterpret_cast<IBase *>(pvResult)->IncrementRef();
    return pvResult;
}
```

As mentioned on the previous slide, the client is responsible for calling IncrementRef() and DecrementRef() on each interface that it uses. Whenever it *duplicates* an interface pointer, it must call IncrementRef(). Similarly, whenever it has finished with an interface pointer, it must call DecrementRef(). Failure to follow these basic rules will result in memory faults, or worse still memory leaks!



## Reference Counting (2)

- Implementation is almost trivial
- **CreateObject()** should auto-increment count

*CAccount implementation*

```
// CTradingAccount.h
class CTradingAccount : public IAccount, public ITrade
{
    long m_cRef;
    ...
public:
    CAccount() : m_cRef(0) {}    // count = 0

    void IncrementRef() {
        ++m_cRef;
    }
    void DecrementRef() {
        if (--m_cRef == 0)
            delete this;        // self destruct
    }
    ...
};
```

33

As shown on the slide, the component-side implementation of reference counting is almost trivial. We simply need to add a member variable to hold the reference count, and override the `IncrementRef()` and `DecrementRef()` functions of the `IBase` base class.

`CreateObject()` should also increment the count automatically, so we'll modify this function as follows:

```
IBase *CreateObject() {
    IBase *pBase = static_cast<IBase *>(new CAccount);
    if (pBase)
        pBase->IncrementRef();
    return pBase;
}
```

## Revised Client Code

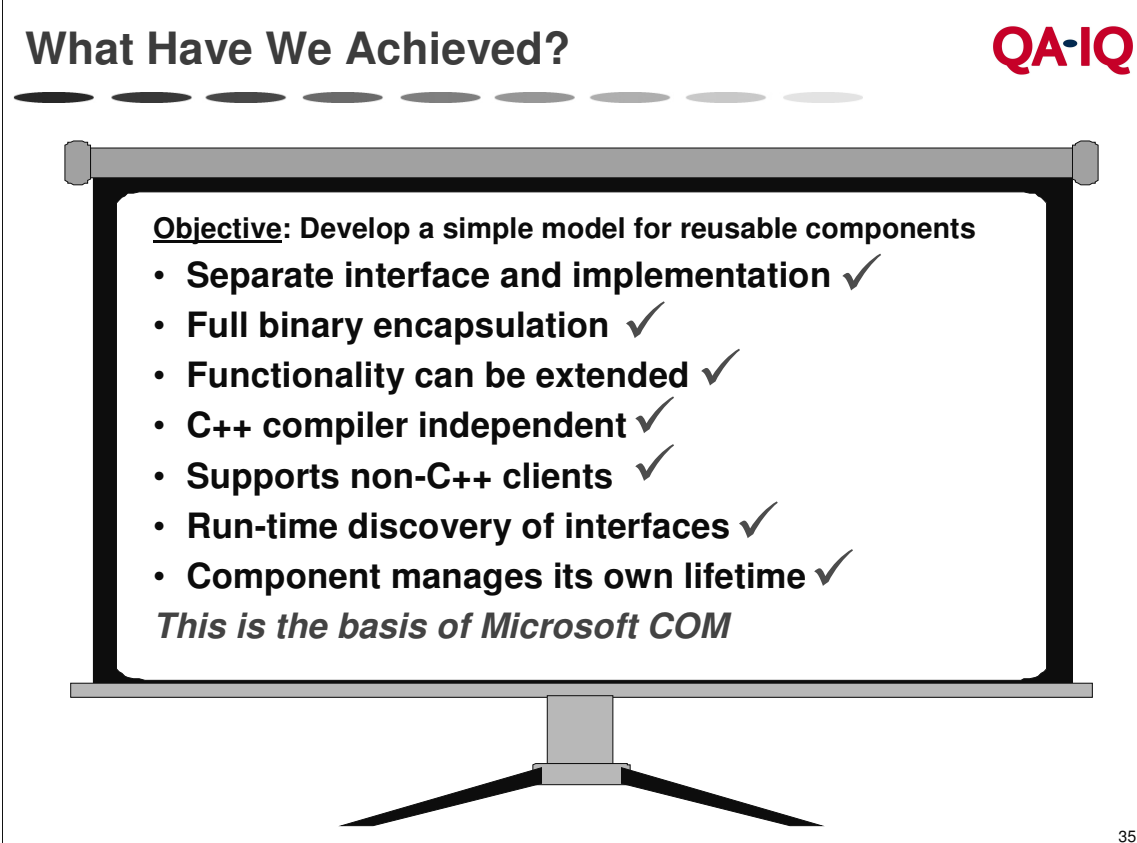


```
int main() {
    IBase *pIBase = CreateObject(); // ref count auto-incremented
    if (pIBase)
    {
        // try and get an IAccount pointer
        IAccount *pIAccount = (IAccount *)pIBase->DynamicCast("IAccount");
        if (pIAccount) { // ref count auto-incremented
            // perform some operations on the Account object
            pIAccount->Deposit(10000.00);
            // now try and get an ITrade pointer
            ITrade *pITrade = (ITrade *) pIBase->DynamicCast("ITrade");
            if (pITrade) { // ref count auto-incremented
                float price = pITrade->GetPrice();
                pITrade->DecrementRef(); // finished with ITrade
            }
            balance = pIAccount->GetBalance();
            pIAccount->DecrementRef(); // finished with IAccount
        }
        pIBase->DecrementRef(); // finished with IBase
    }
    ...
}
```

34

Finally, here is the revised client code. Notice that we call `DecrementRef()` every time we've finished with an interface pointer.

## What Have We Achieved?



**QA-IQ**

**Objective:** Develop a simple model for reusable components

- **Separate interface and implementation** ✓
- **Full binary encapsulation** ✓
- **Functionality can be extended** ✓
- **C++ compiler independent** ✓
- **Supports non-C++ clients** ✓
- **Run-time discovery of interfaces** ✓
- **Component manages its own lifetime** ✓

***This is the basis of Microsoft COM***

35

Our objective was to develop a simple model for implementing reusable components in C++. We've separated the component's interface from its implementation by using an abstract base class to represent the interface. As a result, our component's interface is based on a vtable - a binary standard, which means that non-C++ clients can use the interface.

Once defined, an interface is immutable, but we can easily derive a new interface from an existing interface. We can also extend the functionality of a component by using multiple inheritance, that is, by deriving the implementation class from two or more interface classes.

Using a technique akin to RTTI, we've defined a method in a base interface class that allows a client to request a pointer to a specified interface. If the component implements the requested interface, it returns a pointer to it; otherwise it returns null.

Finally, we implemented reference counting so that a component can manage its own lifetime. However, the client is still responsible for calling appropriate functions in the base interface to increment and decrement this count.

In fact, we've developed the basis of Microsoft's Component Object Model (COM).

## Summary



- **Difficult to rewrite reusable class libraries in C++**
  - Lack of binary encapsulation, binary standard, name mangling, etc.
- **Need to separate a component's interface from its implementation**
  - Can be achieved by defining interface in a "pure" abstract base class
  - Creation and deletion need to be handled by wrapper functions
- **Once defined, an interface is immutable, but**
  - One interface can inherit from one other interface
  - A component can implement multiple interfaces
- **Component interface(s) can be discovered at run time**
  - Base interface must provide a `DynamicCast()` function
- **A component can manage its own lifetime**
  - Base interface must also provide `IncrementRef()` and `DecrementRef()` functions
  - Client must follow the rules to avoid memory leaks

36

---

For a more detailed examination of the topics discussed in this chapter, read the first chapter (*COM as a Better C++*) of *Essential COM* by Don Box.