Visual Studio

Module 4

Advanced Object-Oriented Programming

1 Extending Classes

1.1 Extending a Class

Rather than modifying an existing class you can extend a new class from it. The new class will inherit all the members from the existing class. This will also make the new class fully compatible with the class it extended from. In C++ we can call the existing class the base class and the new one the derived class. You can add new members to the derived class or replace inherited members.

Extending a class: Extend1\main.cpp

```
#include <stdio.h>
class Class1 {
public:
    void f1() { puts("Class1::f1()"); }
    void f2() { puts("Class1::f2()"); }
};

class Class2 : public Class1 {  // extend from Class1
public:
    void f3() { puts("Class2::f3()"); } // new member
    void f2() { puts("Class2::f2()"); } // 'shadow' inherited member
};

class Class3 : public Class2 {  // extend from Class2
public:
    void f4() { puts("Class3::f4()"); } // new member
    void f3() { puts("Class3::f3()"); } // 'shadow' inherited member
};

void main() {
    Class1 o1;
    Class2 o2;
    Class3 o3;
    o1.f1();    // calling original function in Class1
    o1.f2();    // calling original function in Class1
    o2.f1();    // calling inherited function from Class1
    o2.f2();    // calling shadow function in Class2
    o2.f3();    // calling new function in Class3
    o3.f1();    // calling inherited function from Class1
    o3.f2();    // calling inherited function from Class2
    o3.f3();    // calling inherited function from Class2
    o3.f4();    // calling new function in Class4
}
```

When you replace an inherited member it does not mean that the inherited member is no longer there. It only removes it from direct access. You can still access back any of the members you replaced by scoping the members with the base class name. So you can extend an existing function rather than totally replacing it.

Extending a function

```
class Class2 : public Class1 {
        :
    void f2() {
        Class1::f2();           // run inherited code
        puts("Class2::f2()");   // run extended code
    }
}
```

1.2 Virtual Functions

We call the previous method of replacement shadowing. However shadowing does not work through polymorphism. This means that regardless of what object you create a compiler would only generate code to call the inherit members if it thinks the object is from the base class. Note that polymorphism works only through either references or pointers and not by value.

Function to call functions in a Class1 object

```
void UseObject1(Class1& obj) {
    obj.f1();   // always Class1.f1()
    obj.f2();   // always Class1.f2()
}

void main() {
        :
    UseObject1(o1); // passing in a Class1 object
    UseObject1(o2); // passing in a compatible Class2 object
    UseObject1(o3); // passing in a compatible Class3 object
}
```

Regardless of what kind of object you pass to the **UseObject1** function it would only call back **Class1** functions even if you have shadowed them. In order for it to call the correct functions based on object and not type, you have to use overriding instead of shadowing. However you can only override a virtual function in the base class.

Marking functions as virtual

```
class Class1 {
public:
    virtual void f1() { puts("Class1::f1()"); } // allow override
    virtual void f2() { puts("Class1::f2()"); } // allow override
};
```

If you execute the program again you can see that the correct functions are called. When you mark a function virtual the compiler will generate a virtual table (VTBL) for the class. This table will be used to contain the address of each virtual function in the class. Even though virtual table is still inherited by a derived class, if a virtual function has been overridden, the derived class will have its own virtual table that will contain the address of the overriding function instead.

When you create an object, the address of the correct virtual table is assigned to the object. That is why the size of each object is increased by 4 bytes since it has to store a pointer to the virtual table. When a virtual function is to be called, the compiler will generate code to retrieve the address of the function from the virtual table of the object and call the function using the address. This guarantees that the overriding functions are always called. Note that destructors should always be marked virtual to ensure that the right destructor is called.

1.3 Extending Constructors

Constructors are automatically extended. This guarantees that objects are initialized properly. When you call a derived constructor, it will call the base constructor. This can be proven from the following code. The default base constructor is called regardless of which constructor is called in the base class.

Extending constructors: Extend2\main.cpp

```
class Class1 {
    int value1;
public:
    Class1() { value1 = 1; }
    Class1(int value) { value1 = value; }
    void Show() { printf("value1=%ld\n",value1); }
};

class Class2 : public Class1 {
    int value2;
public:
    Class2() { value2 = 2; }
    Class2(int value) { value2 = value; }
    void Show() {
        Class1::Show();
        printf("value2=%ld\n",value2);
    }
};

void main() {
    Class2 obj1;
    Class2 obj2(10);
    obj1.Show();
    obj2.Show();
}
```

As you will see from the results of the example, the default constructor for the parent class is automatically called each time we call any of the constructors in the extended class. But what if we wish to call another constructor of the parent class instead? You can do this by specifying exact constructor to call plus any arguments to pass over from the current constructor in the format shown in the example below. Note that you can put in a list of constructors to call if extending from multiple base classes rather than one.

Calling explicit parent constructors

```
constructor(...) :
    parentconstructor(...),
    parentconstructor(...),
    ... {
}
```

Note that even though the calling of the parent constructor is placed outside the code block, it is also a normal C++ statement. You can code it in the some manner as you would if the statement were inside the code block.

Calling custom constructor in base class

```
class Class2 : public Class1 {
    Class2(int value) : Class1(value / 2) {
        value2 = value;
    }
        :
```

1.4 Inheriting Access Rights

In the above example the derived class cannot access member variables that belong to the base class since they are marked as private. Inheritance does not equal access. To let base classes to have direct access to member variables, use **protected** instead of **private**.

Marking and accessing protected members

```
class Class1 {
protected:
    int value1;
        :
};

class Class2 : public Class1 {
        :
    void Show() {
        printf("value1=%ld\n",value1);
        printf("value2=%ld\n",value2);
    }
};
```

Though access rights are inherited from the base class, a derived class can choose to change access rights of inherited members so that the next generation of classes will not be able to access inherited members even if marked as public or protected. When inheriting members from a base class, you can inherit all the existing access rights by using **public** access modifier. Any public and protected members will remain the same in the class while any private members are still not accessible. If you use **protected** access modifier, all the public members becomes protected in the derived class and using **private** modifier will cause inherited public and protected members to become private in this new class as shown below.

Inheriting existing rights

```
class Class2 : public Class1 {
        :
```

Public members become protected members

```
class Class2 : protected Class1 {
        :
```

Public and protected members become private members

```
class Class2 : private Class1 {
        :
```

1.5 Typecasting Operators

As mentioned above, derived classes are considered compatible since they inherited all the features from the base classes. Compatibility can be tested thru polymorphism as shown below.

Class compatibility by inheritance (CastOp1\main.cpp)

```
class A { public: virtual int foo1(int a) { return a * 2; } };
class B : public A { public: virtual int foo2(int a) { return a * a; }} ;
class C : public B { public: virtual int foo2(int a) { return a / 2; }} ;
class D { public: virtual double foo1(double a,b) { return a * b; } };

void main() {
    A * p1 = new B; // B is compatible with A
    A * p2 = new C; // C is compatible with A
    B * p3 = new C; // C is compatible with B
    B * p4 = new A; // questionable!
}
```

Upcasting is always considered okay, since we know that whatever the features are in the base class is also available in derived classes as well. Upcasting is automatic since compatibility is guaranteed. Downcasting is questionable but as long as you only use inherited features it should be okay but you need explicit casting.

Old-style C typecasting

```
    B * p4 = (B *) new A;    // questionable but no errors.
    B * p5 = (B *) new D;    // not compatible but no errors.
```

The problem with type-casting is that even incompatible types can be casted. There is no checking for compatibility whatsoever when you use C-style type-casting. In C++, there are new casting operators. These operators are intended to remove some of the ambiguity and danger that is inherent for old style C language casts since this type of typecasting. The format of the typecasting operators in C++ is shown as follows.

Format of C++ type-casting operator

```
            castoperator<new type>(expression)
```

Replacing Old-style C typecasting

```
    B * p4 = reinterpret_cast<B *>(new A);
```

The above cast is the same as the old style C typecast, except it follows the syntax of C++ typecasting rather than C. No checking is done to ensure partial compatibility. If you need to check compatibility during compilation, use **static_cast** instead.

Related classes typecasting

```
B * p4 = static_cast<B *>(new A);    // B and A related!
B * p5 = static_cast<B *>(new D);    // no relationship!
```

However **static_cast** only checks at compilation. To check compatibility at runtime, use **dynamic_cast** instead. A **NULL** is assigned to the pointer if the object it is not compatible to what you say it is.

Using dynamic casting

```
B * p4 = dynamic_cast<B *>(new C);   // C is fully compatible to B
B * p5 = dynamic_cast<B *>(new D);   // D is totally incompatible to B
if(!p4) puts("p4 is NULL!");
if(!p5) puts("p5 is NULL!");
```

1.6 Inheritance & Aggregation

There are two ways to re-use existing types; inheritance and aggregation. When you extend a new class from a base class, you gain all the features of the base class. You can then use those features in your new class through inheritance. Another choice is for the new class to contain an object from an existing class. When the outer object is created, the inner object will also be created. The outer constructor will be extending the inner object constructors as well. When the outer object is destroyed, the inner object will also be destroyed as well. The outer destructor will be extending the inner object destructor. This is called aggregation. All you need to do is to write code in the outer object to use or expose the inner object.

Using inheritance: Aggregate1\main.cpp

```
#include <iostream>
#include <vector>
using namespace std;

class List1 : vector<double> {  // List1 is a vector<double>
public:
    List1() {   // List1 : vector<double>()
        push_back(1.1);
        push_back(2.2);
        push_back(3.3);
    }
    double Sum() {
        double total = 0;
        for (auto value : *this)    // requires C++/11
            total += value;
        return total;
    }
};

class List2 {   // List2 aggregates a vector<double>
private:
    vector<double> m_v;
public:
    List2() {   // List2 : m_v()
        m_v.push_back(1.1);
        m_v.push_back(2.2);
        m_v.push_back(3.3);
    }
    double Sum() {
        double total = 0;
        for (auto value : m_v)  // requires C++/11
            total += value;
        return total;
    }
    void push_back(double v) {
        m_v.push_back(v);
    }
};

void main() {
    List1 l1;
    List2 l2;
    l1.push_back(4.4);
    l2.push_back(4.4);
    cout << l1.Sum() << endl;
    cout << l2.Sum() << endl;
}
```

1.7 Polymorphism

As long as you use virtual functions, your objects can be polymorphic. It means that you can replace one type of object with a different type of object at runtime. Think of it as ability to implement replaceable alternative components. Sometimes we create a base class not to extend but to act as a "standard interface" to access different kinds of objects. Polymorphic types must have a virtual destructor which guarantees that all objects are destroyed properly.

Declaring the interface: Poly1\Log.h

```
#pragma once

class CLog {
public:
    virtual void Write(char *message);
    virtual ~CLog();    // destructor must be virtual
};

CLog *GetLog(); // function that returns a CLog compatible object
```

Implementing the interface: Log.cpp

```
#include "Log.h"

// void CLog::Write(char * message) { }
CLog::~CLog() { }
```

As you can see above that we have provided a blank implementation which technically means that a **CLog** object does not really do anything. This is because we only use it to inherit the virtual table. We can now implement derived classes that will actually do real work. Not every class requires a destructor. If no destructor is provided then the base destructor will be called instead. That is why we provide one in the base class so we can inherit it. In C++/11 you can use **override** to highlight that it is an overriding function.

Derived class that logs to console window

```
#include <stdio.h>
class CConsoleLog : public CLog {
public: void Write(char * message) override { printf(message); }
};
```

Derived class that logs to a MessageBox

```
#include <windows.h>
class CWindowLog : public CLog {
    void Write(char * message) override {
        MessageBoxA(NULL, message, "Log", MB_OK);
    }
};
```

Derived class that logs to a text file

```
class CFileLog : public CLog {
private:
    FILE *m_file;
public:
    CFileLog() { fopen_s(&m_file, "Log.txt", "w"); }
    ~CFileLog() { fclose(m_file); }
    void Write(char * message) override { fputs(message, m_file); }
};
```

We want to be able to write programs where components can replaced without having to change code and recompile the program. All you need to do store some information externally; in a file, in an environment variable, or in the Windows registry identifying which component to use. You are then be able to create and return the correct object. For our example we will use an environment variable named **Log**.

Using environment variable to instantiate correct object

```
CLog *GetLog() {
    char logName[256];
    GetEnvironmentVariableA("Log", logName, sizeof(logName));
    if (strcmp(logName, "console") == 0) return new CConsoleLog();
    if (strcmp(logName, "window") == 0) return new CWindowLog();
    if (strcmp(logName, "file") == 0) return new CFileLog();
    return nullptr;
}
```

We can now write and compile a program that can use objects from any of the above classes without having to recompile the program.

Using CLog to access different log objects: Poly1\main.cpp

```
#include "Log.h"

void main() {
    CLog *pLog = GetLog();
    if (pLog != nullptr) {
        pLog->Write("Hello!\n");
        pLog->Write("Goodbye!\n");
        delete pLog;
    }
}
```

Once you have compiled the program you can open a console window to test out the program. Set the environment variable to **console**, **window** or **file** before running it. You can then see that program can work differently because it uses a different object but there is no need to have to change code and recompile the program or having to write multiple programs.

Testing polymorphism

```
C:\CPPDEV\SRC\Module4\Debug>set Log=console
C:\CPPDEV\SRC\Module4\Debug>Poly1
C:\CPPDEV\SRC\Module4\Debug>set Log=window
C:\CPPDEV\SRC\Module4\Debug>Poly1
C:\CPPDEV\SRC\Module4\Debug>set Log=file
C:\CPPDEV\SRC\Module4\Debug>Poly1
C:\CPPDEV\SRC\Module4\Debug>type Log.txt
```

Note that **CLog::Write** function will never be called. This is because CLog is only used for polymorphism but we will never create a CLog object. In this case you can remove the implementation and made it a pure virtual function. A pure virtual function can be called also as an abstract function.

Pure virtual function: Poly1\Log.h

```
class CLog {
public:
    virtual void Write(char *message) = 0; // pure virtual function
    virtual ~CLog();    // destructor must be virtual
};
```

A pure virtual function means that there is no implementation so nullptr will be stored inside the virtual table instead. The purpose is to generate a virtual table so that the function can be overridden by derived classes not that you want to implement it in the base class. However this also means the **CLog** is not complete and thus the compiler will not allow you to create an object from this class. We can call this kind of class as an abstract class.

Attempting to create an object from an abstract class

```
CLog *pLog = new CLog();    // this statement cannot be compiled
```

1.8 Interface-Based Programming

Even though C++ has no built-in support for interfaces, an interface is a pure abstract virtual table. This provides a very high-level of abstraction for polymorphism. While a normal class can contain code and data that you can inherit or choose to override, the interface is completely abstract and only meant for polymorhism, not inheritance. You can use **struct** instead of **class** since there is pointless to have a private interface.

An example interface: Log.h

```
struct ILog {   // replaces CLog
    virtual void Write(char *message) = 0;
    virtual void Release() = 0;
}
ILog *GetLog();
```

You can only use objects through an interface but not create or destroy objects. So it will not contain a declaration of the destructor. Instead provide a **Release** function to be called when you no longer wish to use the object so it can self-destruct.

Classes implement the interface v-table

```
class CConsoleLog : public ILog {
        :
    void Release() override { delete this; }
};

class CWindowLog : public ILog {
        :
    void Release() override { delete this; }
};

class CFileLog : public ILog {
        :
    void Release() override { delete this; }
};

ILog *GetLog() {
        :
}
```

Using objects thru interfaces

```
#include "Log.h"

void main() {
    ILog *pLog = GetLog();
    if (pLog != nullptr) {
        pLog->Write("Hello!\n");
        pLog->Write("Goodbye!\n");
        pLog->Release();
    }
}
```

2 COM Components

2.1 COM Server

Microsoft has provided a language-independent technology called Component Object Model (COM) that allows you to implement and use components in any programming language, even from scripting languages. However, constructing COM components can be quite complex and time consuming because of the amount of code that is required to work. There are also so many steps required to expose your component for use by other languages. If you miss one steps or you did not implement the step correctly, your component will fail to work through COM. To allow COM components to be built more easily, you can make use of the Active Template Library (ATL). ATL makes use of a set of pre-build templates that can be used to generate the code to implement the basic structure of components. Thus you do not need to write COM code yourself.

In this chapter, we will use ATL to construct a simple component. Implementing COM components without ATL is outside the scope of this class. You are also advised to learn COM in more detail if you wish to construct COM components correctly, with or without ATL. To build COM components using ATL, you must create an **ATL** project. When an ATL project is compiled, it generates an EXE or DLL that is technically called as the COM server. You can now add a new ATL Project named LogCom in the solution and accept the default settings.

Project Information

```
Project Name: LogCom
Project Type: Visual C++ | ATL | ATL Project
Location    : C:\VCDEV\SRC
Solution    : Module3
```

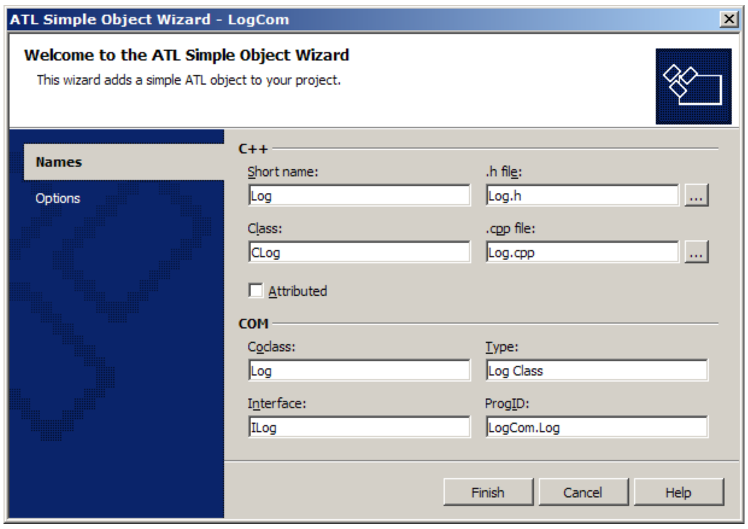
You will notice that it actually creates two projects; LogCom and LogComPS. The extra project is to build a proxy and stub that allows you to access a COM server across the network using a technology called Distributed-COM (DCOM). However it is no longer necessary to do this since the operating systems since Windows XP comes with COM+ that can host remote components and generate the proxy on demand. You can safely remove and delete the additional project from the solution.

You can build the COM server immediately but then it would be useless as there is no component implemented in the server at the moment. A COM server can have one or more COM classes where each class can support one or more COM interfaces. Every interface can be separate or can extend from an existing interface. To be useable by both compiled and scripting languages, minimally a dual-interface is required.

2.2 COM Class

Implementing a COM class completely from scratch is complex and time consuming. You have to write code to do reference counting, declare and implement interfaces. To simplify the process, you can make use of ATL templates to help you generate a COM class. To create a COM class using ATL add an ATL Object into your project. You can create specialized COM classes that can integrate with Internet Explorer or Microsoft Office. To implement a custom COM class, use ATL Simple Object. Right-click over the project and select the *Add | Class...* then choose the **ATL Simple Object** template. A dialog will appear for you to enter the class details. Just enter the short name for your COM component and Visual Studio will generate the class for you. You can customize the class name and also the file names if you wish.

Providing a name for your COM class



Welcome to the ATL Simple Object Wizard

This wizard adds a simple ATL object to your project.

C++

Short name: Log

.h file: Log.h

Class: CLog

.cpp file: Log.cpp

Attributed [ ]

COM

Coclass: Log

Type: Log Class

Interface: ILog

ProgID: LogCom.Log

Scripting languages commonly requires a programmatic ID (ProgID) to instantiate the COM object. Try to use a more unique name to avoid conflict with other components. It is common to combine the name of the COM server with the COM class name for a ProgID. Use **LogCom.Log** as the ID for our COM class. Accept the default options and click *Finish* to generate the basic code to instantiate a COM class from ATL templates as shown below.

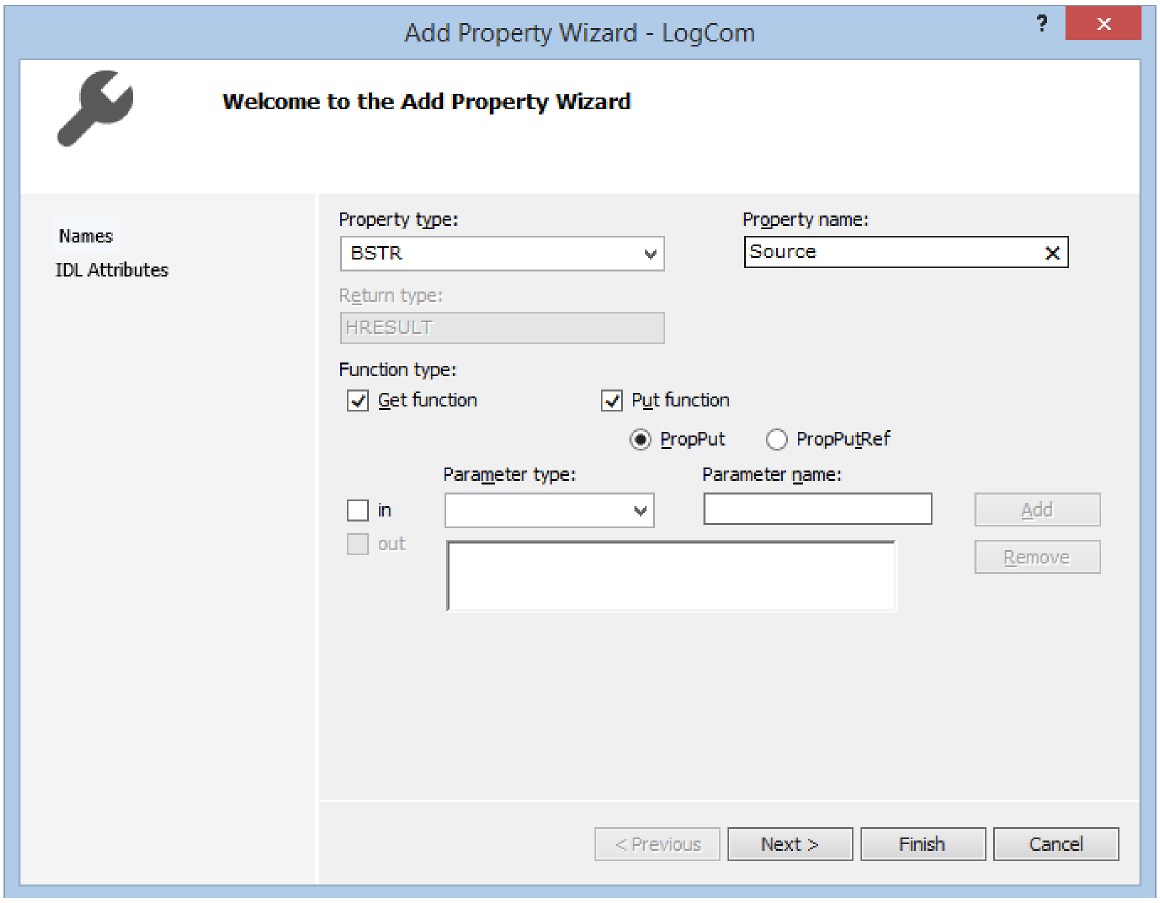
COM class extended from ATL templates: Log.h

```
class ATL_NO_VTABLE CLog :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CLog, &CLSID_Log>,
    public IDispatchImpl<ILog, &IID_ILog, &LIBID_LogComLib, /*wMajor =*/ 1,
/*wMinor =*/ 0>
{
        :
}
```

2.3 COM Interface

A COM interface will be automatically be generated for the class. Switch to *Class View* to see the empty **ILog** interface created for the **CLog** class that has a lollipop icon. It is a common icon used in diagrams to indicate an interface. You can declare methods (custom functions) and properties (accessor functions) in a COM interface. Let us add a **Source** property to the interface. Right-click over the interface and select the *Add* | *Add Property...* option.

Add Property Wizard dialog


Add Property Wizard - LogCom

Welcome to the Add Property Wizard

Names
IDL Attributes

Property type: BSTR

Property name: Source

Return type: HRESULT

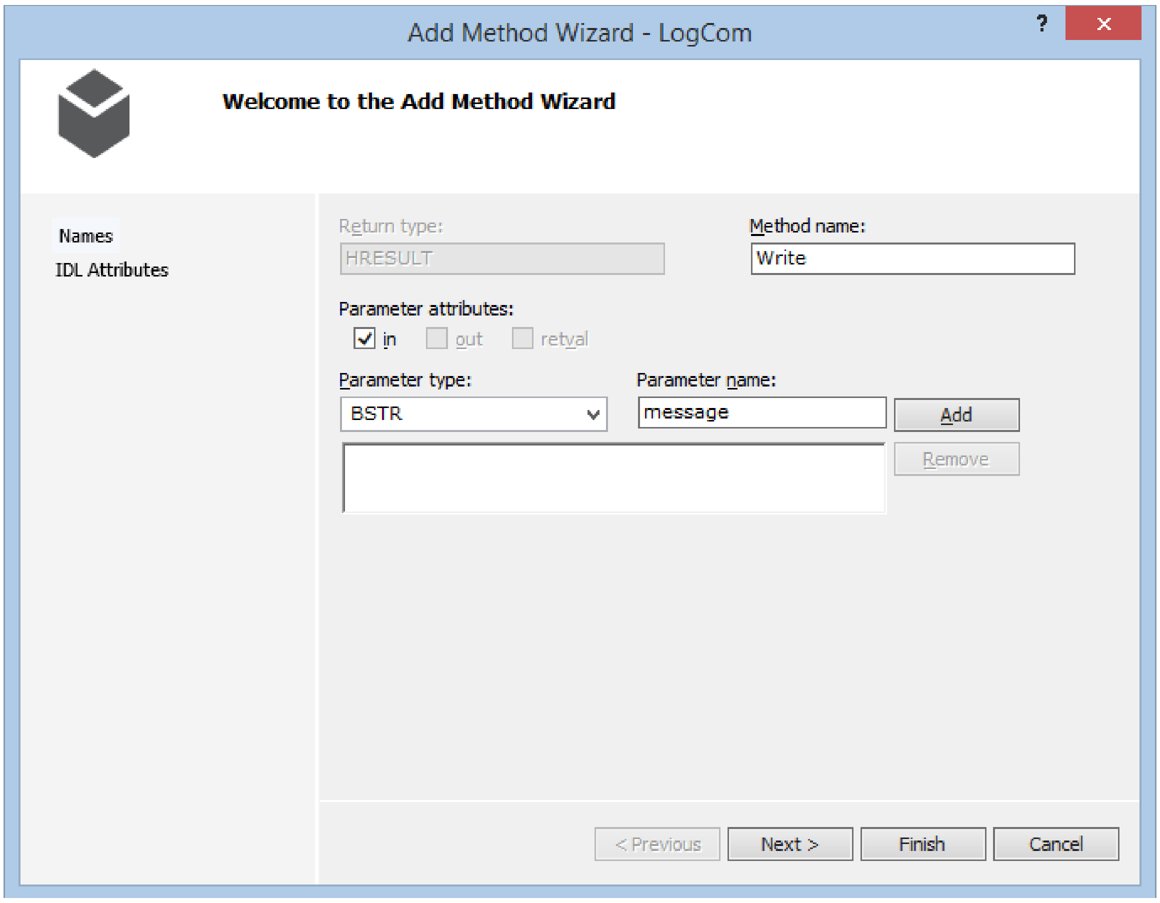
Function type:
[✓] Get function
[✓] Put function
[● PropPut] ○ PropPutRef

Parameter type:
Parameter name:
☐ in
☐ out

Not all languages support the same data types. To support multiple languages use the types specially implemented for COM rather than specific to a certain language. String is definitely different between languages like C/C++ and Visual Basic. Strings in C is a null-terminated array of characters while in VB, strings is a structure that contains a length followed by a list of characters that is not null-terminated. To support scripting languages and VB, use the **BSTR** type in COM instead. You can now use the dialog to create a **BSTR** property named **Source** and make sure it has both a **Get** and a **Put** function. Double-check before confirmation as there is no wizard to help you modify a property. You have to edit both the class and the interface files which is complicated if you are not familiar with COM.

You can also add methods to an interface. Each method will become a single function in the class. Methods can have multiple parameters and a return value. Return value is implemented in the function as an output parameter since all COM functions must always return a COM success or failure code number. You can now add a new method named **Write** that accepts one **BSTR** input parameter named **message**. Make sure to verify that the parameters are correctly added before confirmation.

Add Method Wizard dialog


Add Method Wizard - LogCom

Welcome to the Add Method Wizard

Return type: HRESULT

Method name: Write

Parameter attributes: [✓ in] out retval

Parameter type: BSTR

Parameter name: message

COM interfaces are described using a special language named IDL. You need to know IDL to modify interfaces and declare language-independent enumerations, basic data-structures and other custom types that are not COM classes. An MIDL compiler will be used automatically to generate C++ code for all interfaces and types from the IDL file when you build your project. Clicking on the property or method in the interface takes you to the IDL file. To implement the functions in the class, select the **CLog** class to see the **get_Source** and **put_Source** function as well as the **Write** function. Clicking on the functions will take you to the C++ source file.

2.4 BSTR Type

You can now start implementing the COM class. Open the header file first to declare a **BSTR** member variable named **m_source** to maintain the source string. Then declare a constructor and a destructor for the class since a BSTR is a dynamic resource, it has to be released when the COM object is destroyed. Note that an inline constructor has already been added but for this example, we prefer to implement all the code in the C++ source file instead.

Adding additional class members: Log.h

```
class ATL_NO_VTABLE CLog :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CLog, &CLSID_Log>,
    public IDispatchImpl<ILog, &IID_ILog, &LIBID_LogComLib, /*wMajor =*/ 1,
/*wMinor =*/ 0>
{
private:
    BSTR m_source;
public:
    CLog();
    ~CLog();
        :
};
```

Use **SysAllocString** to allocate a BSTR and **SysFreeString** to release. Note that the BSTR is always Unicode. However conversion between BSTR and other string types is very easy because ATL already provides a set of conversion macros. For example you can convert between TCHAR string to BSTR using **T2BSTR** and **OLE2T** macros.

Allocating and releasing BSTR: Log.cpp

```
CLog::CLog() { m_source = SysAllocString(L"C:\\TEMP\\Log.txt"); }

CLog::~CLog() {
    if (m_source != NULL)
        SysFreeString(m_source);
}

STDMETHODIMP CLog::get_Source(BSTR* pVal) {
    *pVal = SysAllocString(m_source);   // output parameter
    return S_OK;
}

STDMETHODIMP CLog::put_Source(BSTR newVal) {
    if (m_source != NULL) SysFreeString(m_source);
    m_source = SysAllocString(newVal);  // input parameter
    return S_OK;
}
```

Using conversion macros in Write

```
STDMETHODIMP CLog::Write(BSTR message) {
    SYSTEMTIME time;
    TCHAR text[1024];
    DWORD dwBytes;
    GetSystemTime(&time);
    HANDLE hFile = CreateFile(OLE2T(m_source),
        GENERIC_WRITE, NULL, NULL, OPEN_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);
    _stprintf_s(text, _T("[%04d-%02d-%02d %02d:%02d:%02d] %s\r\n"),
        time.wYear, time.wMonth, time.wDay,
        time.wHour, time.wMinute, time.wSecond,
        OLE2T(message));
    SetFilePointer(hFile, 0, 0, FILE_END);
    dwBytes = _tcslen(text) * sizeof(TCHAR);
    WriteFile(hFile, text, dwBytes, &dwBytes, NULL);
    CloseHandle(hFile);
    return S_OK;
}
```

2.5 COM Registration

All COM servers must be registered first before usage. Visual Studio will automatically do this for you when you build your project. You would have to do this manually if you deploy the COM server on a separate machine. Registration will record the location of the COM server in the Windows registry together with all the IDs and other settings in order to instantiate and access the COM object. If you move your COM server you will need to register again. Before removing the COM server, you should unregister it first to clean up the Windows registry.

Registering a COM server

```
regsvr32 LogCom.dll
```

Unregister a COM server

```
regsvr32 /u LogCom.dll
```

COM servers are actually self-registering but since a DLL cannot run by themselves, it has to be loaded by the *regsvr32* program. Once the DLL is loaded, the program wiill then call **DllRegisterServer** or **DllUnregisterServer** functions. These functions are automatically generated when you build your COM server. Once registration completes you can straightaway use the COM components from any language. The following are example VBScript and JScript scripts to test the **Log** component. Note that the ProgID is used to instantiate the COM object.

VBScript script to test the Log component: UseLogCom1.vbs

```
set logger = CreateObject("LogCom.Log")
logger.Source = "C:\CPPDEV\UseLogCom1.log"
logger.Write "Hello!"
logger.Write "Goodbye!"
WScript.Echo "Done!"
set logger = Nothing
```

JScript script to test the Log component: UseLogCom2.js

```
var logger = new ActiveXObject("LogCom.Log")
logger.Source = "C:\\CPPDEV\\UseLogCom2.log"
logger.Write("Hello!")
logger.Write("Goodbye!")
WScript.Echo("Done!")
logger = null
```

2.6 Using COM in C++

Scripting languages uses a special interface named **IDispatch** to access COM objects. However this interface is cumbersome and slow to be used from compiled languages. You should use the specific **ILog** custom interface from languages that can support custom interfaces like *Visual Basic, C#* and *C++*. To use a custom interface you would need to have the interface declaration. Normally in C++, we will declare classes and interfaces in a header file. However C++ header files are useless in other languages. When you build a COM component, the MIDL compiler will generate a type-library file (TLB) for the COM server. In Visual C++, we use a #import statement to import a type library either from the COM server directly or from the separate TLB file. You can choose to import with or without namespace. To test out our own COM server, create a normal Win32 project and import the type library of our component in our source file as shown below.

Importing a type library: TestLogCom\main.cpp

```
#import "F:\VCDEV\SRC\Module3\Debug\LogCom.dll" no_namespace
```

An application needs to initialize the COM library before we create and use any COM objects. Call **CoInitialize** function to initialize. Before your application terminates call the **CoUninitialize** function to release resources allocated during initialization. COM objects are dynamic. Thus you can declare variables just before initializing the library. However, do not create any object before initialization or your application will crash. You should make sure that all objects are released before un-initialization. Deleting objects after this will cause your application to crash.

Initializing and uninitializing COM

```
#include <windows.h>
#include <tchar.h>

void main() {
    CoInitialize(NULL);
        :   <-- Create and use COM objects
    CoUninitialize();
}
```

Importing type libraries also generates smart pointers to simplify the construction and management of COM objects. The smart pointers also automatically also uses special classes like **_bstr_t** and **_variant_t** to perform auto-conversion between C++ types and COM types. The smart pointer class name is the same as the interface except for a **Ptr** suffix.

Declaring a smart pointer object for the ILog interface

```
ILogPtr ptr;
```

You can call **CreateInstance** method to create the actual COM object using either the ProgID, a text description of the COM class recorded in the Windows registry, or the CLSID, a unique ID assigned to each COM class. You can find a CLSID for your component classes inside the IDL file generated by Visual Studio. Since an IDL file is used to generate the type library, you can obtain the CLSID from the imported type library using a **__uuidof** system function. Using CLSID from the type library during compile time will definitely be faster than searching the registry for the ProgID during runtime.

Creating COM object using ProgID & CLSID

```
ILogPtr1; ptr1.CreateInstance("LogCom.Log");
ILogPtr2; ptr2.CreateInstance(__uuidof(Log));
```

If you wish to create the COM object together with the smart pointer, you can use the constructor of the smart pointer class to create the COM object. Once the COM object has been created, you can access the interface of the COM object through the smart pointer object the same method you access dynamic objects in C++. Call **Release** on the smart pointer or wait for it to go out of scope to destroy the COM object.

Creating COM object through smart pointer constructor

```
ILogPtr ptr(__uuidof(Log));
ptr->Source = _T("C:\\CPPDEV\\UseLogCom3.log");
ptr->Write(_T("Hello!"));
ptr->Write(_T("Goodbye!"));
ptr.Release();
```

2.7 Using COM+

To run COM+ on another system open Component Services from Administrative Tools in the Control Panel. You can then register the remote computer or access the current computer to install the COM component. In **COM+ Application**, add an application to host your COM component named as **Logger Application**. Set **Activation Type** as **Server** in order to be accessible both locally and remotely. Assign an account that has enough permissions to run the component. For example the **Log** component need to have enough permission to write to a file. You can select **Local Service** account to use a local administrative account.

Once the application has been created. Right-click on **Components** and select *New | Component...* to install the component. Even if you have previously registered you will need to re-register it again as a new component for COM+. COM+ will inform you at this point if there is a reason why the component cannot run under COM+. Once the component has been installed, you should see an icon appearing for it. When the COM component is being used, the icon should animate. You can try this out by running the test scripts or program that you have implemented in the previous sections. Once the COM object is destroyed, the icon should stop animating.

To access the component from another computer, you must first make sure that any COM+ application using the component on that computer has been deleted. You also need to unregister the COM component if it was registered outside of COM+. You can then generate a proxy to install on that computer. Right-click on **Logger Application** and select the **Export...** option. Select **Application Proxy** and enter the location and name for the proxy installer file to generate. Run the installer on any computers that you want to access the component from.

You can secure your COM+ components by creating roles. In each role you can assign one or more Windows groups and accounts. Right-click on each component in COM+ and then select **Security** tab. Enable security access check and then select the roles that are allowed to access the component. The client gets an access denied exception when attempting to use the component with if their current account is not part of the selected roles. COM+ provides many other features to a COM component and not just to run it remotely.