`

CPP301-C

Module 2

Implementing Component-Based Architecture

1 Implementing & Using Libraries

1.1 Static Library

To convert our application to a static library is quite simple. We still wish to maintain the existing application, so make a copy of the **symbion** project folder to **symbion1** and then rename the project file in the folder to the same name. Remove build folders in the directory contain temporary files build from the previous project. You can start Visual Studio and add the **symbion1** project to the folder. Right-click over the project and select *Properties* option. Make you set *Configuration* to *All Configurations* and set *Configuration Type* to *Static Library (.lib)*. Remove **main.cpp** source file and build the library. Use *Batch Build* option in *Build* menu to build both *Debug* and *Release* version at the same time. All declared classes, functions and variables will be exported into a single **symbion1.lib** file.

We will now need to add an application project to test the library. Add a empty project to the solution named **test_log1**. There are two stages for building a project; compile and link. Compilation only requires the declaration so you can compile the code below as long as the library header file is included. Make sure that you also setup the *Build Dependencies* to **symbion1** project to make sure it is build first and also rebuild if it has been changed for testing.

Testing the library: test_log1\main.cpp

```
#include "..\symbion1\symbion.h"

int main() {
    auto pLog = symbion::CLoggerFactory::GetInstance();
    pLog->PutSource("test_log1");
    pLog->Message("Hello!");
    pLog->Message("Goodbye!");
}
```

To link in the compiled code from the library, Visual Studio needs the location and the library filename. In *Project Properties*, under the *Linker* section add the location using *Additional Library Directories* option to add the location of the library. Location of the Debug and Release version is different so set each configuration separately. The name can be set in the *Linker Input* page using *Additional Dependencies* option which is the same for all configurations. Alternatively you can simply rename the debug version to **symbion1d.lib** and release version to **symbion1r.lib** so both versions can be placed into one directory. Once Visual Studio can locate the library to link to, you should now be able to build and run the test project.

1.2 Dynamic Library

Make a copy of the **symbion1** project and rename the folder and project filename to **symbion2**. In *Project Properties* for *All Configurations*, change *Configuration Type* to *Dynamic Library (dll)*. One big difference between static and dynamic libraries is while all declared elements are exported by default in a static library, nothing is exported by default in a dynamic library. Also only functions are exported, no global variables. You can call C++ functions in the library to access the variables but you cannot directly access the variables directly from the application.

You need to add a **__declspec(dllexport)** to all functions to export them. Attach it to the class to export all the function members of a class. However this should only be done when you build the library while **__declspec(dllimport)** will be needed to use the library. It would be cumbersome to maintain two header files; one for export and another for import. All you need is an extra symbol added to the library project using the *Preprocessor Definitions* in the *C/C++ Preprocessor* page. Sometimes this symbol has already been added when you create a *Dynamic-Link Library* project. If you check there should be a **_WINDLL** symbol already there. Add following code in the header to generate a **SYMBION_API** symbol that can be either import or export depending if you are building or using a library.

Defining a symbol to export/import a library: symbion.h

```
#ifndef _WINDLL
#define SYMBION_API __declspec(dllimport)
#else
#define SYMBION_API __declspec(dllexport)
#endif
```

However this can be a problem if you are building a library that uses another library. So it would be better to use a more specific symbol than depend on **_WINDLL**. Add the following symbol to *Preprocessor Definitions* and update the header file to use it. Now all you need to do is to add **SYMBION_API** to all your classes.

Using a customized symbol

```
#ifndef SYMBION_EXPORTS
#define SYMBION_API __declspec(dllimport)
#else
#define SYMBION_API __declspec(dllexport)
#endif
```

Attaching export/import symbol

```
class SYMBION_API CBaseLogger {
            :
```

You can now test the library by creating a new empty project named **test_log2** using the same code as **test_log1**. However this time you will include **symbion2** header to compile and **symbion2.lib** for linking. However to run, **symbion2.dll** must be in the same directory as the program or from directories in **PATH** environment variable.

1.3 Distribution Files

To distribute your libraries to be used by other developers, you need to provide them the header files (**.h**) and library files (**.lib**) for all libraries. For a dynamic-link library you need to provide them the dynamic-link library file as well (**.dll**). Header files for compilation, library files for linking and dynamic-link library files for execution. When developers receives the files they should be able to install them into any location they wish. If they do use the libraries often, they should add a custom property sheet that points to the locations of the header and library files. Select **test_log2** project and locate *Property Manager* option from *View* menu. You should see different sections for Debug and Release and for 32-bit and 64-bit builds.

In *Debug x64* section, add a **Symbion.Debug.64** property sheet file and rename to **Symbion**. In *Release x64*, add a **Symbion.Release.64** property sheet file and also rename to **Symbion**. In *C/C++* section, add the location of the header file to both of property sheets. Add location of the library file to *Linker* section. Should be the same for dynamic library but different for static library. Then finally provide the name for the library in the *Linker Input* section. You should now be able include the header just by name and remove all other configurations from **test_log2** project. You just need to add **Symbion** property sheet files to any future projects that uses the library so no additional configuration is required to locate the header and library files.

Include header from configured directories: main.cpp

```
#include <symbion.h>

int main() {
    auto pLog = symbion::CLoggerFactory::GetInstance();
    pLog->PutSource("test_log1");
    pLog->Message("Hello!");
    pLog->Message("Goodbye!");
}
```

1.4 Boost Library

Boost is an external collection of more than 164 additional C++ libraries. In fact some of the Standard C++ library started in Boost and then adopted into the C++ Standard library. You can download and install the latest version of Boost for your platform from **http://boost.org** website. Once downloaded you can use create a sample project to setup Visual C++ property sheets to access the Boost libraries.

Create an empty project named **Boost1** and make sure *Configuration Type* has been set to *Application* and *Linker SubSystem* is *Console* for all *Build Configurations* and all *Platforms* in the *Project Properties*. Then add a source file named **main.cpp** with the following content.

Basic console app entry point: main.cpp

```
#include <iostream>
int main() {
    return 0;
}
```

Open up the *Property Manager* from *View* | *Other Windows* menu and you should see the available *Build Configurations* for example *Debug Win32*, *Release Win32*, *Debug x64* and *Release x64*. Each configuration already has a set of default property sheets. Since these are the defaults, it is better not to change them since some of them are used across multiple configurations. Instead you should add custom property sheets. Right-click on any configuration and add a new property sheet named **Boost** for any or all configurations and store all the property sheets in a shared location. Edit your property sheet and use the *Additional Include Directories* option in *C/C++ General* page to add the folder where the Boost library is installed. Use the *Linker General* and *Input* pages to setup the directories and names for static libraries. However Boost is complete template-based so only header files will be required. Add the same property sheet to all of the other configurations or only those that you would use by selecting the configurations before adding the property sheet. So it will be simple to add Boost support to any project by just adding the property sheet those projects that require the library.

Let's do a simple example on using a timer from the Boost asynchronous I/O library. This library is useful for implementing networking applications. We should now be able to include appropriate Boost library header files, then use Boost types and functions provided, and then compile and build the program.

Using timer from Boost library: main.cpp

```
#include <boost/asio.hpp>
#include <iostream>

using namespace boost::asio;

int main() {
    io_context io;
    auto duration = chrono::seconds(5);
    auto timer = steady_timer(io, duration);
    std::cout << "Begin" << std::endl;
    timer.wait();
    std::cout << "Completed" << std::endl;
    return 0;
}
```

1.5 Precompiled Headers

C++ header files can be huge if it contains thousands of declarations of macros, types and templates. Reprocessing all the header files everytime you build your projects can hugely impact build time. All the header files that you use can be pre-processed into a faster format and saved as a Precompiled header files (*.pch). These files will be used the next time you rebuild the project. Normally the projects that you create is already setup to use this feature but not for an empty project. However we can easily enable this feature manually. Add an extra header file and an extra source file. The name of these files does not matter. For our sample project, add **pch.h** and **pch.cpp** files. Add includes for all header files into **pch.h** that you wish to pre-compile and then include it in **pch.cpp** to generate the pre-compiled headers.

The PCH header file: pch.h

```
#include <boost/asio.hpp>
#include <iostream>
```

The PCH source file: pch.cpp

```
#include "pch.h"
```

Right-click over the source file and select Properties option. In C/C++ section, open *Precompiled Headers* page and set *Precompiled Header* option to *Create*. Make sure that the name of the header file is correct. This source file will automatically be used to generate the pre-compiled headers and the source files will automatically use it by default. Again make sure to select *All Configurations* and *All Platforms* before making changes. You can now include **pch.h** in all of the source files that you want to use the pre-compiled headers. The first time you build the project or after you modify the header file will take more time to generate the headers. The subsequent build time is much faster. You can use the **symbion2** project as an additional exercise, collect all the header files used by the source code including **symbion.h** into a single header file and include that file into all the source files. Then generate precompiled headers for that one single header.

2 ANSI & Unicode Support

2.1 C Library

The Windows operating system has Unicode support since Windows NT. ANSI is still supported for backward compatibility for old Windows applications that was built for Windows 95, 98 and ME. Each ANSI character takes up one byte while each Unicode character takes up two bytes. In C++, you can use **char** type for an ANSI character and **wchar_t** for Unicode. You can also use the **CHAR** and **WCHAR** macro defined for these types. Use **TCHAR** if you wish to be able to compile a program to either an ANSI or Unicode. If the **UNICODE** symbol is defined before compilation, then it will be translated to **wchar_t** otherwise it will be translated to **char**. Include **TCHAR** header if you use this type.

ANSI & Unicode character buffers: Unicode1\main.cpp

```
CHAR b1[128];       // 128 bytes ANSI char array (128 bytes)
WCHAR b2[128];      // 128 bytes Unicode char array (256 bytes)
TCHAR b3[128];      // either ANSI or Unicode char array
```

C++ literal strings are compiled to ANSI by default. To use Unicode literal strings, you have to attach an **L** prefix. To be able to compile the string to either ANSI or Unicode, you can use the **TEXT** or **_T** macro which will attach an **L** prefix to the string only if the **UNICODE** symbol has already been defined before compilation. For pointers to strings and character arrays, use **LPSTR** and **LPCSTR** for ANSI, **LPWSTR** and **LPCWSTR** for Unicode and lastly **LPTSTR** and **LPCTSTR** for neutral.

ANSI & Unicode literal strings

```
LPCSTR  p1 = "This is an ANSI string.";
LPCWSTR p2 = L"This is an Unicode string.";
LPCTSTR p3 =  TEXT("This can be an ANSI or Unicode string.");
LPSTR s1 = b1; LPWSTR s2 = b2; LPTSTR s3 = b3;
```

The C library usually provides a set of string functions for manipulating ANSI strings. Microsoft has added additional functions to support Unicode strings. The name of the functions has additional **w** character for wide-character support.

ANSI & Unicode string functions

```
strcpy(s1,p1);      // ANSI string function
wcscpy(s2,p2);      // Unicode string function
_tcscpy(s3,p3);     // either ANSI or Unicode function
```

Visual C++ also provide a set of non-standard safer set of functions that forces you to pass in the length of the buffer to ensure that there is no buffer overrun. An exception will be thrown immediately if there is an overrun.

Safe version of string functions

```
strcpy_s(s1,sizeof(b1),p1);
wcscpy_s(s2,sizeof(b2) / 2,p2);
_tcscpy_s(s3,sizeof(b3) / sizeof(TCHAR), p3);
```

2.2 Windows API

The Windows API will also provide two functions for operations that accept any strings one postfixed with **W** and another with **A**. When the **A** function is called, ANSI strings passed to the function are converted into Unicode and will be then forwarded to the **W** function which performs the actual task. Any string passed back by the **W** function is converted back to ANSI before returning back from the **A** function. Calling functions that ends with **W** will thus be faster. You can leave out the postfix altogether and let the compiler determine the function to call based on whether a **UNICODE** symbol has been defined. You can use Dependency Walker utility (depends.exe) to examine the Windows API Libraries to see the actual function names.

Calling ANSI function from kernel32.dll

```
void GetComputerName1() {
      char computerName[256];
      DWORD dwLen = sizeof(computerName);
      GetComputerNameA(computerName, &dwLen);
      MessageBoxA(NULL, computerName, "Computer Name", MB_OK); }
```

Calling Unicode function

```
void GetComputerName2() {
      wchar_t computerName[256];
      DWORD dwLen = sizeof(computerName) / 2;
      GetComputerNameW(computerName, &dwLen);
      MessageBoxW(NULL, computerName, L"Computer Name", MB_OK); }
```

Calling either ANSI or Unicode

```
void GetComputerName3() {
      TCHAR computerName[256];
      DWORD dwLen = sizeof(computerName) / sizeof(TCHAR);
      GetComputerName(computerName, &dwLen);
      MessageBox(NULL, computerName, TEXT("Computer Name"), MB_OK); }
```

You should use **TCHAR**, **LPTSTR**, **LPCSTR** types and **TEXT** or **_T** macro so that you can compile a Win32 application to ANSI or Unicode. You can select the *Character Set* to use in *Project Properties*.

Supporting both ANSI and Unicode

```
int main() {
    GetComputerName1(); // ANSI version
    GetComputerName2(); // Unicode version
    GetComputerName3(); // ANSI or Unicode version
}
```

2.3 C++ Library

Even though the standard C++ library has support for Unicode, it does not provide macros that can be used to compile the same C++ code to either ANSI or Unicode. However you can easily implement these macros yourself. Use **string** class for ANSI and **wstring** class for Unicode. For console input and output of ANSI strings, you can use **cin** and **cout** objects. For Unicode strings, you need to use **wcin** and **wcout** instead. To support both ANSI and Unicode you can define your own T-style macros by checking for the **UNICODE** symbol.

T-style macros for standard C++ strings: Unicode2\main.cpp

```
#ifdef UNICODE
    #define _tstring    wstring
    #define _tcin       wcin
    #define _tcout      wcout
    #define _t(str)     L##str
#else
    #define _tstring    string
    #define _tcin       cin
    #define _tcout      cout
    #define _t(str)     str
#endif
```

Using macros to compile to either ANSI or Unicode

```
#include <iostream>
void main() {
    std::_tstring t1(_t("Hello!"));             // either wstring or string
    std::_tstring t2(_t("Goodbye!"));           // either wstring or string
    std::_tcout << t1.c_str() << std::endl      // either wcout or cout
                << t2.c_str() << std::endl;
}
```

Compatibility with TCHAR

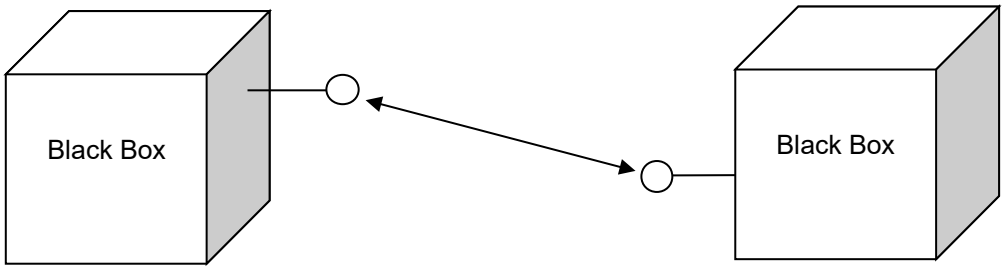
```
#include <tchar.h>
void main() {
    TCHAR buffer[256];
    std::_tcout << _t("Enter your name: ");
    std::_tcin.getline(buffer, sizeof(buffer) / sizeof(TCHAR));
    std::_tstring name(buffer);
    std::_tcout << _t("Hello ") << name.c_str() << _t('!') << std::endl;
}
```

3 Interface-Based Programming

3.1 Interface Declaration

Some components may be reusable across multiple sub-systems. We can implement a high-level of abstraction if each component is designed to be totally independent from other components. Even though there would not be any physical link between components, they can still communicate within the system with one another using a set of common interfaces.

Using abstract interfaces to connect components



Abstraction would also allow us to achieve true polymorphism where it is possible to substitute one component with other different components at runtime. Factories can be established to create instances from different types based on current environment or configuration settings. The sub-system does not need to know what type or library the instance comes from as long as it has the required interface.

Interfaces simplify the development and usage of polymorphic components. Since the copy constructor is not exposed through the interface, passing and returning by value is not possible, thus removing the issues involved with copying objects with dynamic resources. You are not forced to extend from a specific base class to get compatibility, you can choose to re-implement the entire interface if you wish.

The public members of a class can be collectively called an interface as it is what we need to use to communicate or get access to the private internals of objects. Modern object-oriented programming languages including C# and Java allows you to declare an interface separately from the class. An interface is completely abstract and has no implementation. Even though there is no **interface** keyword in C++, you can declare a pure abstract structure or class. Interface-based programming makes polymorphism more simple to implement and eliminates many common issues that we face when we use C++ to implement classes. To demonstrate this, create a new empty Win32 static library project called **MyLog**.

Example of an interface in C#

```
enum LogType {
    Information,
    Warning,
    Error
}

public interface ILogger {
    string Source { get; set; }
    void Write(string message, LogType type = LogType.Information);
    void Message(string message);
    void Warning(string message);
    void Failure(string message);
}
```

Project Information

```
Project Name: MyLog
Project Type: Visual C++ | Win32 Project
Settings    : Static Library, Empty
Location    : C:\CPPDEV\SRC
Solution    : Module3
```

Add a header file named **MyLog.h** to the project. Declare TCHAR-style macros so you can use C++ string types and compile the library to support ANSI or Unicode. Include other required headers and then declare a **mylog** namespace to put everything else including function declarations, type declarations and type definitions.

Basic header file for library: MyLog.h

```
#pragma once

#ifdef UNICODE
#define _tstring    wstring
#define _tcout      wcout
#define _tcin       wcin
#define _t(str)     L##str
#else
#define _tstring    string
#define _tcout      cout
#define _tcin       cin
#define _t(str)     str
#endif

#include <cstdint>
#include <string>

namespace mylog {

}   // namespace mylog
```

You can now declare the **LogType** enumeration and a **GetLogTypeString** function to return a string representation of each enumerator. This is a global function so it is not associated with or dependent on any class.

Declaration of enumeration and support function

```
enum struct LogType : uint16_t {
    Information,
    Warning,
    Error
};

const std::_tstring& GetLogTypeString(LogType type);
```

You can now declare an interface named as **ILogger**. An interface should be a pure abstract struct or class in C++. Usually the C++ compiler will generate a virtual table for any type that has at least one virtual function to store the address of the functions but since every function is abstract there is no need for a virtual table. You can inform C++ not to generate a virtual table for the interface using a **__declspec(novtable)** option. You can only access an existing object through an interface. You cannot create an object by using the interface so it does not contain any constructors. A separate function can help you to create the object and return a pointer to the interface for that object. You can then access the object through the pointer. An interface pointer is basically the address of the virtual table. You can then call functions in the object by using their addresses in the table. Thus there is no need for you to know what is the type of the object. As long as it has a virtual table compatible to the interface you can call its virtual member functions.

Since you do not call the constructor through an interface than you must not call the destructor as well. Instead you can add a **Release** function that releases or destroys the object for you once you have finished using it. You also need a type definition for a pointer that you can use to call a function that instantiates the object for you.

An interface declaration in C++

```
struct __declspec(novtable) ILogger {
    virtual void PutSource(const std::_tstring& source) = 0;
    virtual const std::_tstring& GetSource() const = 0;
    virtual void Write(const std::_tstring& message,
        LogType type = LogType::Information) const = 0;
    virtual void Message(const std::_tstring& message) const = 0;
    virtual void Warning(const std::_tstring& message) const = 0;
    virtual void Failure(const std::_tstring& message) const = 0;
    virtual void Release() const = 0;
    _declspec(property(get = GetSource, put = PutSource))
            const std::_tstring& Source;
};
```

Pointer type definition for function that returns interface pointer

```
typedef ILogger *(*GetLoggerPtr)();
```

3.2 Interface Implementation

A pure abstract class is only used for polymorphism and not for inheritance as it does not have any data or code to inherit. You can still provide a default implementation of the interface to be used for inheritance. As it is not used for polymorphism it can be a complete working class rather than an abstract base class. This class can also be used as a fallback in case no other implementation is available at the time. Since this is the complete class it will have additional members such as constructors, destructor and member variables.

Default implementation class for interface

```
class CDefaultLogger : public ILogger {
private:
    std::_tstring m_source;
public:
    CDefaultLogger();
    void PutSource(const std::_tstring& source);
    const std::_tstring& GetSource() const;
    void Write(const std::_tstring& message, LogType type) const;
    void Message(const std::_tstring& message) const;
    void Warning(const std::_tstring& message) const;
    void Failure(const std::_tstring& message) const;
    void Release() const;
    virtual ~CDefaultLogger();
};
```

You can now implement the code for the global function and the class members. Since each source file is compiled into a single object code file you should put functions that may not be used together into separate source files. An implementation class might choose not to derive from **CDefaultLogger** but may still call the **GetLogTypeString** function. So when the linker links in an entire object code file it does not contain code that you never use.

Separate implementation file for function: MyLog1.cpp

```
#include "MyLog.h"
#include <array>
namespace mylog {
    const std::_tstring& GetLogTypeString(LogType type) {
        static std::array<std::_tstring, 4> strings = {
            std::_tstring(_T("Information")),
            std::_tstring(_T("Warning")),
            std::_tstring(_T("Error")),
            std::_tstring(_T("Message"))
        };
        auto index = (uint16_t)type;
        if (index < 0 || index > 2) index = 3;
        return strings[index];
    }
}   // namespace mylog
```

Separate implementation file for class: MyLog2.cpp

```
#include "MyLog.h"

namespace mylog {
    CDefaultLogger::CDefaultLogger() {
        PutSource(_T("Log"));   // or Source = _T("Log");
    }

    void CDefaultLogger::PutSource(const std::_tstring& source) {
        m_source = source;
    }

    const std::_tstring& CDefaultLogger::GetSource() const {
        return m_source;
    }

    void CDefaultLogger::Write(const std::_tstring& message,
        LogType type) const {
        TCHAR text[1024];
        const std::_tstring& type_str = GetLogTypeString(type);
        _stprintf_s(text, _T("[%s]%s(\"%s\")\n"),
            m_source.c_str(), type_str.c_str(), message.c_str());
        OutputDebugString(text);
    }

    void CDefaultLogger::Message(const std::_tstring& message) const {
        Write(message, LogType::Information);
    }

    void CDefaultLogger::Warning(const std::_tstring& message) const {
        Write(message, LogType::Warning);
    }

    void CDefaultLogger::Failure(const std::_tstring& message) const {
        Write(message, LogType::Error);
    }

    void CDefaultLogger::Release() const {
        delete this;    // destroys the current object
    }

    CDefaultLogger::~CDefaultLogger() {
        //  blank implementation to be inherited by derived classes
    }
}   // namespace log
```

The above is a fully working class and not just for inheritance or polymorphism but in case there are derived classes, the destructor must still be virtual to ensure that the correct destructor is called. There is no need to declare properties in the class as they can be inherited from the interface. Since the interface is used to call a function, any default arguments should be declared in the interface rather than in the class.

3.3 Object Activation

You also need to implement a helper object commonly called as the object activator to help you instantiate an actual object. To implement true polymorphism the activator must be able to create an object from any compatible type not only the types that are compiled into the same library or application. Following is a declaration of a class that can help provide objects from any external DLL at runtime without having to link to it during compilation.

Class declaration of object factory

```
class CLoggerFactory {
private:
    HMODULE m_hModule;
    GetLoggerPtr m_pGetLogger;
public:
    CLoggerFactory();
    CLoggerFactory(const CLoggerFactory& obj);
    ILogger *GetInstance() const;
    ~CLoggerFactory();
};
```

You can load in any DLL at runtime by name using a **LoadLibrary** function. The name can be stored externally so that the DLL can be changed without having to re-compile the application. Once the library is loaded you will get a **HMODULE** handle to access the library. Call **FreeLibrary** to release a library when you no longer need it. A DLL is loaded only once for multiple applications and only unloads when released by all the applications.

Once a library has been loaded you can obtain an address of a function in the export table by name using **GetProcAddress**. You can then call the DLL function by using its address. Our factory will look for the address of a **GetLogger** function that is suppose to instantiate an **ILogger** compatible object and return an interface pointer.

CDefaultLogger will be used instead if an external DLL is not available or it does not have a **GetLogger** function. This guarantees the application will still work if no other logging component is available or made available at runtime.

The object being returned will most probaby be a dynamic object so it is important to remember to call **Release** after using it to avoid memory leaks. Alternatively you can also implement a smart pointer for **ILogger** that will automatically call **Release** when the smart pointer object is destroyed.

Separate implementation for factory class: MyLog3.cpp

```
#include "MyLog.h"

namespace mylog {
    CLoggerFactory::CLoggerFactory() :
        m_hModule(nullptr), m_pGetLogger(nullptr) {
        TCHAR moduleName[MAX_PATH];
        if (GetEnvironmentVariable(_T("ILogger"),
            moduleName, MAX_PATH) == 0) return;
        if ((m_hModule = LoadLibrary(moduleName)) == nullptr) return;
        m_pGetLogger = (GetLoggerPtr)GetProcAddress(m_hModule, "GetLogger");
    }

    CLoggerFactory::CLoggerFactory(const CLoggerFactory& obj) :
        m_hModule(nullptr), m_pGetLogger(nullptr) {
        if (obj.m_hModule == nullptr) return;
        auto process = GetCurrentProcess();
        DuplicateHandle(
            process, obj.m_hModule,
            process, (LPHANDLE)&m_hModule,
            NULL, FALSE, DUPLICATE_SAME_ACCESS);
        m_pGetLogger = (GetLoggerPtr)GetProcAddress(m_hModule, "GetLogger");
    }

    ILogger *CLoggerFactory::GetInstance() const {
        return (m_pGetLogger != nullptr) ?
            m_pGetLogger() : new CDefaultLogger();
    }

    CLoggerFactory::~CLoggerFactory() {
        if (m_hModule != nullptr) FreeLibrary(m_hModule);
    }
}   // namespace log
```

Add a console application named **TestLog**. Set the application to be dependent on the library to ensure that the library always get built first before the application. Include the header file in the following main program and configure the directory and name of the library in the project's Linker settings.

Accessing object through an interface: TestLog\main.cpp

```
#include "..\MyLog\MyLog.h"

using namespace mylog;

void main() {
    CLoggerFactory factory;
    auto pLog = factory.GetInstance();
    pLog->Write(_T("Hello!"));
    pLog->Write(_T("Goodbye!"));
    pLog->Release();
}
```

3.4 True Polymorphism

Once the above program has been compiled, you will never need to change it again. It can use any class as long as it implements **ILogger** or derive from any class that already implements the interface. Each logger can be packed into a separate DLL and exports a **GetLogger** function that returns an instance of the logger class. Since the class is never used directly, only through the interface, there is no need to export the class at all. You do not even need a header file for the logger class as demonstrated in the following content. Create an empty Win32 DLL project named FileLogger. Then add one source file for the entire class. You must link to the **MyLog** static library since it needs to inherit the **CDefaultLogger** class. To rewrite the entire class, then only the **ILogger** interface in the header file is required. Following is the implementation of a **CFileLogger** class. To support writing to ANSI and Unicode files you need to add additional macros to switch between **fstream** for ANSI and **wfstream** for Unicode. Add **GetLogger** function to instantiate an object from the class and return **ILogger** interface pointer. This function will be called by **CLoggerFactory** class.

A different implementation of ILogger: FileLogger.cpp

```
#include "..\MyLog\MyLog.h"
#include <ctime>
#include <fstream>

#ifdef UNICODE
#define _tfstream wfstream
#else
#define _tfstream fstream
#endif

class CFileLogger : public mylog::CDefaultLogger {
public:
    void CFileLogger::Write(const std::_tstring& message,
        mylog::LogType type) const override {
        tm datetime;
        auto now = time(nullptr);
        localtime_s(&datetime, &now);
        TCHAR text[2048];
        _stprintf_s(text, _T("[%04d-%02d-%02d %02d:%02d:%02d]%s(\"%s\")"),
        //  tm_year is relative to 1900, tm_mon is 0 for January
            datetime.tm_year + 1900, datetime.tm_mon + 1, datetime.tm_mday,
            datetime.tm_year, datetime.tm_mon, datetime.tm_mday,
            datetime.tm_hour, datetime.tm_min, datetime.tm_sec,
            GetLogTypeString(type).c_str(), message.c_str());
        std::_tstring contents(text);
        std::_tstring filename(Source);
        filename.append(_T(".log"));
        std::_tfstream file(filename, std::ios::app);
        file << contents << std::endl;
        file.close();
    }
};
```

Usually C++ function names will be decorated to allow multiple functions to have the same name. C does not support overloading and thus the name is not decorated. Use **extern "C"** to ensure that the function name is not decorated in C++ to make it easier for the factory to fetch the address of the function.

Function to create and return an instance of CFileLogger

```
extern "C" mylog::ILogger *GetLogger() {
    return new CFileLogger();
}
```

The **__declspec(dllexport)** is needed for exporting decoration C++ function names. You just need to create a module definition file to export C-style function names. Add the following file to the project to export the above function.

Exporting C-style function names: FileLogger.def

```
LIBRARY FileLogger
EXPORTS
    GetLogger
```

You can now provide another implementation of **ILogger**. Add another empty Win32 DLL project named EventLogger. Then implement the following class and the function. Remember to create a module-definition file to export the function. As usual set the dependency to **MyLog** library and link to it.

Another implementation of ILogger: EventLogger.cpp

```
#include "..\MyLog\MyLog.h"

class CEventLogger : public mylog::CDefaultLogger {
public:
    void CEventLogger::Write(const std::_tstring& message,
        mylog::LogType type) const override {
        WORD eventType = 0;
        switch (type) {
            case mylog::LogType::Information:
                eventType = EVENTLOG_INFORMATION_TYPE; break;
            case mylog::LogType::Warning:
                eventType = EVENTLOG_WARNING_TYPE; break;
            case mylog::LogType::Error:
                eventType = EVENTLOG_ERROR_TYPE; break;
        }
        LPCTSTR messages[] = { message.c_str() };
        HANDLE hEventLog = RegisterEventSource(NULL, Source.c_str());
        ReportEvent(hEventLog, eventType, 0, 0, NULL, 1, 0, messages, NULL);
        DeregisterEventSource(hEventLog);
    }
};

extern "C" mylog::ILogger *GetLogger() {   return new CEventLogger(); }
```

Exporting the GetLogger function: EventLogger.def

```
LIBRARY EventLogger
EXPORTS
    GetLogger
```

There is no need to re-compile **TestLog** to use the new logging component. Just set the following environmental variable before running the application. **CLoggerFactory** will load in the correct DLL and retrieve the address of the **GetLogger** function that is then called to obtain a **ILogger** object. The application doesn't care about what type of object it us as long as it can use the object through **ILogger**.

Setting ILogger environment variable to FileLogger

```
Set ILogger=C:\CPPDEV\SRC\Module2\Debug\FileLogger.dll
```

Setting ILogger environment variable to EventLogger

```
Set ILogger=C:\CPPDEV\SRC\Module3\Debug\EventLogger.dll
```

Different developers can freely choose how to implement their loggers as long as they implement **ILogger** interface. However the components that we implemented can only be used in C++. If you wish to build components that is language independent, use *Component Object Model* (COM) technology. COM objects are accessed through interfaces. Any language that can keep a pointer to a virtual table and call functions through addresses can support COM, even non object-oriented languages like C.

4 Component Object Model

4.1 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 step or 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 a COM server. You can add a new ATL Project named **LogCom** in the solution and accept the default settings.

Project Information

```
Project Name: LogCom
Project Type: ATL Project
```

Notice that two projects are created; **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.

4.2 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, add a ATL Simple Object and named it as **Log**. A dialog will appear for you to enter the class details. Just enter the short name for your COM component and Visual Studio generates all the files for you. You can customize the class name and also the file names if you wish.

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>
{
        :
}
```

4.3 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.

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.

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.

4.4 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.

Members to add or update in class: Log.h

```
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.

COM class extended from ATL templates: Log.h

```
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: Log.h

```
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;
}
```

4.5 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 will 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.

4.6 COM Client

A COM client is basically a program that uses COM servers. Since COM is a feature of the Windows operating system most programming languages built for use in Windows will have support for COM. The simplest way to write COM clients is to use VBScript or JScript scripting languages. In VBScript you can use **CreateObject** function to create a COM object by using the programmatic identifier (ProgID) of the COM server. Once you no longer want to use the object you can release it by assigning **Nothing** to the variable that references the object. In VBScript you must use **SET** command to assign an object to a variable. For JScript you can use **new** to instantiate an **ActiveXObject** object and the ProgID is passed to the constructor. Assign **null** to the object variable to release it when it is no longer used.

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
```

Scripting languages uses a special interface named **IDispatch** to access COM objects rather than a specific custom interface like **ILog**. The dispatch interface is tedious and slow to be used from compiled languages so custom interfaces are used instead. You can always create a dual interface to support both scripting and compiled languages. This basically means the a custom interface is extended from dispatch so that it also implements **IDispatch**. This is the default setting when you create a COM component in Visual C++ using ATL.

5 Common Language Infrastructure

5.1 Common Language Runtime

C++ Common Language Infrastructure (C++/CLI) is a C++ programming language that has been extended to build applications and libraries that execute in a managed environment called Common Language Runtime (CLR). The implementation of CLR by Microsoft for Windows operating systems is called Microsoft.NET Runtime distributed together with numerous libraries, language compilers and tools as one package called Microsoft.NET Framework. Advantage of building CLR-based applications and libraries, called as assemblies, is guaranteed interoperability between them regardless of which programming language was used to construct them. Microsoft.NET Framework comes with compilers for many programming languages including C#, F#, Visual Basic.NET, JScript.NET and C++/CLI while compilers for other languages like Pascal and Python are provided by third-party developers. Most of these are high-level programming languages that have no direct access to operating system and native libraries that are commonly build with C/C++ except for C++/CLI, which has direct access to both CLR and native libraries inclusive of Windows operating system libraries. Thus C++/CLI is a perfect solution for the interoperability of high-level applications written with C# to low-level native libraries written with C/C++.

5.2 Project Setup

First, run the Visual Studio Installer to ensure that C++/CLI support option is enabled for the tools and templates required for C++/CLI development in Visual Studio to be installed. Start Visual Studio and using the following information to create a project to build a C++/CLI class library. You can use .NET Framework for Windows or use .NET for cross-platform.

Library Project Information

```
Project Name: LogCLR
Project Type: CLR Class Library
```

The new project is already configured to use pre-compiled headers so you should see a **pch.h** header file and a **pch.cpp** source file in the project. To use them, place all includes into the header file and the source file will generate pre-compiled headers from those includes. Then make sure **pch.h** is the first include in all the source files in your project. It must be done even if you do not wish to use pre-compile headers or you may get a compilation error from Visual Studio.

The project also contains a header file and source file with the project name to create an initial CLR class. Delete both **LogCLR.h** and **LogCLR.cpp** from the project as we wish to start from a completely empty project. Select the *Build Project* option from the *Build* menu to ensure that you can successfully generate an assembly from the project.

Assembly built from the project

```
C:\CPPDEV\SRC\Module2\Debug\LogCLR.dll
```

We will now create a project to build an application to test the library. We will use C# as the coding language for the application. Use the following information to create the project. Right-click on the solution in *Solution Explorer* window and select *Add* | *New Project...* option to add a new project to the same solution.

C# Project Information

```
Project Name: TestLogCLR
Project Type: C# Console App (.NET Framework)
```

It is simple for one assembly to use other assemblies by just referencing them during compilation. Right-click on the *References* node in the project to select the assemblies to reference. You can select standard assemblies from Global Assembly Cache (GAC), assemblies build from projects in the same solution, or use a *Browse* button to select the actual assembly file. If you have the project that builds the assembly, referencing the project as you get automated builds whenever changes are detected. This ensures that the application project is always referencing the latest build of the library.

Note that a C# project is compiled to Common Intermediate Language (CIL) code and not native code. CIL code is translated to native code by the CLR when the assembly is executed. Whether the code is 32-bit or 64-bit depends on which version of the CLR is running on the system. This guarantees that your assembly will automatically take advantage of the operating system and processor architecture, called as the platform, no matter which platform it is running on. This is why there is only the *Any CPU* Build Configuration available on the C# project. However C++/CLI and C/C++ projects are always compiled to native code so you need to select the correct Build Configuration to use; Win32 for 32-bit and x64 for 64-bit. Use the *Configuration Manager* option in *Build* menu to select the default configuration to use per project.

A 32-bit library cannot be loaded into a 64-bit process and a 64-bit library cannot be loaded into a 32-bit process. If the C# application is running on a 64-bit platform but referencing a 32-bit C++/CLI library, it will fail to load. Similarily if a 32-bit C++/CLI library tries to load a 64-bit C/C++ native library, it would also fail to load. Thus it is important the platform must be aligned across the application, libraries and the native libraries. So to use 64-bit native libraries, make sure **LogCLR** project is set to use the x64 Build Configuration. For the C# project, right-click on the project node in Solution Explorer and select *Properties* option. In *Build* page for *All Configurations* set *Platform Target* specifically to x64. If the native libraries are 32-bit, set all projects to Win32 instead before building the solution.

Note that the location of the C++/CLI library is different depending on configuration and platform. Following is the location of the library for the Debug configuration, x64 platform.

Assembly built with Debug configuration, x64 platform

```
C:\CPPDEV\SRC\Module2\x64\Debug\LogCLR.dll
```

To run or debug an application in Visual Studio, select the project in Solution Explorer, right-click and select *Set as StartUp Project* option. There should not be any warnings or errors during compilation and the application should execute successfully. However the application cannot do anything with the library at the moment since it does not have any public CLR type that can be used by the application.

5.3 Managed Types

You can use C libraries, C++ libraries and .NET libraries in a CLR project, however a .NET client can only access managed types from a CLR project. For enumerations, you need to use **enum class** to define a .NET accessible enumeration. Add a **Log.h** header file to declare the following namespace and enumeration.

Declaring a .NET accessible enumeration: Log.h

```
namespace LogCLR {
    public enum class LogType {
        Information,
        Warning,
        Error
    };
}
```

You need to declare a **public ref class** to define a .NET accessible class that is called as a managed class. Objects created from this class is always passed and returned by a managed reference using the **^** operator. C++ pointers (*) and references (&) is not used for managed objects. Member variables and functions can use and return the basic C++ types but no C++ objects even though C++ objects can be used within the implementation. For example use **.NET System::String** managed class to store strings in a managed object. A managed class also support properties which can have a **get** and a **set** function. Following shows declaration of the **Log** managed class.

Managed class with member variable (field) and constructor

```
public ref class Log {
    private:
        System::String ^source;
    public:
        Log();
}
```

Member functions (methods)

```
void Write(System::String ^message, LogType type);
void Message(System::String ^message);
void Warning(System::String ^message);
void Failure(System::String ^message);
```

Accessor functions (properties)

```
property System::String^ Source {
    System::String^ get() { return source; }
    void set(System::String^ value) { source = value; }
}
```

You can now implement the code for the member functions. Note that you can put all the code in the header file, however C++ header files are not automatically compiled so you still need a source file to include the header file to get it compiled even though the source file does not contain any code. The code uses **.NET String** class to format the output string and **.NET Debug** class to output the string. You are free to use any C/C++ types in the code.

Managed class implementation: Log.cpp

```
#include "pch.h"
#include "Log.h"

namespace LogCLR {

Log::Log() {
    source = L"Log";
}

void Log::Write(System::String ^message, LogType type) {
    auto text = System::String::Format("[{0}]{1}(\"{2}\")",
    source, type, message);
    System::Diagnostics::Debug::WriteLine(text);
}

void Log::Message(System::String ^message) {
    Write(message, LogType::Information);
}

void Log::Warning(System::String ^message) {
    Write(message, LogType::Warning);
}

void Log::Failure(System::String ^message) {
    Write(message, LogType::Error);
}
```

5.4 Namespace Aliases

While you can import .NET namespaces, there may still be problems if there are types with the same name in those namespaces, so you still may need to use namespaces to refer to managed types. However since .NET namespaces can be very long, it will be better to use namespace aliases to shorten them. You can now add the following namespace aliases in the header file and then replace **System** with **sys** and replace **System::Diagnostics** with **diagnostics** across all your files.

Assigning namespace aliases: Log.h

```
namespace sys = System;
namespace diagnostics = System::Diagnostics;
```

5.5 Testing the Library

The **Main** method of a C# application will usually be in the **Program** source file. You can write the following to create a Log object and access its properties and methods. Make sure the CLR library is referenced in the project.

A C# application using a C++ CLR library: TestLogCLR\Program.cs

```
namespace TestLogCLR {
    class Program {
        static void Main(string[] args) {
            var log = new LogCLR.Log();
            log.Source = "TestLogCLR";
            log.Message("Hello!");
            log.Message("Goodbye!");
            log = null;
        }
    }
}
```