Visual Studio

Module 5

Implementing C++ Types & Libraries

1 Implementing C++ Types

1.1 Generic Types & Functions

Whenever possible, implementing generic types and functions to support a variety of types and allowing your functions to work with different containers and iterators, and because they can be completely implemented in a header file, there is no need to link to libraries to use them. For example, here is a simple generic function that can work with all containers by writing iterator access code in the function. This function is able to display all types of items in any container that has iterator support.

Using iterator access code in function: MyTypes.h

```
#include <iostream>

namespace my {
    template<typename T>
    void display(T p, T e) {
        while (p != e) std::cout << *(p++) << std::endl;
    }
}
```

Passing iterators to function

```
#include <vector>
#include <deque>
#include "MyTypes.h"

void main() {
    std::vector<int> l1 = { 10, 20, 30, 40, 50 };
    std::deque<std::string> l2 =  { "John", "Mary", "Anne", "Lilo" };
    my::display(l1.begin(), l1.end());
    my::display(l2.begin(), l2.end());
}
```

1.2 Initializer List Support

Previously only arrays and simple structures can be initialized using initializer lists. In C++/11, now everything can be initialized including basic types. To initialize to default value, use an empty list. The assignment operator is optional. If only one value is in a list, the C++ will automatically find a constructor that accepts the arguments. You can also write a constructor that accepts a specific **initializer_list** container

Initializing basic types: Initializers1\main.cpp

```
int v1 = {};     // initialize to default value for int (0)
short v2 {};     // initialize to default value for short (0)
bool  v3 {};     // initialize to default value for bool (false)
bool v4 {true}; // initialize to true
auto v5 {123};  // initialize int to 123
```

C++ selects the appropriate constructor

```
class A {
public:
    A(int n) { std::cout << n << std::endl; }
    A(double d) { std::cout << d << std::endl; }
    A(int n, double d) { std::cout << n << ',' << d << std::endl; }
};

void main() {
    A obj1{ 123 };
    A obj2{ 1.99 };
    A obj3{ 123, 1.99 };
}
```

Another example below creates a function that can total the elements of any type in a **vector** container. The following also shows a new C++/11 feature where initializer list can be applied to all types. Empty list returns the default value for the type.

Generic function to sum vectors of any type

```
template<typename T>
T sum_values(vector<T> values) {
    T total = {};   // initialize to default type value
    for (auto value : values)
        total += value;
    return total;
}
```

You can also have a constructor that can accept a specific **initialize_list** container. It will allow you to pass in a variable number of initialization values. For example we can add the constructor to **CContact** class to take values from this container instead.

Supporting initializer_list container

```
CContact(initializer_list<const char *> values) {
    if (values.size != 2) exception("not enough initializers");
    auto ptr = values.begin();
    m_name = *(ptr++);
    m_email = *ptr;
}
```

1.3 Extend Existing Types

Rather than implemently completely new types you can always extend existing types. For example rather than implementing your exception class, you can extend the C++ standard library exception class and add missing features. It also guarantees that it is compatible to the base class.

Extending exceptions

```
#include <exception>

class my_exception : public std::exception {
    std::string m_file;
    int m_line;
public:
    my_exception(
        const char * message,
        const char * file, int line)
        : std::exception(message), m_file(file), m_line(line) { }
    const std::string& file() const { return m_file;  }
    int line() const { return m_line; }
};
```

Throwing a custom exception

```
throw my_exception("Error message", __FILE__, __LINE__)
```

1.4 Nullable Values

In C/C++, only pointers can be null. Sometimes a function may use null as indicator to use a default value. For this example we can implement a generic **nullable** type where it may or may not be assigned a value at construction.

Implementing a generic nullable type: MyTypes.h

```
template<typename T>
class nullable {
private:
    T m_value;
    bool m_hasValue;
public:
    nullable() { m_hasValue = false; }
    nullable(void *ptr) {
        if (ptr) throw std::exception("not null"); m_hasValue = false; }
    nullable(T value) { m_value = value; m_hasValue = true; }
    bool hasValue() const { return m_hasValue; }
    operator T() const {
        if (!m_hasValue) throw std::exception("null"); return m_value; }
};
```

Function that uses a nullable int

```
void show(my::nullable<int> v = 123) {
    if (v.hasValue()) cout << v << endl;
}

void main() {
    show();         // use default value
    show(nullptr);  // explicit null
    show(999);      // explicit value
}
```

1.5 Asynchronous Objects

Even though thread support is provided by the C++ standard library, threading is still function based where threads call functions instead of running objects. You can create your own class to implement the concept of asynchronous objects. This will simplify multi-threading even for those that are not familiar with the concept. If every object has its own thread, then it can run independently from other objects. Following is an example of a base class that will create one thread for each object. A thread cannot call object functions so we provide a static **callback** function that the thread will call. This function will then call the **run** function in the object that owns the thread. Since different objects will perform different tasks, the **run** function is abstract. This class is generic to allow you to customize what is passed to the **run** function as each task will require different input and output. For example a task that processes a file would then require the filename.

A base asynchronous object class: MyTypes.h

```
#include <thread>

template<typename T>
class async_object {
protected:
    std::thread m_thread;
private:
    static void callback(async_object *obj, T state) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        obj->run(state);
    }
public:
    virtual void run(T state) = 0;
    async_object(T state) : m_thread(callback, this, state) { }
    void wait() { m_thread.join(); }
};
```

You can then extend the base class to implement custom asynchronous tasks. Specify what is passed to the **constructor** and **run** function and then override **run** function to write the code for the task.

Extending the base class

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

class async_msgbox : public my::async_object<LPCTSTR> {
public:
    async_msgbox(LPCTSTR message) : my::async_object(message) { }
    void run(LPCTSTR message) override {
        MessageBox(NULL, message, _T("Message"), MB_OK); }
};
```

For example the above class will display a message box and the **run** function will then be passed the message to display. If you only have one thread you can only display one message at one time. Now you can easily display multiple messages.

Displaying multiple messages at the same time

```
void main() {
    async_msgbox b1(_T("Hello!"));
    async_msgbox b2(_T("Goodbye!"));
    async_msgbox b3(_T("Sayonara!"));
    b1.wait();
    b2.wait();
    b3.wait();
}
```

C++ is a very flexible language so you can any code to do anything you want. What is the problem is that different programmers can choose to write code in different ways making it confusing for those that are new to C++. Even though standardization will help provide a standard coding style, C++ is not fully standardized yet and different compilers support different the standards. C++ also does not come with a complete class library, most of the time you have to depend on third party libraries like **boost**. Creating a GUI application is also a problem since there is no standard GUI for C++. You have to choose the GUI system that you wish to use and then locate a third party library for it. Java has its own Swing UI system and C# has many UI systems that you can choose including Windows Form, WPF, Silverlight and Windows UWP. So to write a substantial C++ application would be a daunting task using C++ alone. You can also just use C++ to write critical low level code and compiled into a library or component that can be used from high level languages like Java or C#.

2 Static Library

2.1 Creating a Static Library

The C++ compiler generates an object code file from each source file that it compiles. Functions in object file can be then used in other projects without having to provide the source code. You can also create a static library project to combine multiple object files (**.obj**) into a single library file (**.lib**) so simplify the distribution of compiled code. In Visual C++, you can create a **Win32 Static Library** project to build a library.

Static library project

```
Project Type: Visual C++ Win32
Project Template: Win32 Project | Static Library
Project Name: Lib1 (will generate Lib1.LIB)
```

You need to create a header file containing declarations of all elements that you wish to export inclusive of variables, functions and classes. Variables must be declared with the **extern** keyword since is the only way to specify that it is a declaration and not a physical memory allocation. Allocation is specified in the source code.

Functions can be declared with or without **extern** keyword since a compiler will know that it is a declaration if it does not have a function body (**{ }**). Note that the **extern** keyword is not export command, it just tells the compiler that a variable or function exists somewhere either in source or compiled form. Classes do not require **extern** keyword since a class is considered a declaration. You may still choose to implement the class inside the header as inline functions or in a separate source file.

Declaration of elements (Lib1.h)

```
#pragma once
extern int var1; // declare global variables
extern int var2;
extern void foo1(); //  declare global functions
extern void foo2();

class Class1 { //   declare classes
public:
    void foo1();
};

class Class2 {
public:
    void foo2();
};
```

All elements in a static library are automatically exported. However, a program cannot access the element without declarations. So the overall process of creating a header is to allow a program to access all the elements exported by declaring the elements within the header file.

Exporting elements: Lib1.cpp

```
#include <stdio.h>
#include "Lib1.h"

//  exported elements
int var1 = 10;
int var2 = 20;
void foo1() { puts("foo1() called"); }
void foo2() { puts("foo2() called"); }
void Class1::foo1() { puts("Class1::foo1() called"); }
void Class2::foo2() { puts("Class2::foo2() called"); }
```

2.2 Testing a Static Library

Since libraries cannot run by themselves, you need to develop a program in order to test it out. To simplify the process, you can create a test program within the same solution. To make sure the library is always compiled before the prgram, you can set that the program is depended on a library using the **Dependencies** option from the **Project** menu.

Test project

```
Project Type: Visual C++ Win32
Project Template: Win32 Console Application
Project Name: TestLib1
```

Testing application (TestLib1\main.cpp)

```
#include "..\Lib1\Lib1.h"

void main() {
    var1 = 11; foo1();
    var2 = 22; foo2();
    Class1 obj1; obj1.foo1();
    Class2 obj2; obj2.foo2();
}
```

To specify the name of additional libraries to link to for a project, you can right-click over the project and select **Project Properties** option. In the **Linker** node you can also use the **General** page to add additional library directories. Use the **Input** page to enter a list of all the OBJ or LIB files to link to. If you wish to use the same libraries for **Debug** and **Release**, make sure that **All Configurations** configuration is active first before entering library names.

You may also wish to link to a different library for **Debug** and **Release** version. It will be common to attach a ***D*** character to the end of a debug library and an ***R*** to the end of a release library like ***Lib1D.lib*** and ***Lib1R.lib*** to differentiate them and be able to place both files in the same directory. A debug version of the program can link to the debug version of the library while the release version of the program can link to the release version of the library.

2.3 Distributing the Library

Once you tested the library you can distribute the library to others by giving them the header file and the debug and release library files. The files can be placed in standard C/C++ directories so that that compiler and the linker can locate them easily. The compiler can locate the file by searching through the directories. Standard directories for the Windows API can be found in *Microsoft SDKs* folder under the *Program Files* directory. Standard directories for C/C++ libraries can be located in *VC* folder under the *Microsoft Visual Studio X* directory. If you want to add additional directories, open the **Property Manager** from **View** menu. Then select any project and expand it and to see the available project configurations. Expand the right configuration you want to modify. Right click on **Microsoft.Cpp.*X*.User** and select **Properties**. You should see a **VC++ Directories** section for you to add additional directories. You can also assign your library as a standard library in the **Linker** |**Input** section.

Once your headers and libraries are now in standard directories, you can remove the dependency to the library project, change the way you include the header file and you only need to specify the library name if it is not a standard library. Then try to rebuilt the program to see if the header file can be located by the compiler and the libraries can be located by the linker.

Including header from a standard directory

```
#include <Lib1.h>
```

3 Dynamic-Link Library

3.1 Creating a DLL

Unlike static libraries where compiled code is in the LIB files, the compiled code for dynamic libraries is stored in dynamic link library (DLL) files instead. The LIB file only contains function names for linking and thus very much smaller than the LIB files for static libraries. The process of using static and dynamic libraries are the same except that the DLL file will be needed during runtime since all the compiled code of all the functions are not linked into the program. Only DLL and function names are recorded in the program. When program is loaded, the operating system automatically locates and ensure the DLL files are loaded. It will then map the address of the functions called from the libraries into the application and then execute it to run as a complete application.

There are many advantages to use dynamic libraries such as reducing disk and memory usage where you only need to keep one copy of the DLL on disk and only one copy of the DLL is ever loaded into memory regardless of how many applications are using it at the same time. Another advantage is that you can update the library without having to re-compile all the programs. Windows will first look in the current directory for a DLL file. If the file cannot be found, it will then search through the directories specified through a **PATH** environment variable. You have to make sure that the DLL file is stored into one of those directories or you can add the directory to the **PATH** variable.

There are two types of elements that you can export from a dynamic link library which are functions and classes. Global variables cannot be exported since variables are allocated in application's memory space and not in the library. Note that elements are not exported automatically for a DLL. To export a C++ element in a DLL, you need to use **__declspec** with the **dllexport** option. Then applications which want to use the functions from the DLL can declare them by using **__declspec** with the **dllimport** option.

Exporting a C++ functions and classes

```
_declspec(dllexport) void foo1();
class _declspec(dllexport) Class1 { … };
```

While it is not necessary to explicitly **dllimport** for declaring the external functions that you want to use from a dynamic link library, doing it will allow the compiler to generate more efficient code since it knows exactly the external functions comes from a dynamic link library.

You do not have to create two headers, one to export and one to import, you can use conditional compilation to export and import symbols from a single header file. Based on the definition of a macro, the compiler generate either exportation or importation declarations. The implementation code will define the macro to allow the compiler to export the symbols while the program that uses the library will not define the macro. In a DLL project, Visual C++ will add a **_USRDLL** and an **EXPORTS** macro that you detect and use in the header file as shown below.

Header to export and import symbols: Lib2.h

```
#pragma once
#ifndef LIB2_EXPORTS
     #define LIB2_API __declspec(dllimport)
#else
     #define LIB2_API __declspec(dllexport)
#endif
class LIB2_API Class1 {
    int var1;
public:
    void foo1();
};
LIB2_API void foo1();
LIB2_API void foo2();
```

Exporting implementing functions: Lib2.cpp

```
#include <stdio.h>
#include "Lib2.h"

void foo1() { puts("foo1() called"); }
void foo2() { puts("foo2() called"); }
void Class1::foo1() {   puts("Class1::foo1() called"); }
```

3.2 Testing a DLL

To make sure that the symbols are exported, you can use a Dependency Walker utility (DEPENDS.EXE) to examine the export table for the dynamic link library as shown in the following screenshot. This application can check DLL bindings of applications and libraries. You can view what are exported and imported by both applications and also libraries.

You can now create an application project to test the above library just like you to test the static library. You can then include the header and link to the library as you did for the static library. The only difference is that you must make sure that the DLL file is in the same directory on in one of the directories registered in the **PATH** environment variable.

Testing application (TestLib2\main.cpp)

```
#include "..\Lib2\Lib2.h"

void main() {
    foo1();
    foo2();
    Class1 obj1; obj1.foo1();
    Class2 obj2; obj2.foo2();
}
```

3.3 Using Namespaces

When implementing C++ libraries, you may consider declaring all the symbols within a namespace. This allows you to avoid naming conflict when using multiple libraries that have symbols with the same name. When you have naming conflicts between elements, you can access the elements through their namespaces. For those elements that have no conflicts, you can import them into the global namespace so that you do not have to specify the namespace to access them. If there are no naming conflicts for the entire library, you can import the entire namespace.

Syntax to importing elements in namespaces

```
using namespace_name::class_name;
using namespace_name::function_name;
using namespace namespace_name;
```

Declaring elements in namespace: Lib2.h

```
namespace lib2 {
      class LIB2_API Class1 {
         int var1;
      public:
         void foo1();
      };
      LIB2_API void foo1();
      LIB2_API void foo2();
}   // namespace lib2
```

Implementing elements in a namespace: Lib2.cpp

```
void lib2::foo1() { puts("foo1() called"); }
void lib2::foo2() { puts("foo2() called"); }
void lib2::Class1::foo1() { puts("Class1::foo1() called"); }
void lib2::Class2::foo2() { puts("Class2::foo2() called"); }
```

Simpler namespace element implementation: Lib2.cpp

```
namespace lib2 {
	void foo1() { puts("foo1() called"); }
	void foo2() { puts("foo2() called"); }
	void Class1::foo1() {  puts("Class1::foo1() called"); }
	void Class2::foo2() {  puts("Class2::foo2() called"); }
}
```

Accessing elements through namespace: main.cpp

```
void main() {
    lib2::foo1();
    lib2::foo2();
    lib2::Class1 obj1; obj1.foo1();
    lib2::Class2 obj2; obj2.foo2();
}
```

Importing specific elements into global namespace: main.cpp

```
using lib2::foo1;
using lib2::Class1;

void main() {
    foo1();
    lib2::foo2();
    Class1 obj1; obj1.foo1();
    lib2::Class2 obj2; obj2.foo2();
}
```

Importing entire namespace: main.cpp

```
using namespace lib2;

void main() {
    foo1();
    foo2();
    Class1 obj1; obj1.foo1();
    Class2 obj2; obj2.foo2();
}
```

3.4 Runtime Binding

When you link to a DLL, it will be automatically loaded. However, you can also load a DLL dynamically without linking to it. This makes your program more flexible since it can decide whether libraries are loaded and which ones to use during runtime. This is the technique used to load different device drivers. Use the **LoadLibrary** function to load in a DLL. Once the library is no longer required, use **FreeLibrary** to unload it.

Loading a DLL dynamically: RuntimeBinding1\main.cpp

```
#include <windows.h>
#include <tchar.h>
void main() {
    HMODULE hModule = LoadLibrary(TEXT("KERNEL32.dll"));
    if(hModule == NULL) {
        MessageBox(NULL, TEXT("Cannot load DLL!"), TEXT("Error"), MB_OK);
        return;
    }
    FreeLibrary(hModule);
}
```

When a DLL is loaded dynamically, you can retrieve the address of the function in the DLL by calling a **GetProcAddress** function. You need to declare a pointer type for the function to call. You can then declare a variable that will be used to store the address of the function retrieved from the DLL.

Declaring a pointer type to GetComputerName function

```
typedef  BOOL (WINAPI *GetComputerNamePtr)(
  LPTSTR lpBuffer,  // computer name
  LPDWORD lpnSize   // size of name buffer
);
```

Determining name of function to retrieve

```
#ifdef UNICODE
    LPCSTR functionName = "GetComputerNameW";
#else
    LPCSTR functionName = "GetComputerNameA";
#endif
```

Retrieving address of exported function

```
GetComputerNamePtr *pFunction = (GetComputerNamePtr *)
GetProcAddress(hModule, functionName);
if(pFunction == NULL) {
        MessageBox(NULL, TEXT("Function not found!"), NULL, MB_OK);
        FreeLibrary(hModule); return;
}
```

You can call the function repeatedly using the address. When the library is no longer required, you call **FreeLibrary** function to detach it from the current process. The address is no longer valid once that it has been detached.

Calling function through address

```
TCHAR name[256]; DWORD size=sizeof(name) / sizeof(TCHAR);
(*pFunction)(name,&size); FreeLibrary(hModule);
MessageBox(NULL, name, TEXT("Information"), MB_OK);
```

4 Process Control

4.1 Command Line & Exit Code

A process is virtual memory space where an application is loaded into when launched. Launching multiple applications will create multiple processes. It does not matter if you are launching the same application or a different application. You can try this by launching Notepad multiple times and use Windows Task Manager to check number of processes created. Each process can receive a list of command line arguments and return an exit code in its **main** function. All command line arguments are strings but you can use **atoi** and **atof** from **stdlib.h** to convert to **int** and **double**.

Following is an example of the full **main** function. The **argc** parameter indicates how many arguments on the command line and the **argv** parameter contains a string array of all the arguments on the command line including the application name. When a process completes successfully, we usually return 0. A non-zero or a negative value can be used to indicate the process has failed to complete successfully. You can also return an exit code by terminating the process. You can terminate a process from any function by using the standard C-library **exit** function.

Processing command line arguments

```
int main(int argc, char *argv[]) {
    printf("Number of command line arguments=%ld\n", argc);
    for (int i = 0; i < argc; i++)
        printf("Argument %ld = %s\n",
            i, argv[i]);
    return 0;
}
```

Functions to terminate current process and return exit code

```
if (argc < 2) exit(1);
```

Notice that we had to use **printf** function to display command line arguments as they are passed to the **main** function as an ANSI strings. To accept Unicode strings declare a **wmain** function instead. To be able to compile the program to use either ANSI or Unicode, declare **_tmain** function instead.

Accepting Unicode command line arguments

```
int wmain(int argc, wchar_t *argv[]) {
        :
}
```

Accepting either ANSI or Unicode command line arguments

```
int _tmain(int argc, TCHAR *argv[]) {
    if (argc < 2) exit(1);
    wprintf(_T("Number of command line arguments=%ld\n"), argc);
    for (int i = 0; i < argc; i++)
        wprintf(_T("Argument %ld=%s\n"),
            i,argv[i]);
    return 0;
}
```

You can launch applications from a console window or a batch program and use the command line to pass arguments to the application. The following example is a batch program that runs an application with command line arguments and checks the exit code.

Example batch program: batch1.bat

```
@ECHO OFF
process1 "Phillip Madden" tango@symbolicon.com.my
IF ERRORLEVEL 1 GOTO ERROR
ECHO Application process1 executed successfully.
PAUSE
GOTO END
ERROR
ECHO Error occurred running process1.
:END
PAUSE
```

4.2 Environment Variables

A process can read environment variables by calling a C-library **getenv** function or use the **GetEnvironmentVariable** Windows API function. Allocate memory to store the result before calling the function. If the memory is not large enough to store the result, an empty string will be stored instead.

Accessing environment variables

```
TCHAR env1[256], env2[2048];
GetEnvironmentVariable(_T("USERNAME"), env1, 256);
GetEnvironmentVariable(_T("PATH"), env2, 2048);
MessageBox(NULL, env1, _T("USERNAME"), MB_OK);
MessageBox(NULL, env2, _T("PATH"), MB_OK);
```

4.3 Creating Processes

The C-library provides a set of **exec** and **spawn** functions to create a process. The **exec** functions simply launches an application in a new process replacing the current process. The **spawn** functions does not end the current process and will also allow you to decide whether you want to wait for the child processes to complete before returning. Both function sets will allow you to pass in command line arguments. There is also a **system** function that can be used to run console commands, batch files and launch processes by just using a command line. The **system** function always wait for the process to complete.

Run and transfer control to new process: Process3\main.cpp

```
_execl("c:\\windows\\notepad.exe",
    "notepad", "c:\\cppdev\\instructor.txt");
```

Start new process and wait for process to complete

```
auto exitCode = _spawnl(P_WAIT, "c:\\windows\\notepad.exe",
    "notepad", "c:\\cppdev\\instructor.txt");
```

Use cmd.exe to process command line

```
system("notepad c:\\cppdev\\instructor.txt");
```

4.4 Precompiled Headers

Sometimes building an application using Windows API and a large number of libraries can be slow because the need to reprocess a number of large header files. Visual C++ allows you to pre-compile the header files into a PCH file that can then be reused when the individual CPP files are compiled. This will be done when you do not create a C++ empty project in Visual Studio. However you can still set this up quite easily. First of all add a single header file that includes all the other header files that you use. Then create a source code file that includes the single header file. Visual Studio C++ projects commonly use **stdafx** as the name for the header file and source file.

Header file: Process1\stdafx.h

```
#pragma once
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <iostream>
```

Source file: Process1\stdafx.cpp

```
#include "stdafx.h"
```

Right-click on the stdafx source code file and select to view the properties of the file. There will be an option that you can select the use this file to generate the PCH pre-compiled header file. To use the pre-compiled header file, right-click on other source files that includes the header file and select the option to reuse the pre-compiled header file rather than generating one.

Using the pre-compiled header: Process1\main.cpp

```
#include "stdafx.h"

int _tmain(int argc, TCHAR *argv[]) {
        :
}
```

When you create a new Win32 project and you do not specify an empty project, Visual C++ will create and setup your project to use pre-compiled headers and also add in the **_tmain** function for you.