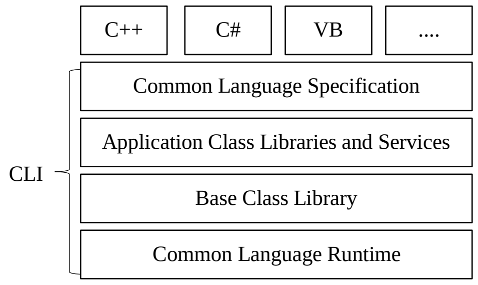
Visual C++

Lecture #2 C++/CLI Programming Basics

Introduction

Microsoft Visual C++ is an integrated development environment (IDE) from Microsoft for developing applications using C, C++, and C++/CLI programming languages. C++/CLI is the C++ language modified for **C**ommon **L**anguage **I**nfrastructure. It is a variation of C++ language designed and implemented by Microsoft for the .Net platform and is the primary language Visual C++ for developing GUI applications to be run in Windows operating systems. The following figure illustrates the relationship between C++ and CLI.


C++
C#
VB
....
Common Language Specification
Application Class Libraries and Services
Base Class Library
Common Language Runtime
CLI

Starting from this lecture, instructional contents will gear towards the discussion of using C++/CLI to develop GUI-based applications, known as “Windows Forms” application.

Basic Anatomy of a generic C++/CLI program

Unlike a standard C++ code, a generic C++/CLI program that displays “Hello World!” in the console screen must the use the “**using**” directive to import the “**System.dll**” file, which is a dynamic link library (DLL) file that provides instructions for the C++/CLI program to call upon to work with the Windows operating system.

C++/CLI codes also use the “using” directive to specify the use of “**System**” namespace, which contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions. The following compares the C++/CLI version with the standard C++ version of code. They produce exactly the same output.

C++/CLI	Standard C++

`#using <System.dll>`
 	
`using namespace System;`
 	
`int main()`
`{`
` Console::Write("Hello World!");`
 	
` return 0;`
`}`
	`#include <iostream>`
	
	`using namespace std;`
	
	`int main()`
	`{`
	` cout << "Hello World!";`
	
	` return 0;`
	`}`

Similar to Standard C++ codes, all executable C++/CLI programs must have a “main()” function which is the starting point of the program. The “main()” function in C++/CLI is typically declared as *int* (short for “integer) type. Yet, it can be declared as “void” type as shown below.

int	void

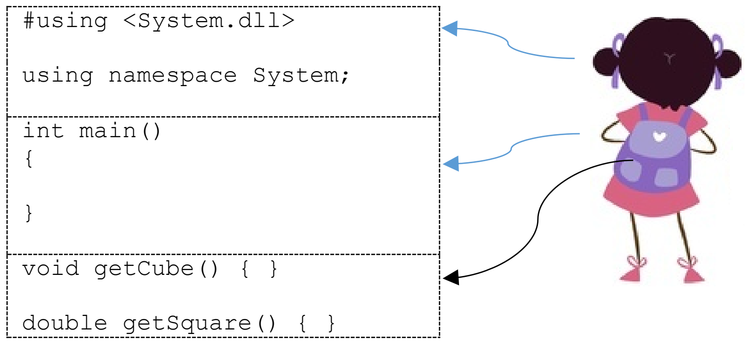
`#using <System.dll>`
 	
`using namespace System;`
	`#using <System.dll>`
	
	`using namespace System;`

`int main()`
`{`
` Console::Write("Hello World!");`
 	
` return 0;`
`}`
	`void`` main()`
	`{`
	` Console::Write("Hello World!");`
	`}`

Throughout this course, students can treat the following codes as the template because all C++/CLI must contain them. The "template" code: (1) imports the "Sysem.dll" library; (2) specifies the "System" namespace; and (3) creates the "main()" function and declare it as the *int* type.

```
#using <System.dll>                //Import the "System.dll" library

using namespace System;            //Specify the "System" namespace

int main()                         //Create the "main()" function
{

}
```

The above "template" code also illustrates the basic anatomy of a C++/CLI program. The "importing" segment can be considered the "head area", the "main()" function is considered the "body area", and all the "user-defined functions" are considered items in a backpack.



```
#using <System.dll>

using namespace System;
```

```
int main()
{

}
```

```
void getCube() { }

double getSquare() { }
```

For those who do not have previous programming background in standard C++ (which is not required for this course by the way), the following provides a basic explanation of the standard C++ version of the code.

With namespace	Without namespace

`#include <iostream>`
 	
`using namespace std;`
 	
`int main()`
`{`
` cout << "Hello World!";`
 	
` return 0;`
`}`
	`#include <iostream>`
	
	`int main()`
	`{`
	` `**`Std`**`::cout << "Hello World!";`
	
	` return 0;`
	`}`

The first line **`#include <iostream>`** imports the standard input/output library, as specified in the "iostream" header file, to the source code. In C++, namespaces group a set of global classes, objects and/or functions under one single identifier (name). By specifying **`using namespace std`**, programmers can omit the use of **`std::`** inside the code.

In C++, the scope resolution (::) operator is used to specify the origin of an object. For example, **cout** (standard output) is used in conjunction with the insertion operator (<<) to insert string literal into the stream, and then display the string on the screen. In the Standard C++, **cout** is defined in the "**std**" namespace; therefore, it is denoted as **std::cout**. The "std" namespace contains all the classes, objects and functions of the standard C++ library.

Declaring the following statement eliminates the need to use **std::cout** format. With the declaration, the compiler will assume that "cout" is from "std::cout" and will go to the "std" namespace to obtain the "cout" function.

```
using namespace std;
```

The ***cout*** method is a "built-in method" (meaning it comes with the Standard C++ language) to display a variety of things, such as strings, numbers, and individual characters, on the screen. In the following example, the *cout* method sends a string to the screen. The two less than signs, <<, is known as **insertion operator** which indicates the direction of output flows. In a console, the output direction is from the *cout* method to the screen.

```
cout << "Hello World!";
```

The advantage of a console application is that it is likely to be very small, and easy to distribute. The disadvantage is that user input and application output will be by text only. This means that the GUI (Graphical User Interface) will be absent, making the user interface perhaps unfamiliar and difficult for average users to use.

Basic input, processing, and output

The input-process-output (IPO) model is a widely used approach in systems analysis and software engineering for describing the structure of an information processing program or other processing. All computer programs perform a combination of I, P, and O. In other words, some programs take inputs, some take inputs and process them, others take input, process them, and return outputs.

In C++/CLI, the "System" namespace contains the "**Console**" class which provides basic input and output tools for console applications. The **Write()** function, for example, displays the specified string on the console screen. The following table compare the basic input and output functions of C++/CLI and standard C++.

I/O	C++/CLI	Standard C++
Output	`Write()` and `WriteLine()`	`cout`
Input	`Read()` and `ReadLine()`	`cin`

The difference between Read() and ReadLine() is that Read() only takes one character from the keyboard while ReadLine() takes the entire line of input. In the following example, the question requires user to enter either "*y*" or "*n*"; therefore, the Read() function is functionally sufficient to take the input because the input is a single character. By the way, a later lecture will discuss the primitive data type (such as "char") of C++/CLI in detail.

```
#using <System.dll>

using namespace System;

int main()
{
 Console::Write("Are you a human? [y/n] ");
 char c = Console::Read();
 Console::Write("You said, " + Convert::ToChar(c));

 return 0;
}
```

The above code uses the "template" to build the program structure. Then, it uses the "Write()" function to display a question, uses the "Read()" function to take a character from the keyboard and temporarily store the input in a variable of *char* type named "*c*". Interestingly, the *char* data type in C++/CLI stores the character's ASCII (or Unicode) value, not the character. For example, the character 'A' will be stored in "*c*" as 65 because the ASCII value of 'A' is 65. Finally, the above code converts the input from ASCII (or Unicode) value back to character and displays it with a string on the console screen. This program is simple, but it performs:

- **I**nput: Take a character and store it as ASCII (or Unicode) value.
- **P**rocessing: Convert the ASCII (or Unicode) value back to character and combine the character with a string.
- **O**utput: Display the string.

The following demonstrates how to use the "ReadLine()" function to read a line of input. In programming, a "line of input" is treated as a combination of characters and is known as a "string". In the following code, the input will be temporarily stored in variable of *String* type named "*fn*". A later lecture will discuss primitive data type of C++/CLI in detail.

```
#using <System.dll>

using namespace System;

int main()
{
 Console::Write("What is your full name? ");
 String^ fn = Console::ReadLine();
 Console::Write("Welcome, " + fn);

 return 0;
}
```

The difference between Write() and WriteLine() functions is that the WriteLine() function writes the string followed by a new line, while Write only writes the string without a new line, as illustrated below.

Function	Write()	WriteLine()

Code	`#using <System.dll>`
	
	`using namespace System;`
	
	`int main()`
	`{`
	` Console::Write("Monday");`
	` Console::Write("Tuesday");`
	` Console::Write("Wednesday");`
	` return 0;`
	`}`
		`#using <System.dll>`
		
		`using namespace System;`
		
		`int main()`
		`{`
		` Console::WriteLine("Monday");`
		` Console::WriteLine("Tuesday");`
		` Console::WriteLine("Wednesday");`
		` return 0;`
		`}`

Output	`MondayTuesdayWednesday`	`Monday`
		`Tuesday`
		`Wednesday`

The above two sample codes all contain an interesting statement, as shown below, because in a standard C++ programs that defines its "main()" function as *int* type, it is linguistically correct to return an integer value to its calling party. The returned value of the main function is considered the "Exit Status" of the application. Most operating systems treats the returned 0 as an indicator of "success status" and interpret it as "the program worked fine".

```
return 0;
```

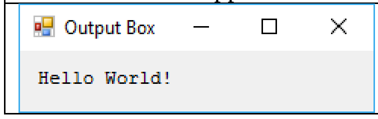
In C++/CLI, it is optional to add “return 0;” at the end of the “main()” function because the compiler will include it automatically if the programmer chooses not to add it. The following two versions work in Visual C++.

With	Without
`#using <System.dll>`	`#using <System.dll>`
`using namespace System;`	`using namespace System;`
`int main()`	`int main()`
`{`	`{`
` Console::Write("Monday");`	` Console::Write("Monday");`
	`}`
` **return 0;**`	
`}`	

In most of later lectures, the instructors will tend to ignore the “return 0;” statement.

From console applications to Windows Forms applications

The term “**Windows Forms**” (or WinForms) is a set of graphical (GUI) libraries provided by the Microsoft .NET Framework as application programming interfaces (APIs) for building rich client applications to run in Windows desktop environments. Any application created using this platform is known as “Windows Forms applications” (or WFPs). In other words, a **Windows Forms application** is an application that has a graphical user interface (GUI), such as Button and Label controls, designed to be run in the desktop of a Windows machine. A Windows form application can run on the Windows desktop as an individual, self-executable, GUI application.

Windows Forms application	Console application
 Output Box — Hello World!	`C:\>lab2_0.exe` `Hello World!`

As shown in the above figure, a console application displays its output in the console environment which is a text-based environment. A Windows Forms application typically creates a rectangular area with a scheme of colors and a set of icons, known as graphical user interfaces (GUIs), for taking input(s) and displaying output(s).

The .Net Framework provides many GUI-based tools, such as the “Show()” function of the “MessageBox” class, for programmers to use. One interesting analogy is to treat the “MessageBox::Show()” function as the GUI version of “Write()” (or “WriteLine()”) function. The following is a sample C++/CLI code that displays a string “Welcome!” in a GUI “form” known as “message box”. Figure 1 is a sample “message box”.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
 MessageBox::Show("Welcome!");

 return 0;
}
```

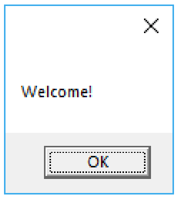

Welcome!

Figure 1

The above “Windows Form” example produces a small pop-up message box created by the **Show()** method of the **MessageBox** class. The MessageBox class defines a message box that can contain text, buttons, and symbols to inform and instruct the user. For Windows 10 and 8.1, a message box is a rectangular graphical user interface containing text, a title, one or more clickable button(s), and a few icons.

The following are two "console" versions: C++/CLI and standard C++ codes. They will only display a text on the command-line interface (such as the Command Prompt). They do not display output with any graphical interfaces.

C++/CLI	Standard C++
`#using <System.dll>`	`#include <iostream>`
`using namespace System;`	`using namespace std;`
`int main()`	`int main()`
`{`	`{`
` Console::Write("Welcome!");`	` cout << "Welcome!";`
` return 0;`	` return 0;`
`}`	`}`

Since one of the objectives of Visual Studio is a develop applications to run in the GUI mode of Windows operating systems, it is probably practical for students to learn Visual C++ programming using GUI-based C++/CLI codes. It is necessary to note that, in this course, students will still learn all the basic programming skills, although the header files and namespaces as well as directive and preprocessors are a little more complicated than those of Standard C++. The following illustrates how GUI and console application are similar in anatomy and architecture.

GUI (WFP)	C++/CLI

`#using <System.dll>`
`#using <System.Windows.Forms.dll>`
	`#using <System.dll>`

`using namespace System;`
`using namespace System::Windows::Forms;`
	`using namespace System;`

`int main()`
`{`
	`int main()`
	`{`

MessageBox::Show`("Welcome!");`	**Console::Write**`("Welcome!");`

`return 0;`
`}`
	`return 0;`
	`}`

From this section on, the lecture will focus on guiding students to explore how to hand-code GUI-based applications. By the way, for the sake of learning, it is necessary to revisit some topics that have been discussed in Lecture #1 with a more detailed discussion.

Basics of GUI-based programming in .Net Platform

With the Visual C++ IDE, developers can build Windows Forms applications by adding controls to the form and then adding code to respond to user actions (such as mouse clicks or key presses). A "control" is an interactive user interface (UI) such as a Button, a TextBox, a Label that displays data, accepts data input, and so on.

The .Net Framework provides many "dynamic-link library" files for programmers to use. A dynamic link library (DLL) is a module that contains functions and data to be used by another module (application or DLL). The entire "System" namespace, for example, is packed into a single .dll file named "System.dll" which contains fundamental classes and base classes to define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions. DLL files can be directly "imported" to any C++/CLI codes.

To the minimum, source code of a "Windows Forms application" must include the following DLL (Dynamic Linking Libraries) files, whereas "**#using**" is a preprocessor directive that imports the code provided by the .dll file. These two .dll files can be considered the most essential libraries for all .NET programs.

```
#using <System.dll>
#using <System.Windows.Forms.dll>
```

"Using" a DLL file in Visual C++ is like "importing" a **header file** in Standard C++, except that C++ uses the "#include" preprocessor directive to import header files. Libraries of Standard C++ provide tools (such as classes or functions) to be referenced by other C++ source files, and C++/CLI preserves the convention. Almost every GUI-based application created by Visual C++ need the references provided by some dynamic link libraries to obtain specialized tools provided by the .NET Framework to perform tasks in the Windows environment.

The "System.Drawing.dll" file provides tools defined in the "System::Drawing" namespace for basic graphics functionality, such as the Color, Size, and Point. It is a useful .dll file for building Windows Forms applications (WFPs).

```
#using <System.Drawing.dll>
```

Like the Standard C++, the .NET Framework organizes toolsets into "namespaces". A **namespace** is a declarative region that provides a scope or territory for code entities (data types, functions, variables, classes, structure, etc.) to be packed inside it. Namespaces are used to organize reusable codes into logical groups and to prevent name collisions that can occur especially when the programmers' code base includes multiple libraries.

An example of name collision is: There are 41 towns or cities named "Springfield", such as Springfield, Alabama, Springfield, Arkansas, Springfield, Colorado, Springfield, Florida, Springfield, Georgia, Springfield, Idaho, Springfield, Illinois, and so on. In the format of C++/CLI, Springfield of Illinois will be expressed as `Illinois::Springfield` in order to distinguish from `Colorado::Springfield`.

In computing, structure of naming convention that implies the hierarchy, scope, or territory is called a "qualified name". The following is another example that explains the concept of "qualified name". At least two cities are named "Vancouver": Vancouver, British Columbia, Canada and Vancouver, Washington, USA. From the country to a park of the city, the following is a "qualified name" that can help distinguish which Vancouver is the correct one that has a park named "Stanley Park".

```
Canada::British_Columbia::Vancouver::Stanley_Park
```

Throughout the course, students will always use the following namespaces to regulate how the .NET Framework tools are properly referenced.

```
using namespace System;
using namespace System::Windows::Forms;
```

The following table illustrates the features of the above two namespaces.

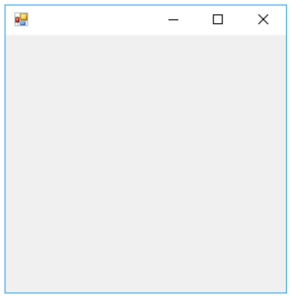
System	**System::Windows::Forms**
Contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.	Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.

By the way, the Drawing namespace is also a frequently referenced namespace.

```
using namespace System::Drawing;
```

The following is the bare minimum code needed to create a generic Windows "form". It simply creates one instance of the "System::Windows::Forms::Form" class named "form1" and uses the **Run()** method of the **Application** class to actually create the Windows "form".

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
 Form^ form1 = gcnew Form;

 Application::Run(form1);
}
```



From the developers' perspectives, there is an advantage for using both "System" and "System::Windows::Forms" namespaces--to omit the need of "qualified name". As illustrated previously, in computing, a **fully qualified name** is an unambiguous name containing the absolute hierarchy of the structure in which a class resides. The "System::Windows" namespace, for example, contains "Application", "Forms", and "DataFormat" classes as well as many others. If all the three classes contain a member "Length", the following fully qualified names can avoid the ambiguity. It clearly specifies which "Length" is supposed to be 17.

```
System::Windows::Application::Length = 15;
System::Windows::Forms::Length = 16;
System::Windows::DataFormat::Length = 17;
```

The following, if the namespaces are not previously specified by the "using" keyword, will inevitably confuse the compiler and could cause the compilation to produce incorrect results.

```
Length = 15;
Length = 16;
Length = 17;
```

Assuming the "`using namespace System::Windows::Forms;`" statement is absence, then programmers must use the fully qualified name, as shown below, to clearly specify to the compiler that the "Form" class of the "System::Windows::Forms" namespace is the correct one to use.

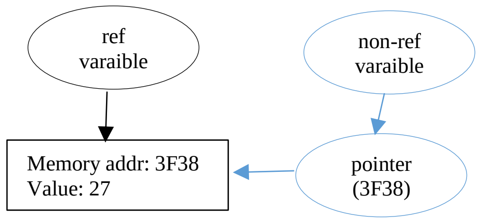
```
public ref class Form1: public System::Windows::Forms::Form { }

public ref class Form2: public System::Windows::Forms::Form { }

public ref class Form3: public System::Windows::Forms::Form { }
```

By using the "using" keyword to specify the "System::Windows::Form" namespace, the above code can be simplified to the following, which "Form" will be recognized by the program as "System::Windows::Forms::Form" because the hierarchical structure has been declared by the "using" keyword.

```
using namespace System::Windows::Forms;
.........
```

```
public ref class Form1: public Form { }

public ref class Form2: public Form { }

public ref class Form3: public Form { }
```

On the other hand, Visual C++ codes are managed codes; therefore, the "Form1", "Form2", and "Form3" classes must be declared as "**ref**" type. In programming language theory, a "reference type" is a data type whose value indicates an area of physical memory (typically the DRAMs), not a scalar value like 23, 57.21, "Apple", or true. In other word, objects of reference type provide a memory address as reference to tell the computer where in the physical memory to find the demanded data, while objects of non-reference type (typically known as value types) directly store the data.



Assuming a piece of data 27 is stored in memory at the memory address 3F38. A reference variable stores "3F38" (not 27); therefore, the CPU can use the reference variable as "reference" to obtain that memory address and then get the value 27. A non-reference variable stores the value 27 without know where the value is stored in physical memory. A non-reference variable needs to work with another variable, known as "pointer", to obtain the memory address.

In Visual C++, classes of reference type must be marked with the "ref" keyword. Only a "ref" class or "ref" struct can inherit from members provided by managed classes and structs. A managed class and a managed struct are object-oriented components written in managed codes. They come with language-specific mechanisms for memory management and garbage collection, and that simply means Visual C++ developers do not need to write custom-made codes to release physical memory after executing the core content of the program.

In C++/CLI, instances of reference types must be declared with the "handle to object" operator ("^", a punctuator pronounce as "hat"), as shown below. In Standard C++, asterisk "*" denote a native pointer. In C++/CLI, the "^" symbol denote a **handle** which can be considered a managed pointer. The handle provides the reference to the memory address where the "form1" object resides. Such references are designed to avoid memory leaks if resources are not properly freed or delete.

```
Form^ form1 = gcnew Form;
```

The following is another example. A handle will provide reference to the memory address where "label1" resides. The statement creates an instance of the "System::Windows::Forms::Label" class named "label1" with a handle denoted by "^".

```
Label^ label1 = gcnew Label;
```

In addition to the two essential namespaces, if necessary, Visual C++ developers can use other namespaces and their DLL files depending on tools the developers choose to use. For example, the instructor needs to use the **System::Drawing** namespace if the instructor has to specify the "Location" property, which is typically used to specify where to place a GUI component. The following is an example which uses the "Location" property to place a TextBox control at a point (48, 488) of a Windows "form". The developer has two options: (1) use the fully qualified name; or (2) declare the namespace first to omit the need of fully qualified names.

```
#using <System.Drawing.dll>
......
......
textBox1->System::Drawing::Location = System::Drawing::Point(48,
488);
```

or

```
#using <System.Drawing.dll>
......
......
Using namespace System::Drawing;
......
......
textBox1->Location = Point(48, 488);
```

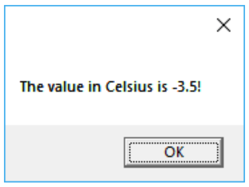
Inside the "main()" function, the "**Run()**" method of the "**Application**" class will coordinate with the .Net Framework (installed in the operating system) to build the "form1" object as specified by the "Form^ form1 = gcnew Form" statement.

```
int main()
{
 Form^ form1 = gcnew Form;

 Application::Run(form1);
}
```

Although the sample code presented in this section can illustrate the basic structure of a Windows Forms application, the following section will discuss a more secured structure recommended by Visual C++.

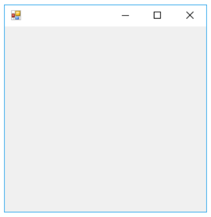
From message box to "secured" Windows "form"

A message box created by the MessageBox::Show() method is the simplest Windows "form" application. In a nutshell, a "form" is a rectangular area with UI (user interface) components. The following figure is a generic message box. It contains two clickable components (buttons) and on static component (a Label that display the text).


The value in Celsius is -3.5!

While MessageBox::Show() is a tool provided by the .Net Framework, developers can manually write simple Visual C++ codes to create such a "form" without using Visual Studio IDE. Typically, developers need to declare a public class with a unique ID (e.g. **Form1**) to inherit the "System::Windows::Forms::Form" class. The "Form1()" constructor inside the "Form1" class is the one that actually create the "form". There must be a mechanism to "instantiate" an instance of the "Form1" class. The following Visual C++ code is the bare-minimum code to create a Windows "form". The orange lines indicate the flow of execution. It starts from "Application::Run()" (instantiation), continue with the "Form1" class, and then the "Form1()" constructor.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

public ref class Form1: public Form
{
 public: Form1() { } // constructor
};

[STAThread]
int main()
{
 Application::Run(gcnew Form1);
}
```



The above code starts with declaring a "reference" class named "Form1". The declaration adopts a *A*:*B* format (or "*A* is *B*") to specify that the "Form1" class is a "child class" (or a "derived class") of the "System::Windows::Forms::Form" class. This declaration allows "Form1" to immediately inherits all public members of the "System::Windows::Forms::Form" class.

```
public ref class Form1: public Form { }
```

The following figure illustrates how the inheritance works. With the above inheritance, the "Form1" class obtains all the public tools provided by the .Net Framework, particularly from the "System::Windows::Forms::Form" class.

A	B	A:B
A is empty	B has many components: Text, Size, Wifth, Height, etc.	A:B, so A now has all of B's public components: Text, Size, Wifth, Height, etc.

Visual C++ Windows Forms application must be created as a managed class. The "**ref**" keyword indicates that the **Form1** class is a managed class that can inherit tools from other managed classes, structs, and interfaces as well as managed data types.

```
public ref class Form1: public Form { }
```

Inside the "Form1" class, Visual C++ suggests programmers to define a "default constructor" which is a special type of function that has exactly the same identifier as the class ("Form1"). The word "default" implies that it will be carried out by default at the moment when the instance of the "Form1" class is created.

```
public ref class Form1: public Form
{
 public: Form1() { }
};
```

Inside the default constructor (the "Form1()" function), programmers can define properties of the "form", as shown below: **Text** and **Size**. These two properties describe the appearance of this form. The keyword "**this**" represents the current form (in case the application contains two or more "forms").

```
public ref class Form1: public Form
{
 public: Form1()
 {
  this->Text = "My Form";
  this->Size = Drawing::Size(250, 150);
 }
};
```

Reference (ref) types of components (methods and properties) can only be instantiated as the managed type; therefore, properties are referenced by using the "arrow operator" (->), not the dot operator (.) which is used in many programming languages. Methods are references by using the "scope resolution operator" (::), not the dot operator (.).

```
this->Size = Drawing::Size(250, 150);
```

The following illustrates the relationship between the arrow operator and its operands. Basically, "this->Size" means "the current form has Size property".

Keyword and Symbol	`this`	`->`	`Size`
Meaning	the current form	has	a property

Member of a managed class is specified by the scope resolution operator (::). The scope resolution operator (::) is used to specify the origin of an object. It is also used to identify and disambiguate identifiers used in different scopes.

```
this->Size = Drawing::Size(250, 150);
```

The following illustrates the relationship between the scope resolution operator and its operands. Basically, "Drawing::Size()" means "the Size() method of the System::Drawing namespace".

Keyword and Symbol	`Drawing`	`::`	`Size()`
Meaning	System::Drawing	of	a method

In terms of object-oriented programming, a "**property**" is a member of a "class" that provides a flexible mechanism to read, write, or compute the value of a private field. A "field" is a variable of the "class". A "class" is a logical container (no physical shape) that is created by programmers to define an "object" (such as a user). A "**method**" is a code block that contains a series of statements to perform a task. In the above example, the "Size" property keeps the value assigned by the "Size()" method. In programming, a single equal to sign (=) is known as the "assignment operator" and is used to assign a value to a "data holder".

`this->Size`	`=`	`Drawing::Size(250, 150)`
property	operator	method

Insider the "Form1()" constructor, properties can be assigned values using the following format.

```
ObjectID -> PropertyName = value;
```

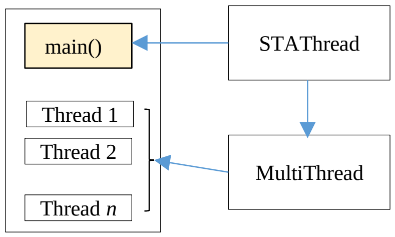
In the following statement, "My Form" is a value of *string* type assigned to the "Text" property of the "form".

```
this->Text = "My Form";
```

As standard C++ or console C++/CLI programs, GUI-based based C++/CLI program must have a starting point (the main() function) in order to be self-executable. Visual C++ suggests programmers to place the "main()" function under the [STAThread] section.

```
[STAThread]
int main()
{
 Application::Run(gcnew Form1);
}
```

The **STAThread** attribute marks the STA (single-thread apartment) mode, which means to treat the main() function as a single thread; therefore, the main() function will be the only thread of the program to be executed by CPU (or processor) during the program initialization phase. If there is any attempt to access other part of the code, the "STAThread" attribute will force the operating system to give the sole priority to the "main()" function.



Inside the "main()" function, the "**Run()**" method of the "**Application**" class will coordinate with the .Net Framework (installed in the operating system) to launch a "virtual machine" as host to build the "form" that is specified by the "Form1()" constructor. Since a Visual C++ Windows Forms application is declared as a managed type class, programmer must use the "**gcnew**" operator (not the regular "new" operator) to create an "anonymous" instance of the "Form1" class on the garbage collected heap. An "anonymous" instance is an instance without identifier.

```
Application::Run(gcnew Form1);
```

By the way, the above statement is functionally equivalent to the following.

```
Application::Run(gcnew Form1());
```

Interestingly, the instance should never be called any party, so it does not need an identifier. Do not use the following statements. They could bring in some security issues.

```
Form1^ form1 = gcnew Form;

Application::Run(gcnew form1);
```

With any generic text editor, such as Notepad, students can assemble the above code segments into a complete piece, as shown below, to hand-code a GUI application. In a later lecture, students will learn to hand-code a Windows Forms application. The orange lines indicate the flow. While "gcnew Form1" creates an instance of the "Form1" class, the "Form1()" constructor of the "Form1" class creates the "form".

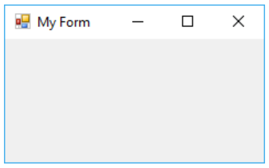
```
#using <System.dll>
#using <System.Windows.Forms.dll>
#using <System.Drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;
```

```
public ref class Form1: public Form
{        creation

 public: Form1()
 {

  this->Text = "My Form";
  this->Size = Drawing::Size(250, 150);

 }                                           instantiation
};

[STAThread]
int main()
{
 Application::Run(gcnew Form1);
}
```

The above sample code can be compiled using the following statement, assuming "test.cpp" is the file name of the above code.

```
cl /clr test.cpp /link /subsystem:windows /ENTRY:main
```

The following is a sample output of the above code.



Throughout this course, students can treat the following code as "template" code of a Windows Forms application. A later lecture will discuss about hand-coding Windows forms in details.

```
#using <System.dll>
#using <System.Windows.Forms.dll>
#using <System.Drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref class Form1: public Form
{
 public: Form1()
 {
  // form components
 }
};

[STAThread]
int main()
{
 Application::Run(gcnew Form1());
}
```

A later lecture will discuss about how to add components like ComboList, drop-down menu, and Progress Bar to a "Windows Forms" application. Throughout this course, students will frequently add Label, TextBox, and Button controls to the above "template code" to build simple GUI-based applications.

Adding basic components to a Windows "form"

The following is a Windows "form" created using the "template code" (discussed in a previous section) with the declaration and creation of a "Label" control (as component of the "form") that displays a text message "Welcome to Visual C++!". The Label control is defined by the "Label" class (precisely System::Windows::Forms::Label) of the .NET Framework. The "Label" class supports several properties such as **Text** which stores the caption of the Label control, and **AutoSize** that allows the form to automatically adjust the size of the Label control. The **Location** property gets or sets the coordinates of the upper-left corner of the control relative to the upper-left corner of its container.

```
#using <System.dll>
#using <System.Windows.Forms.dll>
#using <System.Drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref class Form1: public Form
{
 public: Form1()
 {

   Label^ label1 = gcnew Label;
   label1->Text = "Welcome to Visual C++!";
   label1->AutoSize = true;
   label1->Location = Point(10, 10);

   Controls->Add(label1);

 }
};

[STAThread]
int main() {
 Application::Run(gcnew Form1());
}
```

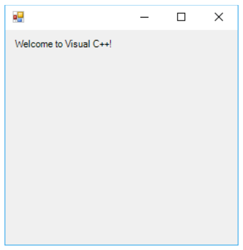


Figure 2

The following statement declares an instance of the "Label" class named "label1". The "Label" class is a managed class; therefore, its instance must be declared with the "handle to object" operator (^) which is pronounced as "hat". This operator is also known as "handle declarator" and is designed for the operating system to automatically delete the designated object when the system determines that the object is no longer valid or needed or accessible. In the following example, the designated object is "label1". Due to the garbage collection mechanism provided by the .Net Framework, all instances of a managed class must be created using the "gcnew" operator.

```
Label^ label1 = gcnew Label;
```

The above statement can also be written as:

```
Label^ label1 = gcnew Label();
```

Depending on the need of access control. The above statement may be broken down to two statements.

```
Label^ label1;   # declaration
..........
```

```
        label1 = gcnew Label;    # object creation
```

The following codes illustrates the concept of scope-based access control. The "class level" version of code declares the "label1" object as a member of the "class", the "Form1()" constructor does not have the ownership. The "Show()" method as another member of the "Form1" class can access the "label1" object.

Class level

```
public ref class Form1: public Form
{
 Label^ label1;

 public: Form1()
 {

  label1 = gcnew Label;
  label1->Text = "Welcome!";
  label1->AutoSize = true;
  label1->Location = Point(10, 10);

  Controls->Add(label1);

 }

 void Show()
 {
  label1->Text = "Hi!";
 }
};
```

Constructor level

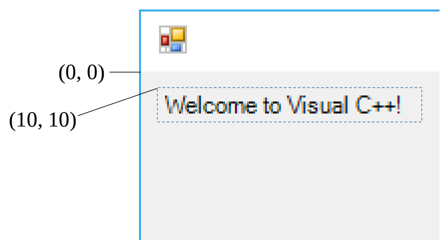
```
public ref class Form1: public Form
{
 public: Form1()
 {

  Label^ label1 = gcnew Label;
  label1->Text = "Welcome!";
  label1->AutoSize = true;
  label1->Location = Point(10, 10);

  Controls->Add(label1);

 }

 void Show()
 {
  label1->Text = "Hi!";
 }

};
```

The "constructor version" declares the "label1" object as a member of the "Form1()" constructor; therefore, the "Show()" method cannot access it.

The "**Add()**" method of the "Controls" property ties the Label control to the form. Without the following statement, the Label control is not set to be a component of the "form", and the "form" will not contain the Label control.

```
        Controls->Add(label1);
```

Figure 2 is a sample output of the above code. The following figure illustrates how the "Location" property specifies where in the form to place the Label control. The dashed line represents the invisible border of the Label control. Its upper-leftmost point is the point used by the "Location" property to designate its starting point. In a "form", there is typically a "title bar" above the body area. The coordinate of components of the "form" are determined based on the "body area" (excluding the "title bar").



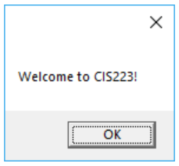
In a nutshell, adding a component like a Label control to a form starts with declaring an instance of the control, continues with defining appropriate properties of the control, and the uses the "Add()" method of the "Controls" property to set the control to a "form".

Custom-made tools created by

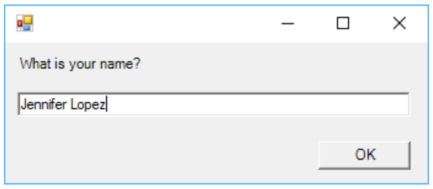
The .Net Framework provides many useful tools, but not "all" the tools. Programmers often need to manually develop the demanded tools. In the following sample code, the output is the instructor for students to use in this course

displayed in a message box because the .Net Framework provides the "Show()" method from its "MessageBox" class.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
 MessageBox::Show("Welcome to CIS223!");
}
```


Welcome to CIS223!

Interestingly, the .Net Framework does not provide Visual C++ with a generic "input box" (or "prompt box") to take inputs from the users. The following illustrates the concept of a dialog box for taking user input. It is basically a "form" with a title bar and a body area. The title bar contains a few icons, while the body area contains a Label control that displays the question and a TextBox control for the user to enter answer(s). There is a Button control for the user to click on to trigger the data processing.


What is your name?

Jennifer Lopez

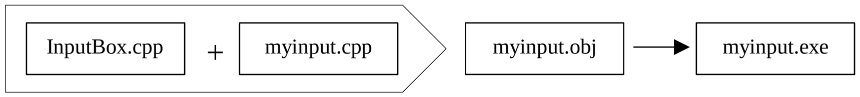
The instructor has to create the following C++/CLI codes and save it as an individual Visual C++ file (e.g. "**InputBox.cpp**"). This file will behave like a "library code" because it can be "included" (or imported) into other C++ source file through the use of ***include*** directive to create a Windows form that serves as an input box. This newly created input box can then take the inputs from keyboard and pass them to its calling party. Do not worry if you cannot understand the following code. A later lecture will discuss the details to hand-code Windows form applications.

```
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref struct InputBox
{
 private: Form^ form1;
 private: Label^ label1;
 private: TextBox^ textBox1;
 public: static String^ Text;

 private: Void button1_Click(Object^ sender, EventArgs^ e)
 {
  if (textBox1->Text == "")
  {
   MessageBox::Show("No value given.");
   Text = "";
  }
  else
  {
   Text = textBox1->Text;
   form1->Close();
  }
 }

 public: static String^ Show(String^ s)
 {
  Text = s;
  InputBox();
  return Text;
 }

 public: InputBox()
 {
  form1 = gcnew Form;

  label1 = gcnew Label;
  label1->AutoSize = true;
  label1->Text = Text;
  label1->Location = Point(10, 10);
  form1->Controls->Add(label1);

  textBox1 = gcnew TextBox;
  textBox1->Size = Size(320, 25);
  textBox1->Location = Point(10, label1->Height + 10 + 10);
  form1->Controls->Add(textBox1);

  form1->Width = textBox1->Width + 40;

  Button^ button1 = gcnew Button;
  button1->Location = Point(form1->Width - button1->Width - 30,
textBox1->Top + textBox1->Height + 10);
  button1->Text = "OK";
  button1->Click += gcnew EventHandler(this,
&InputBox::button1_Click);

  form1->Controls->Add(button1);

  form1->Height = label1->Height + textBox1->Height + button1-
>Height + 80;

  Application::Run(form1);
 }

};
```

With the above codes as a “header file”, students can begin writing the following codes (in another individual source file (such as “myinput.cpp”) to take inputs from keyboard and then display the results in a message box.

```
//File name: myinput.cpp
#include "InputBox.cpp" //import library

int main()
{
 String^ str = InputBox::Show("What is your name?");

 MessageBox::Show("Welcome, " + str + "!");
}
```

The **#include** directive instructs the preprocessor to copy and paste the source code in the "InputBox.cpp" file to the "myinput.cpp" file. The following figure illustrates how the two codes merged.



It is necessary to note that the "InputBox.cpp" file already contain the following statements; therefore, programmers do not have to add the following lines to the "myinput.cpp" file to avoid redundancy.

```
//Filename: InputBox.cpp
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;
```

The following explains the redundancy.

No Redundancy	Redundancy

`//File name: myinput.cpp`
`#include "InputBox.cpp"`
 	
`int main() { .......... }`
	`//File name: myinput.cpp`
	`#using <System.dll>`
	`#using <System.Windows.Forms.dll>`
	
	`using namespace System;`
	`using namespace System::Windows::Forms;`
	
	`#include "InputBox.cpp"`
	
	`int main() { .......... }`

The "**str**" variable is declared with a handle (^). A variable that is declared with the "handle to object" operator behaves like a pointer to the object. The following redirects the user input to the "str" variable whose location in the memory will be kept by the handle (^).

```
String^ str = InputBox::Show("What is your name?");
```

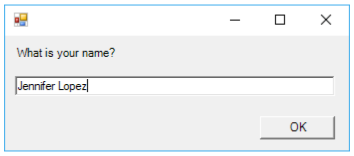
As shown below, the **InputBox::Show()** method use the "return" statement to return a string to its calling party. The returned string can be assigned to a *string* variable such as ***str*** in the above code without going through data type conversion. The orange lines illustrate the flow.

```
public: static String^ Show(String^ s)
{
 Text = s;
 InputBox();
 return Text;
}
```

To compile the program, use the following statement. Details about the compiler options are available in Lecture #1.

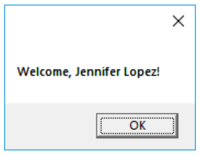
```
cl.exe /clr myinput.cpp /link /subsystem:windows /ENTRY:main
```

The following is a sample output.


What is your name?

Jennifer Lopez

and


Welcome, Jennifer Lopez!

InputBox::Show() MessageBox::Show()

The following is another example that uses the output of "InputBox::Show()" method provided by the "InputBox.cpp" file. The obtained user input will become content of the MessageBox::Show() method.

```
#include "InputBox.cpp" //import library

using namespace System;
using namespace System::Windows::Forms;

int main()
{
 MessageBox::Show(InputBox::Show("How old are you?"));
}
```

The following is another sample program that creates a Windows "form" for displaying outputs. It is designed for students to use as an alternative of the "MessageBox::Show()" method. Do not worry if you cannot understand the following code. A later lecture will discuss the details to hand-code Windows form applications.

```
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref class OutputBox: public Form
{
 Label^ label1;

 public:

  OutputBox(String^ s)
  {
   this->Text = "Output Box";

   label1 = gcnew Label();
   label1->AutoSize = true;
   label1->Location = Point(10, 10);
   label1->Text = s;
   label1->Font = (gcnew System::Drawing::Font(L"Courier New", 9));
   Controls->Add(label1);

   if (label1->Width < 200) { this->Width = 250; }
   this->Height = label1->Height + 60;
  }

  static void Show(String^ s)
  {
   Application::Run(gcnew OutputBox(s));
  }
};
```

The identifier of the class is "OutputBox". The "OutputBox" class contains one constructor named "OutputBox()" and a method named "Show()". Both the "OutputBox()" constructor and "Show()" method take one string literal as parameter (whose identifier is "*s*"). It is necessary to note that the instantiation of the instance of "OutputBox" class is done by the "Show()" method, because there is no "main()" function available. This program is designed as a supportive library, not a self-executable program; therefore, it does not need to have the "main()" function.

The following is a sample statement to create an instance of the "OutputBox" class from a different Visual C++ source file. The "include" directive cause the compiler to copy and paste the entire content of the OutputBox.cpp" file to the following code.

```
#include "OutputBox.cpp"

int main()
{
 OutputBox::Show("Hello World!");
}
```

In the above example, "Hello World!" is the string literal that will be assigned to the "*s*" parameter of the "Show()" method, and then transfer to the "*s*" parameter of the following statement.

```
Application::Run(gcnew OutputBox(s));
```

The "OutputBox(*s*)" part of the above statement will immediately create an instance of the "OutputBox" class and pass the value of "*s*" to the "OutputBox()" constructor to build a "form" with a Label control. The "Text" property of the Label control will display the value of "*s*" sent from the "OutputBox()" constructor.

Using tools provided by the .Net Framework

The .NET Frameworks provides many methods for the programmers to use without spending time and efforts to re-develop similar features. The methods provided by the .NET Framework is referred to as "**built-in methods**" in this course. The Show() and ToDouble() methods are examples of them.

The **Math** class provides constants and static methods for trigonometric, logarithmic, and other common mathematical functions. Details are available at **`http://msdn.microsoft.com/en-us/library/system.math.aspx`**. For example, the **`Math::Pow(`*`n, m`*`)`** method can raise a value *n* to the *m*th power, n^m. The following demonstrates how to use the **Pow(*n, m*)** method to return a specified number to a specified power, particularly it returns the result of 2 to the 5th power, 2^5.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

void main()
{
 MessageBox::Show(Math::Pow(2, 5)+"");
}
```

It is necessary to reference the MSDN site for the syntax of these useful methods. For example, the syntax of the **Round()** method requires the indicator of the fractional digits to be an Int32 type.

```
Math::Round(Double, Int32)
```

The **Math::Sqrt(*n*)** method returns the square root of a specified number. The following demonstrates how to use this method.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
 MessageBox::Show(Math::Sqrt(6.25)+""); // 2.5
}
```

Review questions

1. Given the following code, which is a "handle" in managed code of Visual C++?

```
String^ str = InputBox::Show("How are you?");
```

A. String
B. ^
C. str
D. InputBox

2. Which is the scope resolution operator of Visual C++?
A. //
B. .
C. ->
D. ::

3. The following statement of the main() method will return the value 0 to the __ to indicate that the program ended normally.

```
return 0;
```

A. operating system
B. compiler
C. memory
D. the source code

4. Given the following Visual C++ code, which line will be ignored by the compiler?

```
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()  {  /* end1 */
 MessageBox::Show("Hello, " + "\n" + "world!");
  return 0;
}
```

A. using namespace std;
B. int main()
C. /* end1 */
D. return 0;

5. Given the following code, which is the identifier of a program object?

```
public ref class Form1: public Form { }
```

A. ref
B. public
C. Form1
D. Form

6. Given the following code, which can create the "label1" instance?

```
Label^ label1;
```

A. new Label;
B. gcnew Label;
C. gcnew Label("label1");
D. new Label("label1");

7. Given the following code, which statement is NOT correct?

```
#include "InputBox.cpp"
```

A. "InputBox.cpp" is an individual file.
B. It copies and pastes the source code from the current file to the "InputBox.cpp" file.
C. It copies and pastes the source code from the "InputBox.cpp" file to the current file.
D. It is a example to use library file in Visual C++.

8. In C++, the output of the following code segment is __.

```
MessageBox::Show("Wait! " + "Jack" + "Kyle" + "!");
```

A. Wait! JackKyle!
B. Wait!JackKyle!
C. Wait! Jack Kyle!
D. Wait! Jack Kyle !

9. Given the following code, which statement is correct?

```
this->Size = Drawing::Size(250, 150);
```

A. The word "Size" of this->Size and Drawing::Size refers to a property.
B. The word "Size" of Drawing::Size(250, 150) refers to a method.
C. The word "Size" of this->Size refers to a method.
D. The word "Size" of this->Size and Drawing::Size(250, 150) refers to a method.

10. The output of the following code is __.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()  {
 MessageBox::Show(Math::Sqrt(6.25) + "") ;
}
```

A. 2.5
B. 2
C. 39.0625
D. Math::Sqrt(6.25)

Visual C++

Lab #2 C++ Programming Basics

Learning Activity #1:

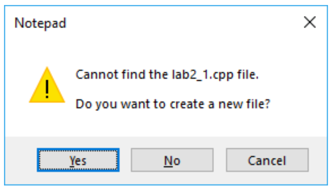
1. Create a new directory called C:\cis223 if it does not exist.

2. Launch the **Developer Command Prompt**. Do not use regular Windows Command Prompt.

3. In the prompt, type **`cd C:\cis223`** and press [Enter] to change to the C:\cis223 directory. The prompt change to:

```
C:\cis223>
```

4. Type **`notepad lab2_1.cpp`** and press [Enter] to use Notepad to create a new source file called lab2_1.cpp. Click Yes if the following window appears.



Notepad

Cannot find the lab2_1.cpp file.

Do you want to create a new file?

5. Type the following contents. The objective is to build a "template" code for creatin a Windows "form" which can be use later. Replace *YourFullNameHere* with your correct name.

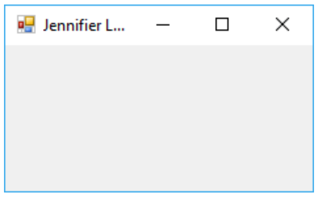
```
#using <System.dll>
#using <System.Windows.Forms.dll>
#using <System.Drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref class Form1: public Form
{

 public: Form1()
 {

  this->Text = "YourFullNameHere";
  this->Size = Drawing::Size(250, 150);

 }
};

[STAThread]
int main()
{
 Application::Run(gcnew Form1);
}
```

6. Click **File**, and then **Save** to save the file.

7. In the prompt, type **`cl /clr lab2_1.cpp /link /subsystem:windows /ENTRY:main`** and press [Enter] to compile the source code. The compiler creates a new file called lab2_1.exe.

```
C:\cis223>cl /clr lab2_1.cpp /link /subsystem:windows /ENTRY:main
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version.....
................
................
/out:lab2_1.exe
lab2_1.obj
```

8. Type **`lab2_1.exe`** and press [Enter] to test the program. A sample output looks:



9. Download the "assignment template" and rename it to lab2.doc if necessary. Capture a screen shot similar to the above figures and paste it to the Word document named lab2.doc (or .docx).

Learning Activity #2:

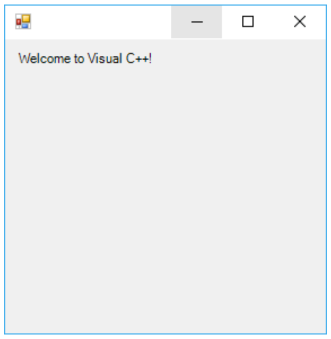
1. Launch the Developer Command Prompt (not the regular Command Prompt) and change to the C:\cis223 directory.

2. Type **`notepad lab2_2.cpp`** and press [Enter] to use Notepad to create a new source file called lab2_2.cpp with the following contents. The objective is to create a Windows "form" with a static Label control in it to display a message.

```
#using <System.dll>
#using <System.Windows.Forms.dll>
#using <System.Drawing.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref class Form1: public Form
{
 public: Form1()
 {

  Label^ label1 = gcnew Label;
  label1->Text = "Welcome to Visual C++!";
  label1->AutoSize = true;
  label1->Location = Point(10, 10);

  Controls->Add(label1);

 }
};

[STAThread]
int main() {
 Application::Run(gcnew Form1());
}
```

3. Type **`cl /clr lab2_2.cpp /link /subsystem:windows /ENTRY:main`** and press [Enter] to compile the source file to create the executable object file.

4. Type **`lab2_2.exe`** and press [Enter] to test the executable file. A sample output looks:


Welcome to Visual C++!

5. Capture a screen shot similar to the above figures and paste it to the Word document named lab2.doc (or .docx).

Learning Activity #3:

1. Launch the Developer Command Prompt (not the regular Command Prompt) and change to the C:\cis223 directory.

2. Under the C:\cis223 directory, use Notepad to create a new text file named **InputBox.cpp** with the following codes. Make sure it is free of typo and error. (This is a custom-made library file. It will not execute alone. It is a supportive library that will build an input box when being imported to another C++ file. **Be sure to keep this file for later use.** This learning activity is a preparation for the rest of lectures.)

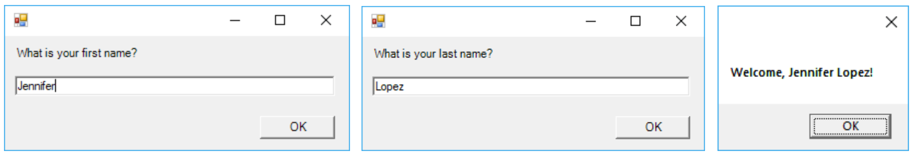
```
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref struct InputBox
{
 private: Form^ form1;
 private: Label^ label1;
 private: TextBox^ textBox1;
 public: static String^ Text;

 private: Void button1_Click(Object^ sender, EventArgs^ e)
 {
  if (textBox1->Text == "")
  {
   MessageBox::Show("No value given.");
   Text = "";
  }
  else
  {
   Text = textBox1->Text;
   form1->Close();
  }
 }

 public: static String^ Show(String^ s)
 {
  Text = s;
  InputBox();
  return Text;
 }

 public: InputBox()
 {
  form1 = gcnew Form;

  label1 = gcnew Label;
  label1->Size = Size(320, 25);
  label1->Text = Text;
  label1->Location = Point(10, 10);
  form1->Controls->Add(label1);

  textBox1 = gcnew TextBox;
  textBox1->Size = Size(320, 25);
  textBox1->Location = Point(10, 40);
  form1->Controls->Add(textBox1);

  Button^ button1 = gcnew Button;
  button1->Location = Point(label1->Width - 65, 80);
  button1->Text = "OK";
  button1->Click += gcnew EventHandler(this, &InputBox::button1_Click);

  form1->Controls->Add(button1);

  form1->Size = Size(label1->Width + 40, label1->Height + textBox1->Height +
button1->Height + 85);

  Application::Run(form1);
 }

};
```

3. Type **`notepad lab2_3.cpp`** and press [Enter] to use Notepad to create a new source file called lab2_3.cpp with the following contents. The objective is to import the source code of the “InputBox.cpp” file and use it to take user input.

```
#include "InputBox.cpp"  // import the library

int main()
{
 String^ fn = InputBox::Show("What is your first name?");

 String^ ln = InputBox::Show("What is your last name?");

 MessageBox::Show("Welcome, " + fn + " " + ln + "!");
}
```

4. Type **`cl /clr lab2_3.cpp /link /subsystem:windows /ENTRY:main`** and press [Enter] to compile the source file to create the executable object file.

5. Type **`lab2_3.exe`** and press [Enter] to test the executable file. A sample output looks:



What is your first name?

Jennifer

OK

What is your last name?

Lopez

OK

Welcome, Jennifer Lopez!

OK

6. Capture a screen shot similar to the above figures and paste it to the Word document named lab2.doc (or .docx).

Learning Activity #4:

1. Launch the Developer Command Prompt (not the regular Command Prompt) and change to the C:\cis223 directory.

2. Under the C:\cis223 directory, use Notepad to create a new text file named **OutputBox.cpp** with the following codes. (This is a custom-made library file. It will not execute alone. It is a supportive library that will build an input box when being imported to another C++ file. **Be sure to keep this file for later use.** This learning activity is a preparation for the rest of lectures.)

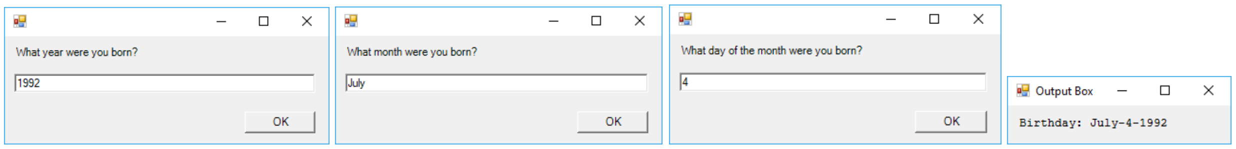
```
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Drawing;
using namespace System::Windows::Forms;

public ref class OutputBox: public Form
{
 Label^ label1;

 public:

  OutputBox(String^ s)
  {
   this->Text = "Output Box";
   label1 = gcnew Label();
   label1->AutoSize = true;
   label1->Location = Point(10, 10);
   label1->Text = s;
   label1->Font = (gcnew System::Drawing::Font(L"Courier New", 9));
   Controls->Add(label1);

   if (label1->Width < 200) { this->Width = 250; }
   this->Height = label1->Height + 60;
  }

  static void Show(String^ s)
  {
   Application::Run(gcnew OutputBox(s));
  }
};
```

3. Type **`notepad lab2_4.cpp`** and press [Enter] to use Notepad to create a new source file called lab2_4.cpp with the following contents. The objective is to create a custom-made output box (similar to a message box).

```
#include "InputBox.cpp"
#include "OutputBox.cpp"  // import the library

int main()
{
 int y = Convert::ToInt32(InputBox::Show("What year were you born"));

 String^ m = InputBox::Show("What month were you born");

 int d = Convert::ToInt32(InputBox::Show("What day of the month were you born"));

 OutputBox::Show("Birthday: " + m + "-" + d + "-" + y);
}
```

4. Type **`cl /clr lab2_4.cpp /link /subsystem:windows /ENTRY:main`** and press [Enter] to compile the source file to create the executable object file.

5. Type **`lab2_4.exe`** and press [Enter] to test the executable file. A sample output looks:


What year were you born?
1992
OK
What month were you born?
July
OK
What day of the month were you born?
4
OK
Output Box

```
Birthday: July-4-1992
```

6. Capture a screen shot similar to the above figure and paste it to the Word document named lab2.doc (or .docx).

Learning Activity #5:

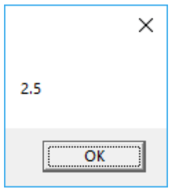
1. Launch the Developer Command Prompt (not the regular Command Prompt) and change to the C:\cis223 directory.

2. Type **`notepad lab2_5.cpp`** and press [Enter] to use Notepad to create a new source file called lab2_5.cpp with the following contents. The objective is to

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
 MessageBox::Show(Math::Sqrt(6.25)+""); // 2.5
}
```

3. Type **`cl /clr lab2_5.cpp /link /subsystem:windows /ENTRY:main`** and press [Enter] to compile the source file to create the executable object file.

4. Type **`lab2_5.exe`** and press [Enter] to test the executable file. A sample output looks:



```
2.5
```

5. Capture a screen shot similar to the above figure and paste it to the Word document named lab2.doc (or .docx). You can also save the document as .pdf file.

Submittal

1. Complete all the 5 learning activities and the programming exercise in this lab.

2. Create a .zip file named **lab2.zip** containing ONLY the following self-executable files.
   - Lab2_1.exe
   - Lab2_2.exe
   - Lab2_3.exe
   - Lab2_4.exe
   - Lab2_5.exe
   - Lab2.doc (or lab2.docx or .pdf) [You may be given zero point if this Word document is missing]

3. Log in to course web site (e.g. Canvas or Blackboard) and enter the course site.

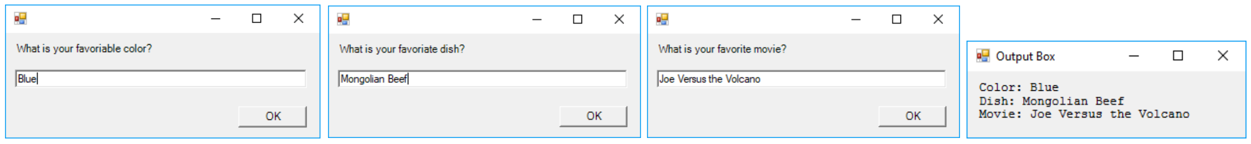
4. Upload the zipped file to **Question 11** of Assignment as response.

Programming Exercise 02:

1. Launch the Developer Command Prompt.

2. Use Notepad to create a new text file named **ex02.cpp**.

3. In the Notepad, write a program that use the input box provided by the InputBox.cpp file to ask user to enter three questions: (1) "What is your favoriable color?", (2) "What is your favoriate dish?", and (3) "What is your favorite movie?". Then display then in the output box provided by the OutputBox.cpp file.



What is your favoriable color?

Blue

What is your favoriate dish?

Mongolian Beef

What is your favorite movie?

Joe Versus the Volcano

```
Color: Blue
Dish: Mongolian Beef
Movie: Joe Versus the Volcano
```

4. Compile the source code to produce executable code.

5. Download the "programming exercise template" and rename it to ex02.doc if necessary. **Copy your source code to the file** and then capture the screen shots similar to the above ones and paste it to the Word document named "ex02.doc" (or .docx).

6. Compress the source file (**ex02.cpp**), executable code (**ex02.exe**), and Word document (**ex02.doc**) to a .zip file named "**ex02.zip**".

Grading criteria:
You will earn credit only when the following requirements are fulfilled. No partial credit is given.
- You successfully submit both source file and executable file.
- Your source code must be fully functional and may not contain syntax errors in order to earn credit.
- Your executable file (program) must be executable to earn credit.

Threaded Discussion

Note: Students are required to participate in the thread discussion on a weekly basis. Student must post at least two messages as responses to the question every week. Each message must be posted on a different date. Grading is based on quality of the message.

Question: Class, after reviewing the lecture note, do you agree that the fundamental programming skills and knowledge you might have obtain from other programming course(s) apply to Visual C++? Why or why not? If you can, provide an example to support your perspective. [There is never a right-or-wrong answer for this question. Please free feel to express your opinion.]

Be sure to use proper college level of writing. Do not use texting language.