Application of Programming in Engineering Problems

9. Building Graphical User Interface

M. Heege, *Expert C++/CLI: .NET for Visual C++ Programmers.* Berkeley, CA: Apress, 2007. [Electronic resource]

I. Horton, *Ivor Horton's Beginning Visual C++ 2010.* Indianapolis, Ind.: Wiley, 2010. (Ch. 1, and C++/CLI sections from Ch. 2 to Ch. 10) [Electronic resource]

W.L. Chan, *C++ Graphical User Interface Programming Using Microsoft .NET Framework.* Singapore: McGraw-Hill, 2011.

9.1 Developing a Simple Graphical User Interface (GUI)

C++/Common Language Infrastructure (C++/CLI)

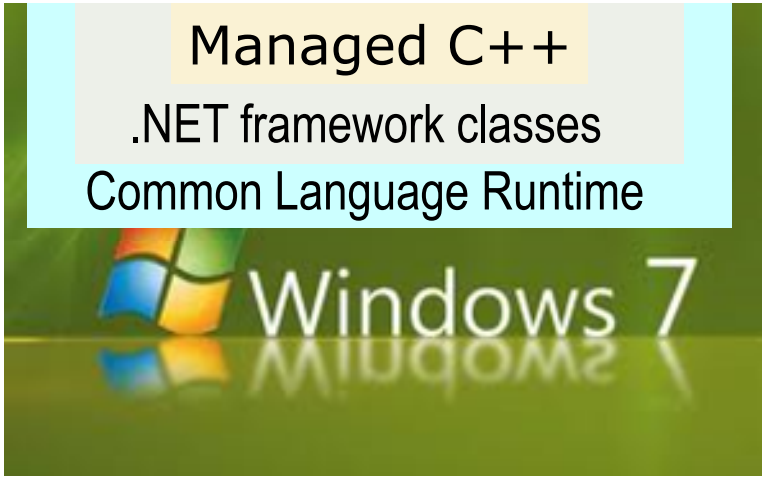
- So far the C++ programs we have learnt essentially follow the ANSI standard (Native C++)
  - It means that those programs should be able to be compiled and run in any computer platform, including Windows, UNIX, Linux, Mac OS, etc.
- Microsoft has developed in its Visual C++ 2010 a new set of **extension** keywords (known as the C++/CLI) to generate code that targets the .NET framework. The code is referred to as ***managed code***
  - Hence the codes that are not controlled by the .NET framework are known as **unmanaged** code.

Native C++ → Unmanaged code
C++/CLI → Managed code

- When developing applications using Visual C++ 2010, it can be either managed or unmanaged applications
  - An unmanaged application runs directly under Windows

  ⇒ No extra software is needed to run the application
  - A managed application, however, runs in CLR (**Common Language Runtime**) - Microsoft's implementation of CLI

  ⇒ Your computer needs to install the Microsoft .NET Framework in order to run a managed application.



- So why do we want to develop managed application?
  - ⇒ Most new Windows systems have included the .NET Framework – It is installed when installing Windows or its Updates
  - ⇒ More importantly, Microsoft provides a set of powerful .NET class libraries that can be used **only** by managed applications

Aims: Productivity and Security

- One of the important tasks that greatly benefits from .NET class libraries is the development of graphical user interface (**GUI**).

Graphical User Interface

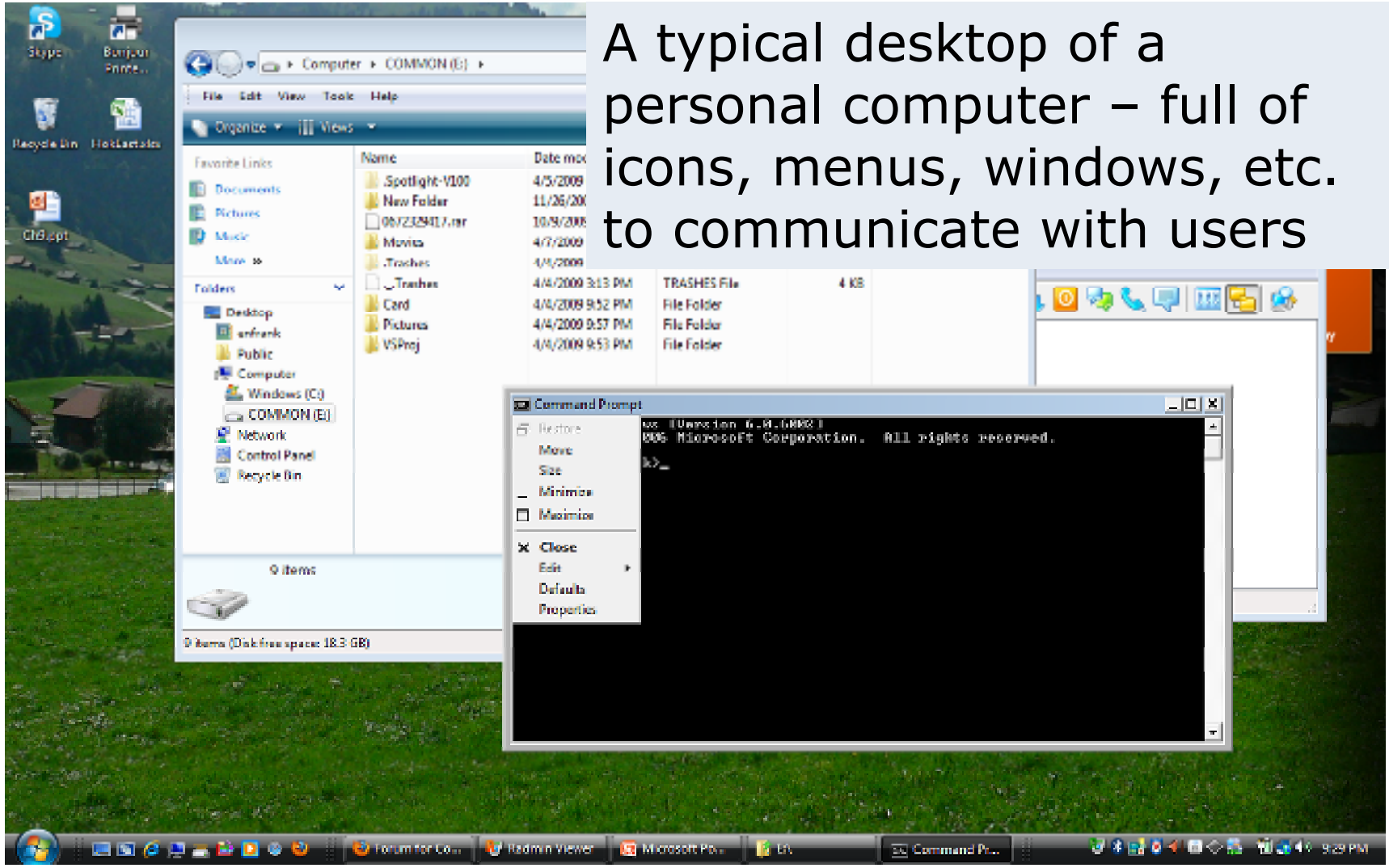
- User-friendliness is one of the important criteria to evaluate the merit of a computer system
  - Particularly for home users, a user friendly operating system is very important
  - Command-line interface has been replaced by Graphical User Interface (GUI) for personal computers

Console application

Mouse vs Keyboard

- Graphical User Interface
  - ⇒ Using a graphical approach, through icons, menus, windows, etc. to **interact** with users.

9 Building Graphical User Interface



A typical desktop of a personal computer – full of icons, menus, windows, etc. to communicate with users

`System::Windows::Forms`

MFC: Microsoft Foundation Library

ATL: Active Template Library

Creating Windows Forms

window **layout** and **controls**

- For Windows, GUI controls are contained in a **form**. Hence developing Windows Forms means developing GUIs
  - For unmanaged applications, developing Forms needs knowing the MFC or ATL libraries - difficult and NOT supported by C++/CLI.
- Visual C++ 2010 offers Rapid Application Development (RAD) support for building managed Windows Forms applications
  - Managed applications allow GUI to be incorporated in a program following a simple click-and-pick procedure
    - Something similar to using Visual Basic

Partially vs fully known

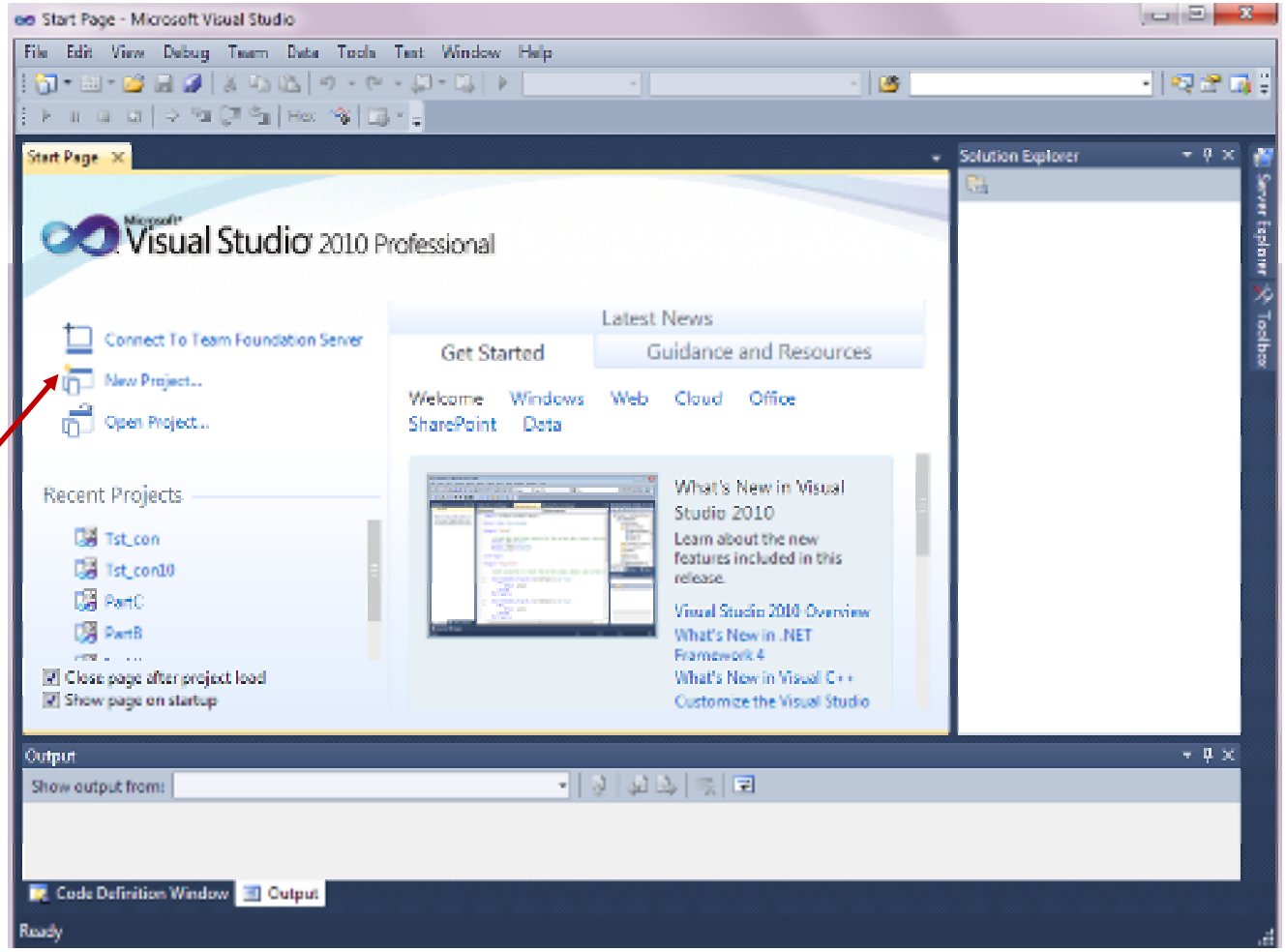
- Managed Windows Forms applications **can also use** native **C++ libraries**
  - **Allow GUI to be introduced into our previously developed unmanaged applications.**

Developing a simple GUI

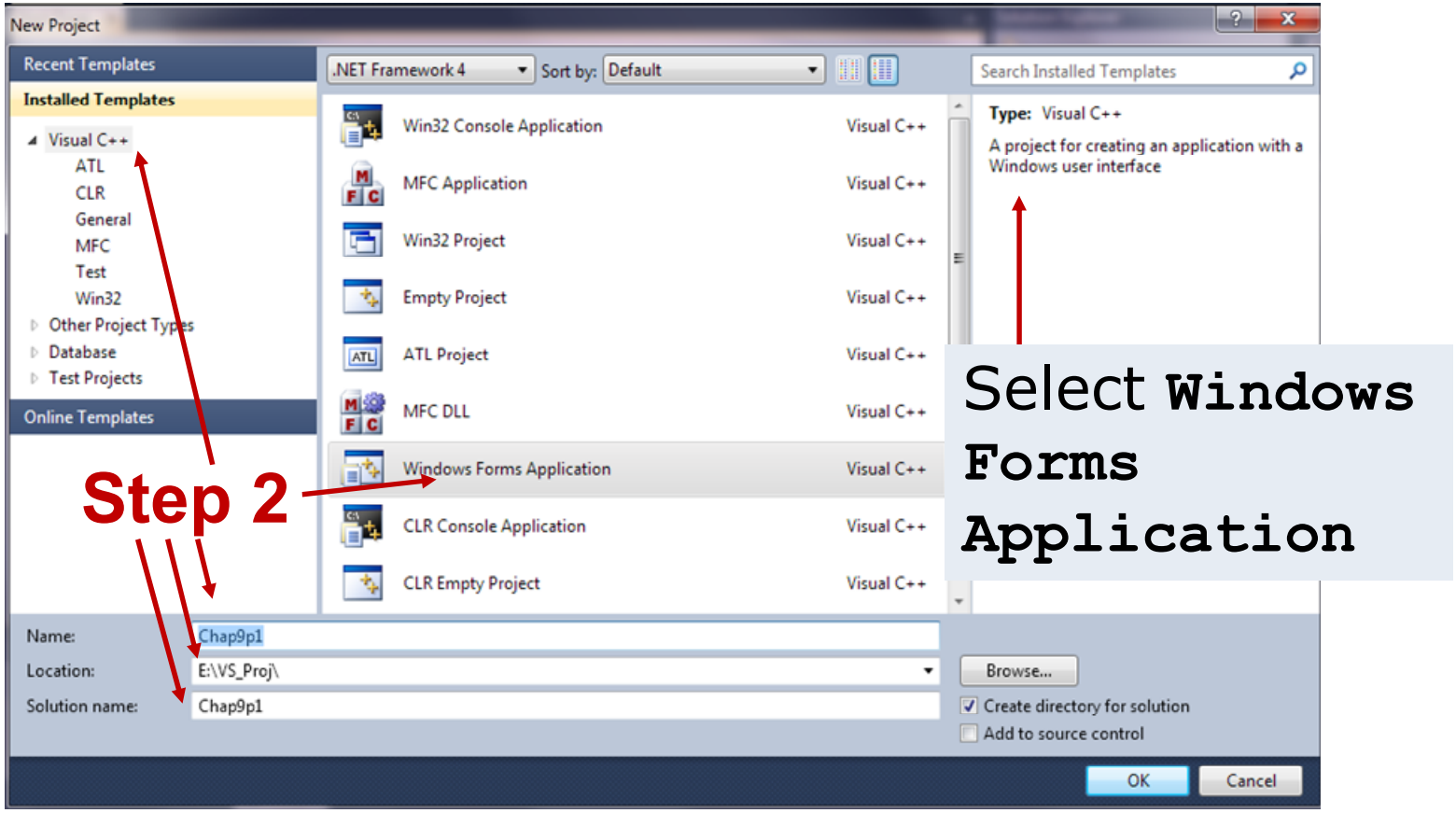
Step 1:

Start Visual Studio 2010.

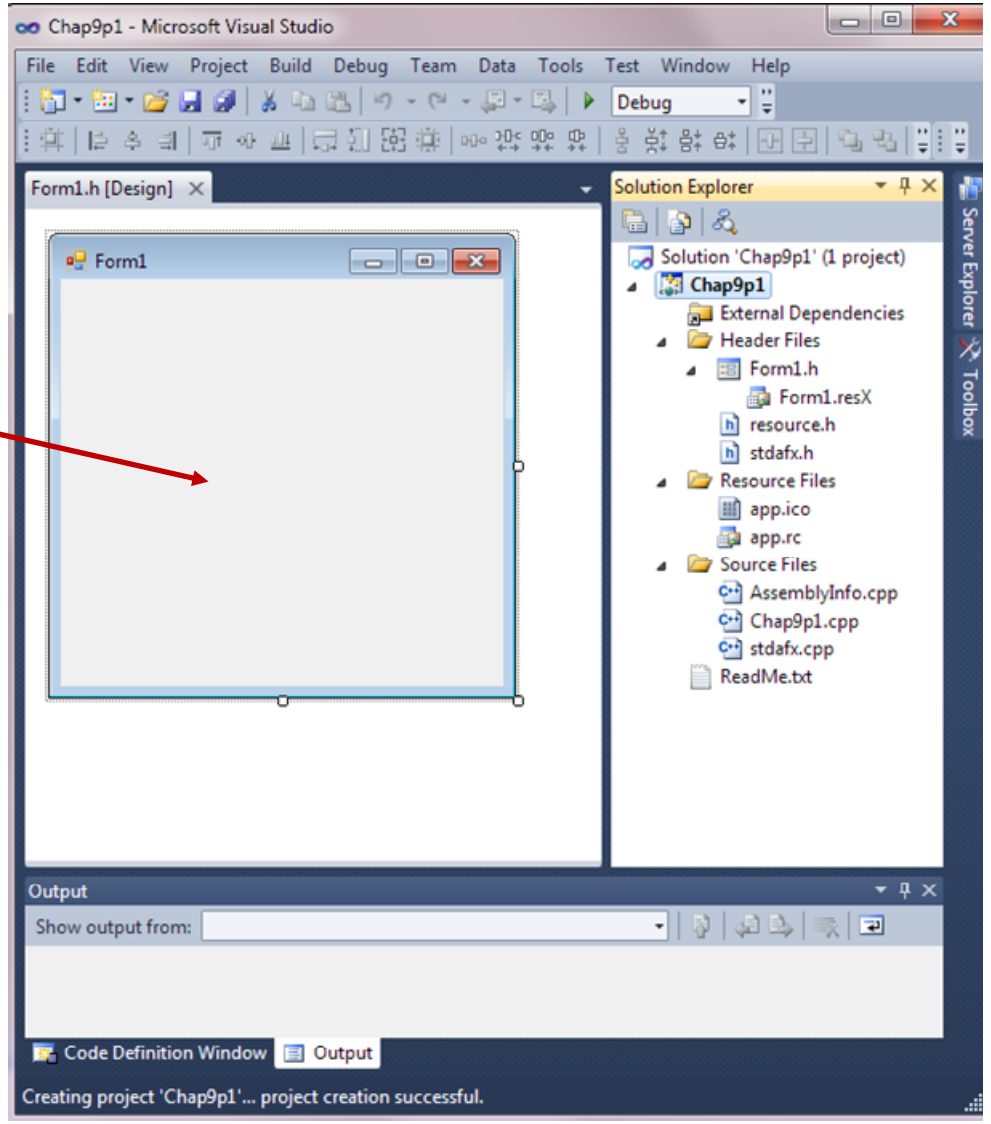
Click **`New Project...`**



9 Building Graphical User Interface



A plain form will be created for you to fill in the details



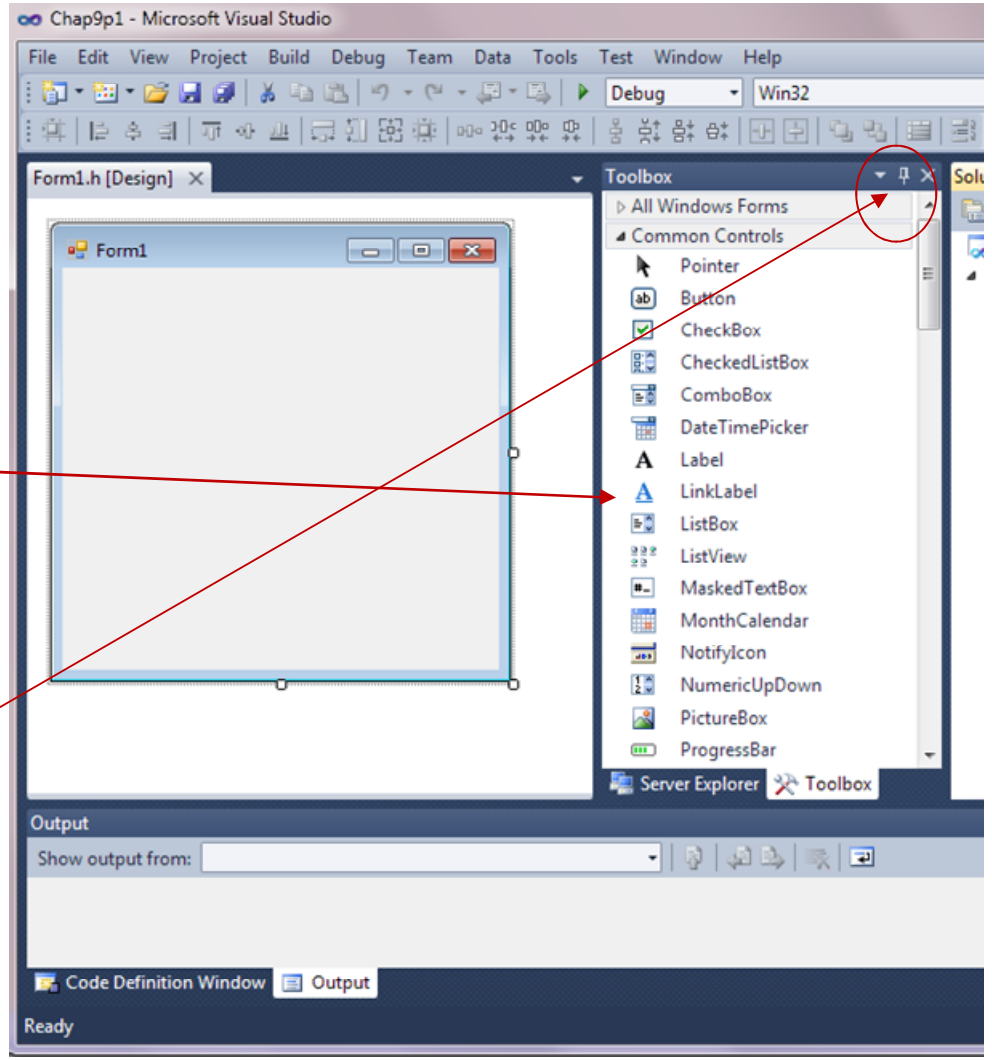
Using the Toolbox to Build the User Interface

Step 3

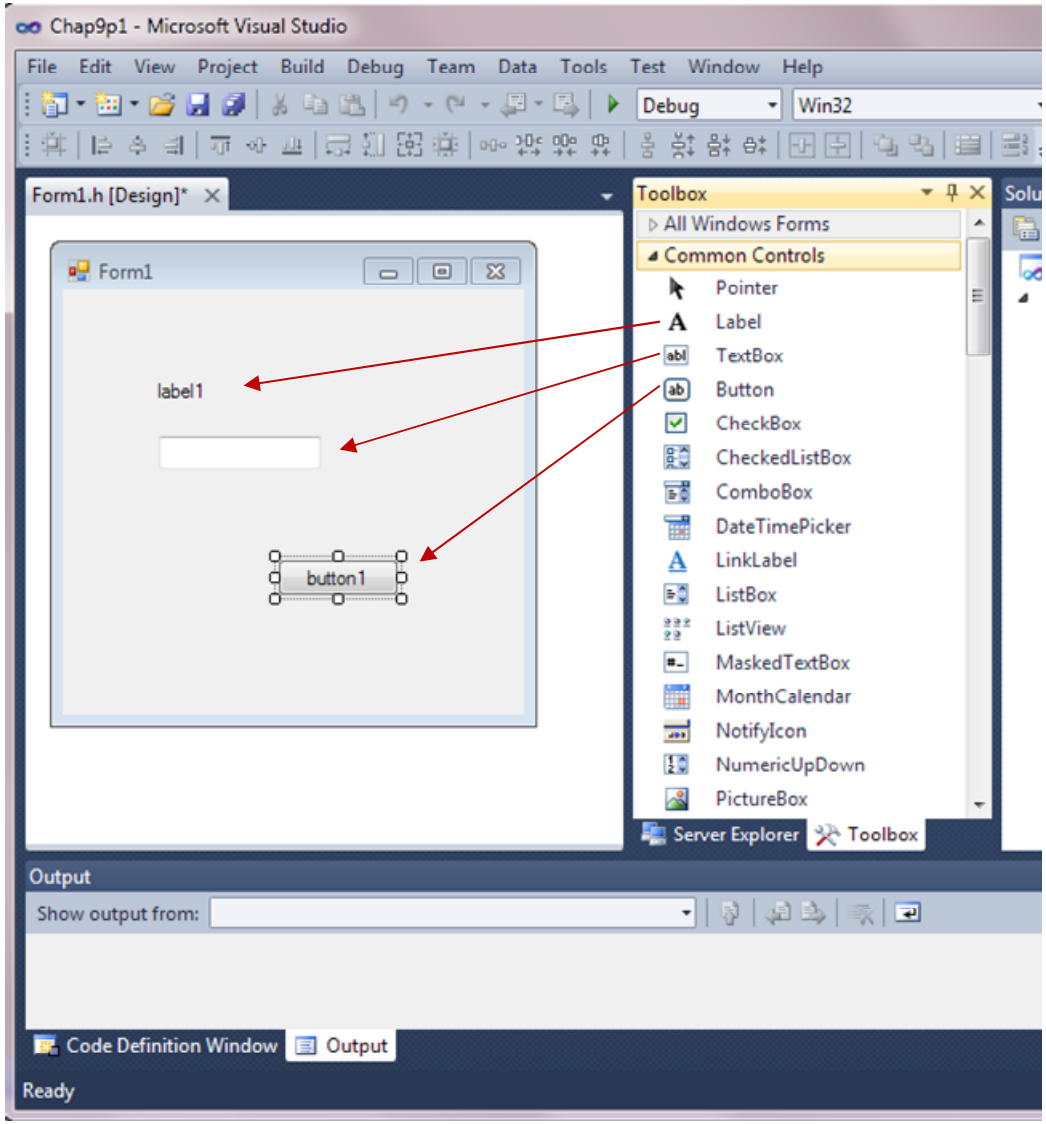
The toolbox is a tabbed window that contains all GUI control elements

If you cannot see it, click **`view` → `ToolBox`**

Try to click the pin icon, see what happen

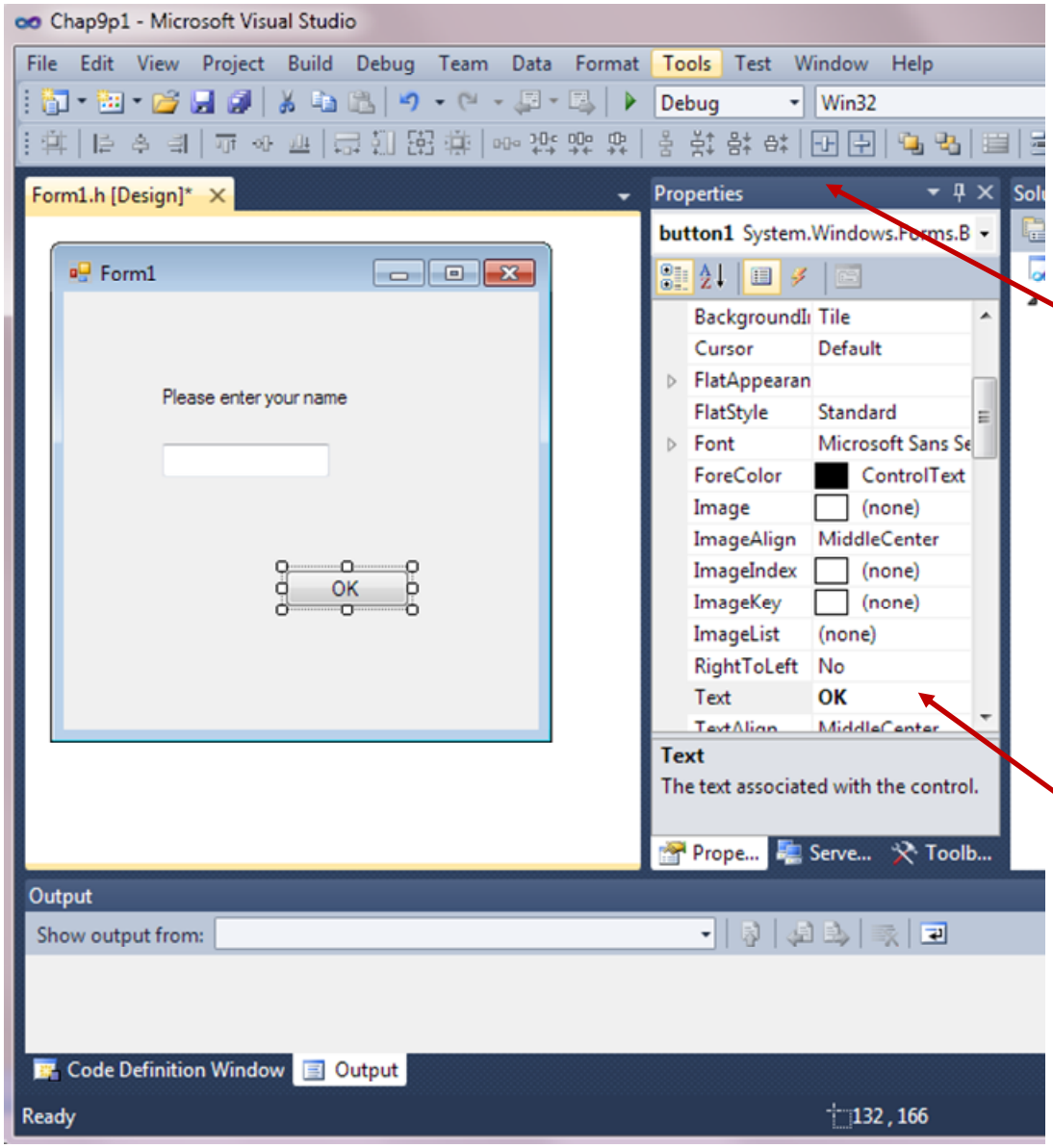


9 Building Graphical User Interface



Step 3

From the toolbox, drag a **`Label`**, a **`TextBox`** and a **`Button`** to the form

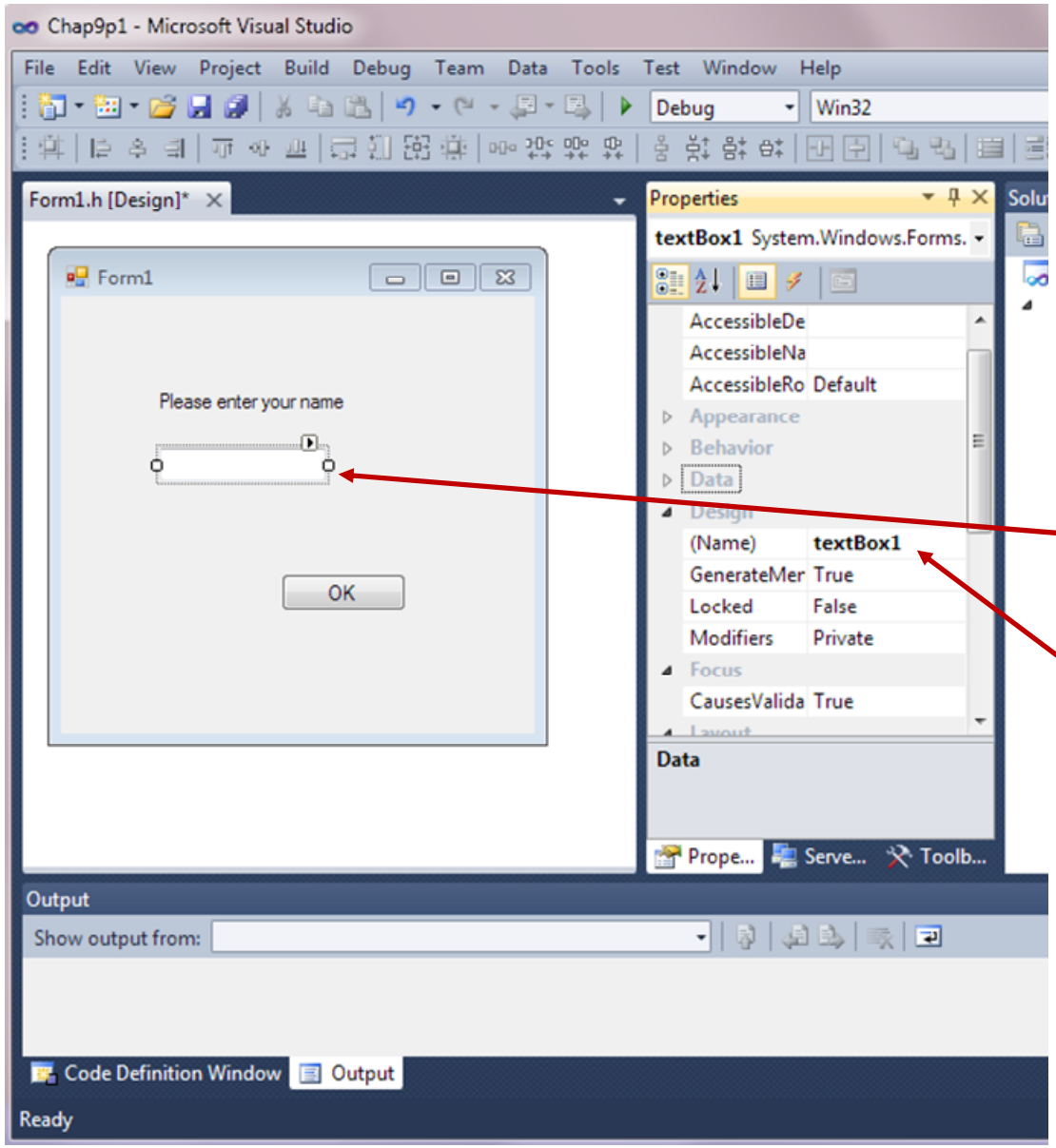


Step 4

You can find the property of each component in the **`Properties`** window (If cannot find it, select: **`View-> Other Windows -> Properties Window`**)

Try to modify the label to "**`Please enter your name`**" and the label of the button to "**`OK`**" by modifying the **`Text`** item in their **`Properties`**

9 Building Graphical User Interface

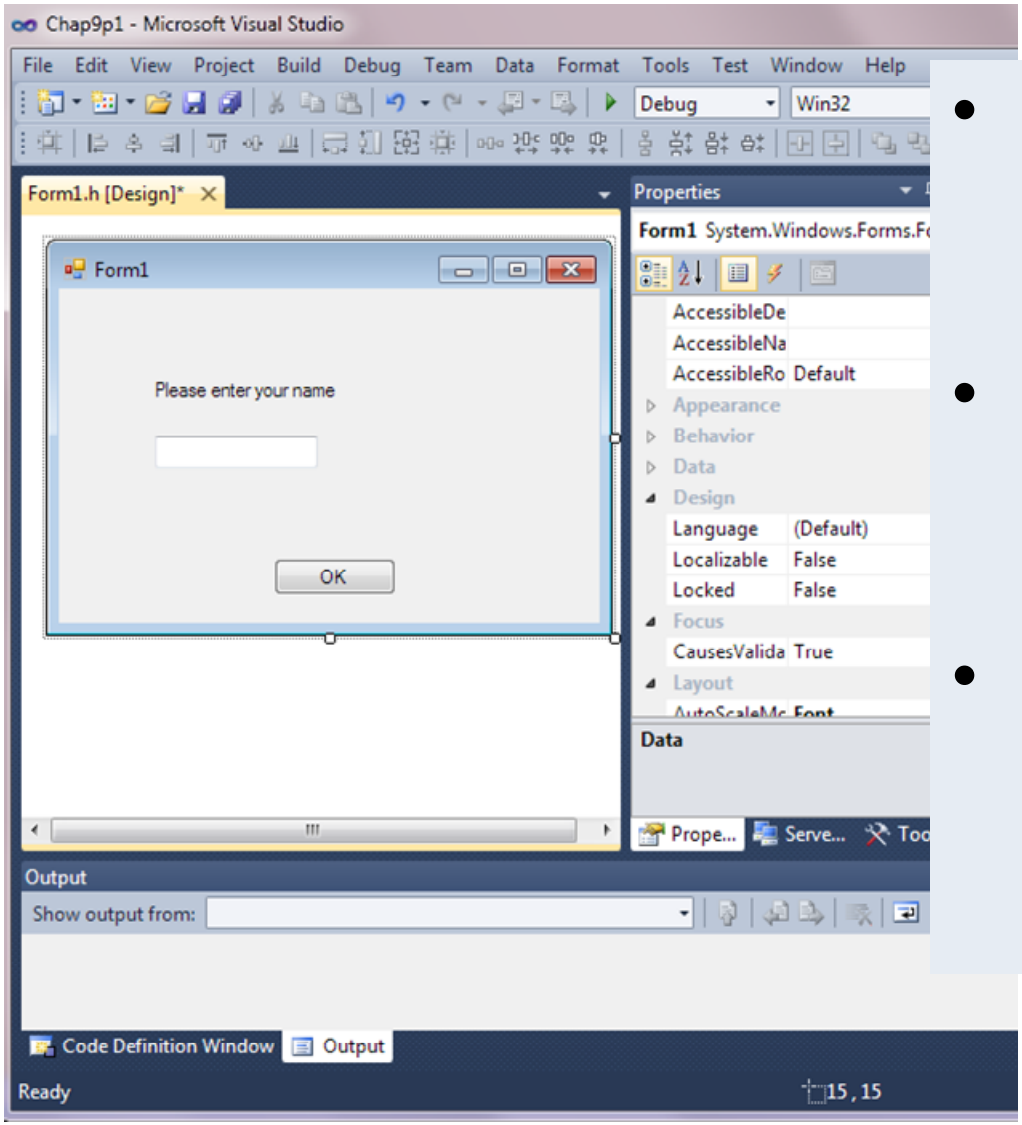


You can find in the **`Properties`** window the name of each control.

e.g. the name for this text box is **`textBox1`**

It is important for developing program codes for them

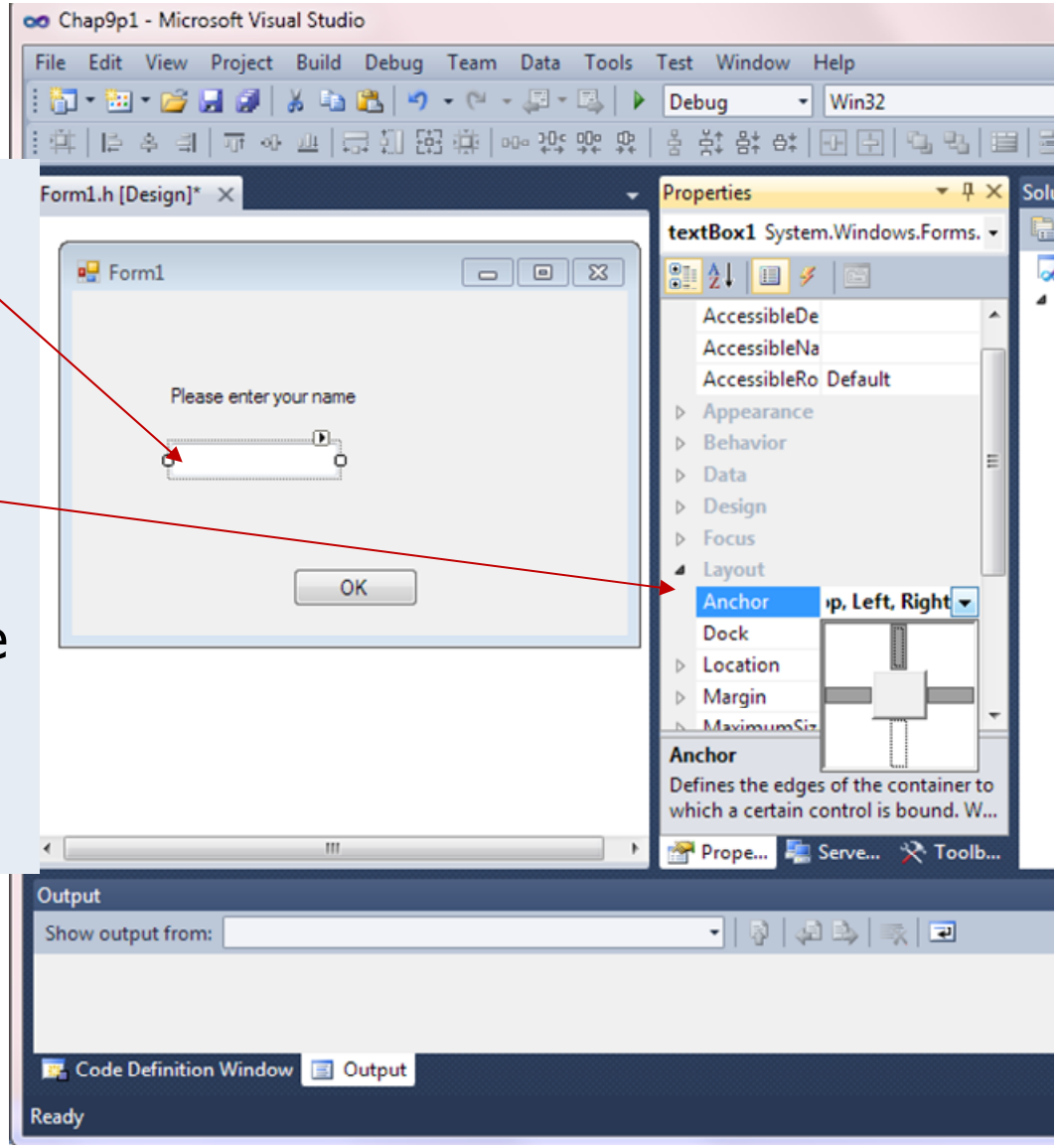
9 Building Graphical User Interface



- Try to resize the form. You will find that the size of the textbox will not change accordingly.
- By default, all components are **anchored** with respect to the top and left boundaries of the form.
- Hence, not only the size, but the position of all components will not change when resizing the form

Step 5

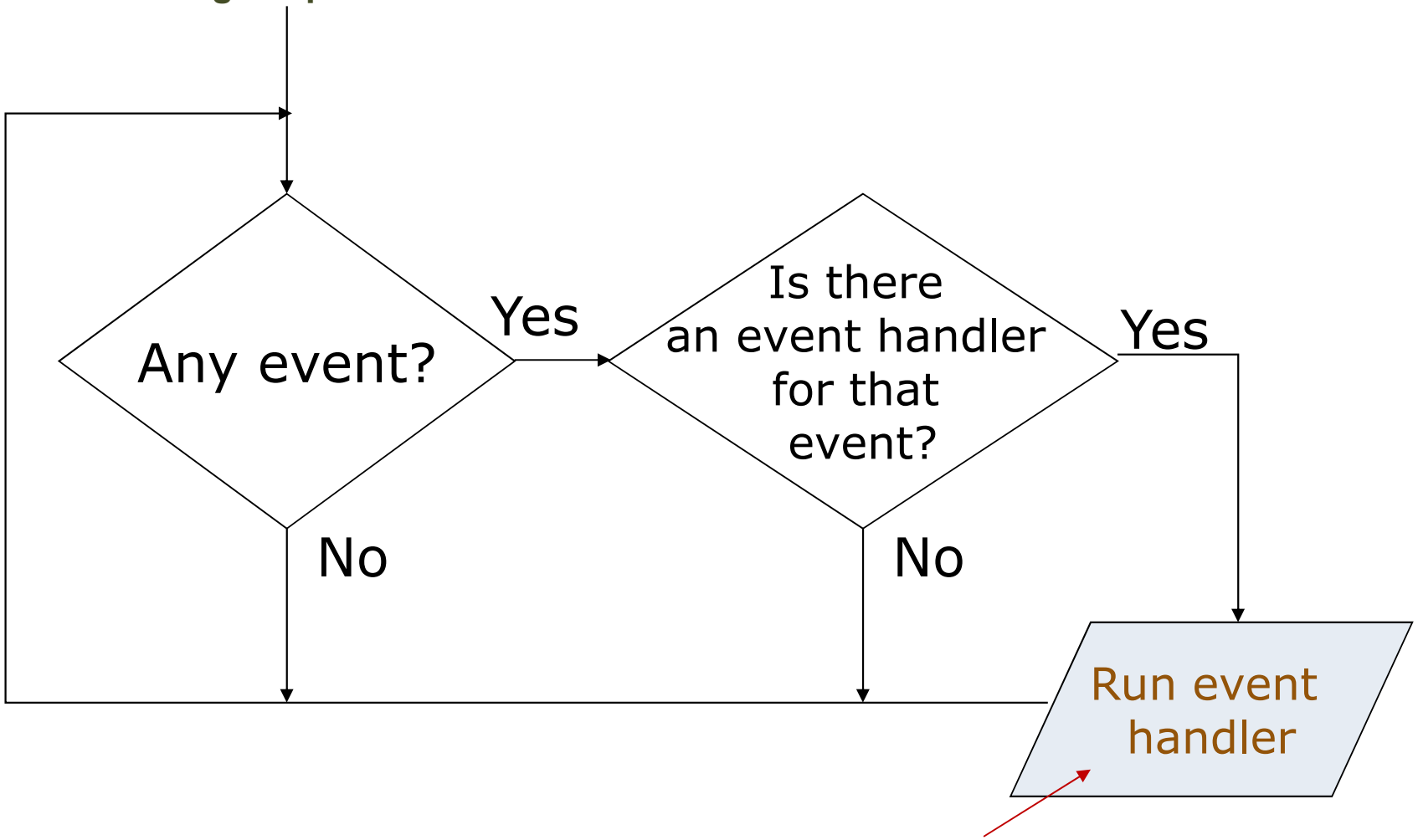
- Click the textbox
- On the **Properties** window, change the **Anchor** property to "**Top Left Right**"
- Try to enlarge your form. You will see the size of the textbox changes with the form size



Writing Event Handler

- A major task that a GUI designer needs to do is to determine what will happen when a GUI is invoked
- Every GUI component may generate different kinds of "events" when a user makes access to it using his mouse or keyboard
  - E.g. if a user moves his mouse on top of a button, an event of that button will be generated to the Windows system
  - E.g. if the user further clicks, then another event of that button will be generated (actually it is the click event)
- For any event generated, the **system** will first check if there is an event handler, which defines the action for that event
- For a GUI designer, he needs to develop the event handler to determine the action that he wants Windows to take for that event.

Writing C++ programs



The part that a GUI designer needs to develop

Step 6 – Add Event Handler

- Double-click the **OK** button such that the default event handler **button1_Click** will be introduced in **Form1.h**
  - Add the following codes to the function **button1_Click()**

```
System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
{ String^ message = "Hello "; // ^ specifies a tracking handle
  String^ title = "Welcome"; // String is a managed class
  MessageBox::Show(message,title,
                   MessageBoxButtons::OK);
}// A tracking handle is similar to a native C++ pointer
```

When the **OK** button is clicked, a message box will be invoked to show a message.

```
String^ message = "Hello ";
```

- The symbol "**^**" identifies a tracking handle of the class **`String`**
- A tracking handle is similar to a native C++ pointer which stores an address, but the address it contains is automatically updated if the object it references is moved
- NO address arithmetic or casting is allowed for tracking handles
- Tracking handle is for referencing objects created in **CLR heap**, including objects that are of reference class type (e.g. the **`String`** class)

```
MessageBox::Show(message,title,
                 MessageBoxButtons::OK);
```

- **Show()** is a static member function of the **MessageBox** class. We can call any member function of a class **without** instantiating an object of that class
  - **message** is the message to be displayed in the message box. It is of the managed class **String^**
  - **title** is the title of the message box. It is of the managed class **String^**
  - **OK** is a static member enumeration of the class **MessageBoxButtons**. If **OK** is used, only the **OK** button will be shown. There can be other buttons, such as, **YesNo**, **OKCancel**, **AbortRetryIgnore**, etc.

Use **Help** of Visual C++ 2010 to learn **MessageBox** and **MessageBoxButtons**

Step 7 – Build and run the program

Managed data types - Common Type System (CTS)

- All GUI inputs are of **managed** type, data received from these GUIs may be different from the unmanaged version
  - For example, the string received from the textbox is of managed type
    - It needs conversion if we want to use it with unmanaged codes

```
char abc[] = textBox1->Text;
```

unmanaged type

`Text` is the input made by user in `textBox1`. It is of managed type

- Conversion between managed type and unmanaged type: **String^ → char ***

```
#include <string.h>  // Due to strcat; ADD at the beginning

using namespace System::Runtime::InteropServices; //Must add

private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
{   char message[100] = "Hello "; //Unmanaged type
    String ^ tbstr = textBox1->Text;
    char *name =
    (char*)Marshal::StringToHGlobalAnsi(tbstr).ToPointer();
    strcat(message,name);  //Combine "Hello " with name
    String^ Mmge=Marshal::PtrToStringAnsi((IntPtr)message);
    String^ title = "Welcome";
    MessageBox::Show(Mmge,title,MessageBoxButtons::OK);
    Marshal::FreeHGlobal((IntPtr)name);  // just like delete
}
```

```
using namespace
System::Runtime::InteropServices;
```

- The class name such as **Marshal** is defined under the **System::Runtime::InteropServices** namespace

  i.e. the full name of the class is
  System::Runtime::InteropServices::Marshal

  - The above statement needs to be added at the beginning of **Form1.h** with the other namespace definitions.

```
String ^ tbstr = textBox1->Text;
```

- For C++/CLI, any string should be defined as a handle to an object of the **System::String** class
  - Here, **textBox1->Text** is a managed string. That's why it should be assigned to the handle **tbstr** as above.

After copying, will return an object of type **IntPtr**

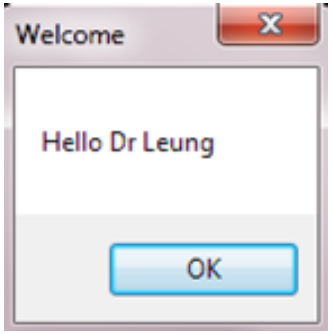
```
char *name = (char*)Marshal::
      StringToHGlobalAnsi(tbstr).ToPointer();
    :
Marshal::FreeHGlobal((IntPtr)name);
```

- To convert a managed string to an unmanaged string, we use the function **StringToHGlobalAnsi()**, which is a static member function of the class **Marshal**
  - The function will return an object of class **IntPtr**
  - We need to call the member function **ToPointer()** of **IntPtr** to convert the object to a pointer, but we haven't specified what kind of pointer it is
  - Finally, casting the pointer to a character pointer
  - As some memory is used in the unmanaged heap (pointed by **name**), we free it using the function **FreeHGlobal()**.

HGlobal = Global **heap**

```
strcat(message,name);
String^ Mmge=Marshal::PtrToStringAnsi((IntPtr)message);
```

- `strcat()` combines the string "`Hello `" in `message[]` with the string the user entered into the textbox, and stores it in `message[]`
- `Marshal::PtrToStringAnsi((IntPtr)message)` allocates a managed `String` and copies the unmanaged ANSI string in `message[]` into it.



- Conversion between managed type and unmanaged type: **`String^` → `char *`**
  - Notice that both native C++ and managed C++ codes can be present in the same C++/CLI program
    - However, the heap for the native C++ variables is the unmanaged heap, which needs to be freed after use. In the above, the memory pointed by **`name`** should be freed.
    - The heap for the managed C++ variables is the CLR heap, which is also termed garbage-collected heap.
      - We do **NOT** need to worry about the management and clean-up of this heap.
      - To allocate memory in the CLR heap, use the operator **`gcnew`**
  - The above example is for illustrating how to convert between managed and unmanaged types
    - A **simpler managed** C++ program can do the same job.

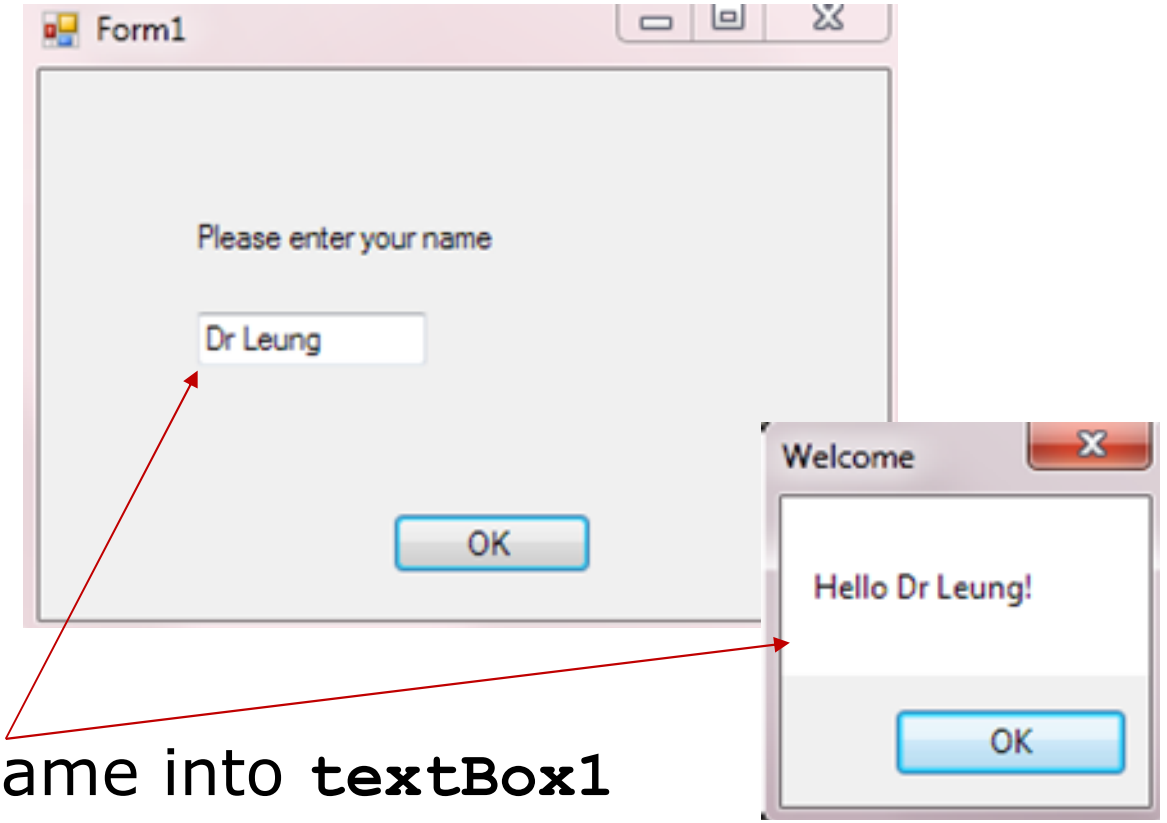
Step 8 – Receive TextBox Input

- Modify the function **`button1_Click()`** as follows:

```
System::Void button1_Click(System::Object *
sender, System::EventArgs * e)
{   String ^ message = "Hello ";
    String ^ title = "Welcome";
    Char a='!';  //A managed character
    MessageBox::Show(message + textBox1->Text + a,
title, MessageBoxButtons::OK);
}
// A text or string can easily be joined to
another string by direct "adding", if they are of
managed type, e.g. String^, Char, etc.
```

`Char` in fact is the **`System::Char`** base class for CLR; it is equivalent to **`__wchar_t`** in C++/CLI

Step 9 – Build and run the program



Enter a name into **`textBox1`**

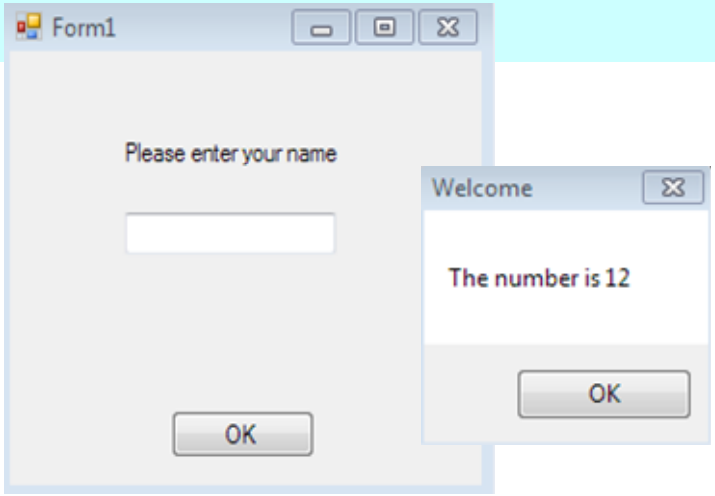
- Conversion between managed types: **`int` → `String^`**

```
System::Void button1_Click(System::Object * sender,
System::EventArgs * e)
{   String^ title = "Welcome";
    String^ message = "The number is ";
    int a=12;
    String^ no =""+a;//int converted to string automatically
    MessageBox::Show(message+no,title,MessageBoxButtons::OK);
}
```

  - We can replace the above by:

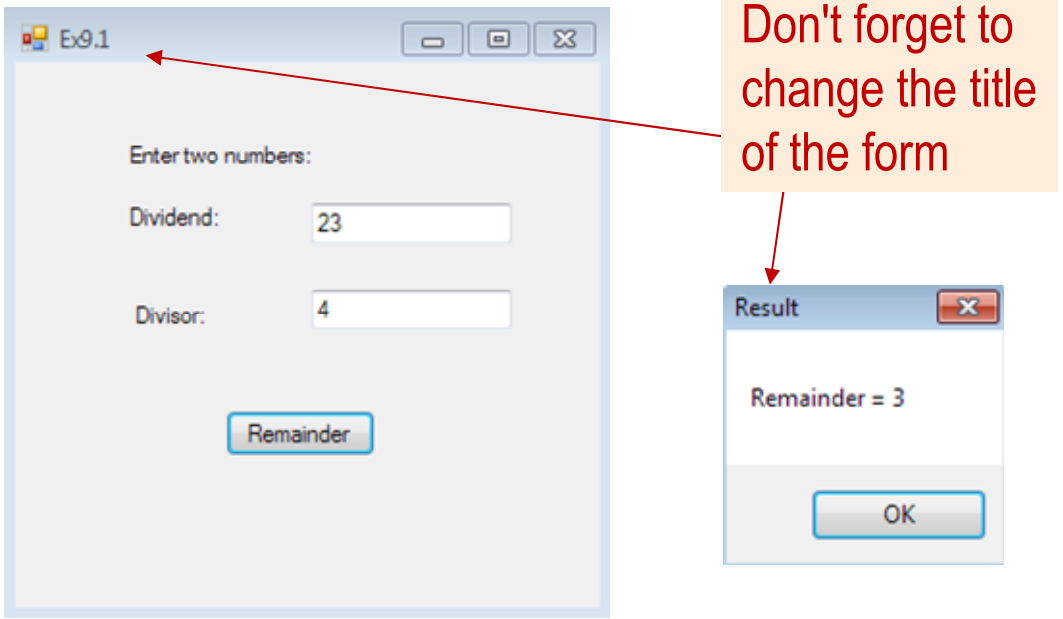
```
String^ no;  //default is ""
no=no+a;
```

What about **`String^` → `int`**?



Exercise 9.1

Modify the form as shown below. When the user enters 2 numbers and clicks **`Remainder`**, a message box will be shown to display the remainder of the integer division.



Exercise 9.1 (cont.)

Hint 1 : To convert an unmanaged string to an integer, you may include the library **<stdlib.h>** and use

```
int  atoi (char *abc)
/* Return the integer converted, if
possible, from the string pointed by abc */
```

Hint 2: If managed type is used, you may consider using the member function of the **Convert** class:

```
int  System::Convert::ToInt32(String)
```

9.2 Working with Unmanaged Code

Working with Unmanaged Code

- Managed C++ application can directly work with the legacy C++/C codes
  - It means all C++ programs developed previously can be used directly with **managed** programs
  - It is a unique feature of Visual C++, not the case for other components of Visual Studio, such as Visual Basic or Visual C#
- The **simplest** way for a managed C++ application to call legacy C++/C functions is by means of using **static library**.

General Approach

Main Program

MainProject.cpp

```
#include "DogClass.h"
#include "CatClass.h"
void main()
{      :
  DOG Bobby;
  CAT Frisky;
  Bobby.DrawShape();
  Frisky.DrawShape();
       :
  Frisky.PlaySound();
       :
}
```

DogClass.h

CatClass.h

```
class CAT
 {
        :
   public:
          int DrawShape();
          int PlaySound();
```

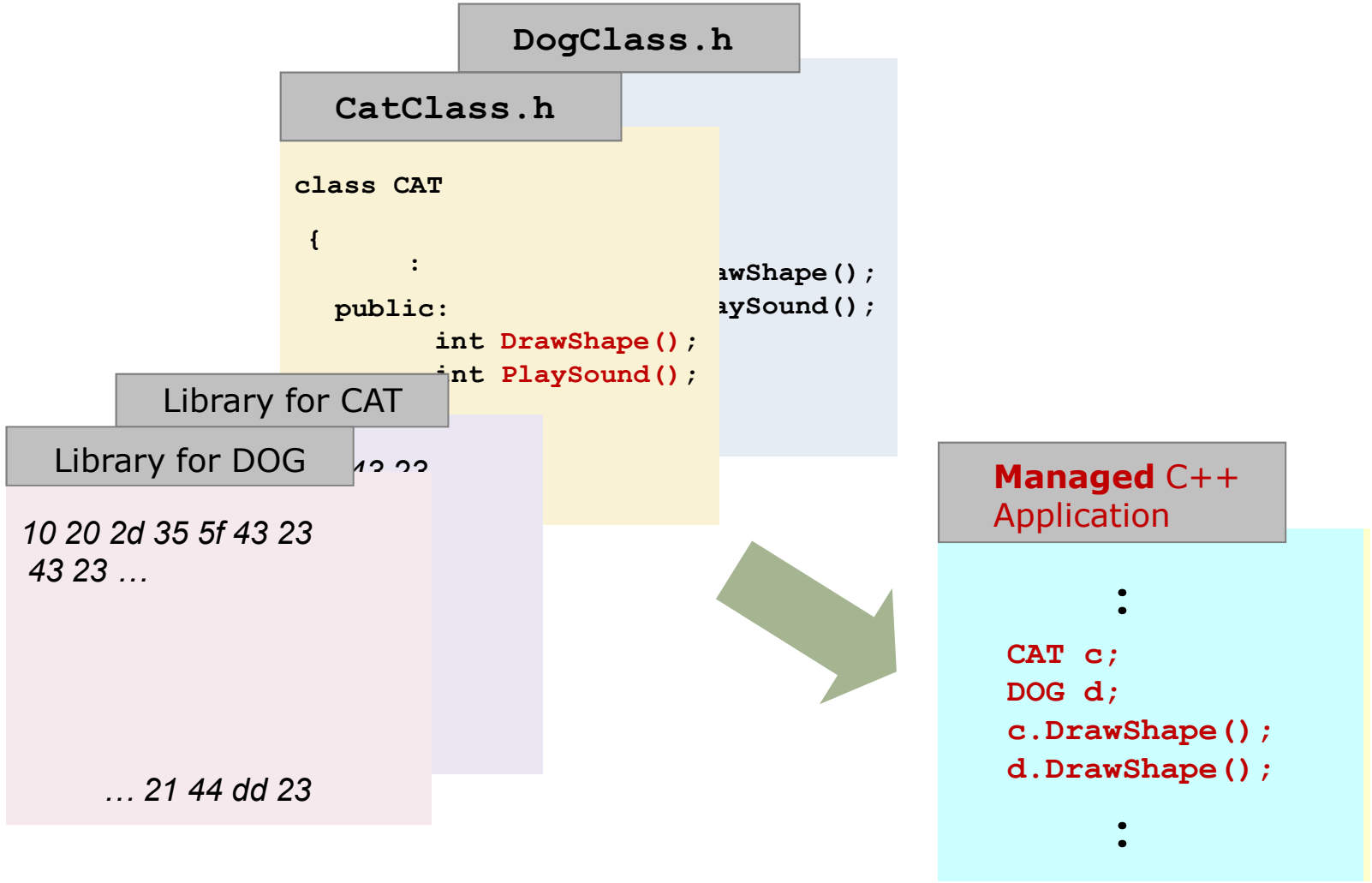
Library for CAT

Library for DOG

10 20 2d 35 5f 43 23
43 23 …

:

… 22 6f
… 21 44 dd 23


DogClass.h
CatClass.h
class CAT
{
:
public:
int DrawShape();
int PlaySound();
awShape();
aySound();
Library for CAT
Library for DOG
10 20 2d 35 5f 43 23
43 23 …
… 21 44 dd 23
Managed C++
Application
:
CAT c;
DOG d;
c.DrawShape();
d.DrawShape();
:

Example: Adding two numbers

Step 1: Create the class required

```
class Arithmetic                    // declare the class
{
public:  void SetNum (int in1, int in2);
         int GetNum1();
         int GetNum2();
         int Add();                 // return num1 + num2
private: int num1;
         int num2;
};
```

Arithmetic.h

Step 2: Implement the class

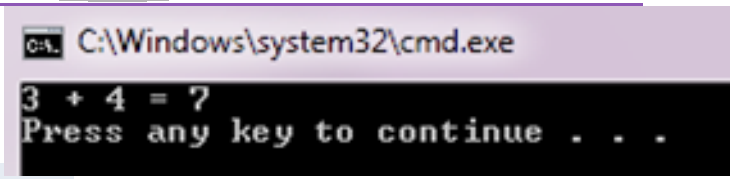
```
void Arithmetic::SetNum(int in1, int in2)
{       num1 = in1; num2 = in2;
}
int Arithmetic::GetNum1()
{       return num1;
}
int Arithmetic::GetNum2()
{       return num2;
}
int Arithmetic::Add()
{       return num1 + num2;
}
```

- For each member function, develop the required codes

Step 3: Test the class

- Create the header file and a **`main()`** for testing the functionality of the class
  - This **`main()`** is only **for testing purpose**, not supposed to be used in real application

```
class Arithmetic                    // declare the class
{
public:  void SetNum (int in1, int in2);
         int GetNum1();                          Arithmetic.h
         int GetNum2();
         int Add();                 // return num1 + num2
private: int num1;
         int num2;
};
```



MainTest.cpp

```
#include <iostream>
using namespace std;
#include "Arithmetic.h"
// Implement the functions and test them
void Arithmetic::SetNum(int in1, int in2)
{       num1 = in1; num2 = in2;
}
int Arithmetic::GetNum1()
{       return num1;
}
int Arithmetic::GetNum2()
{       return num2;
}
int Arithmetic::Add()
{       return num1 + num2;
}
```

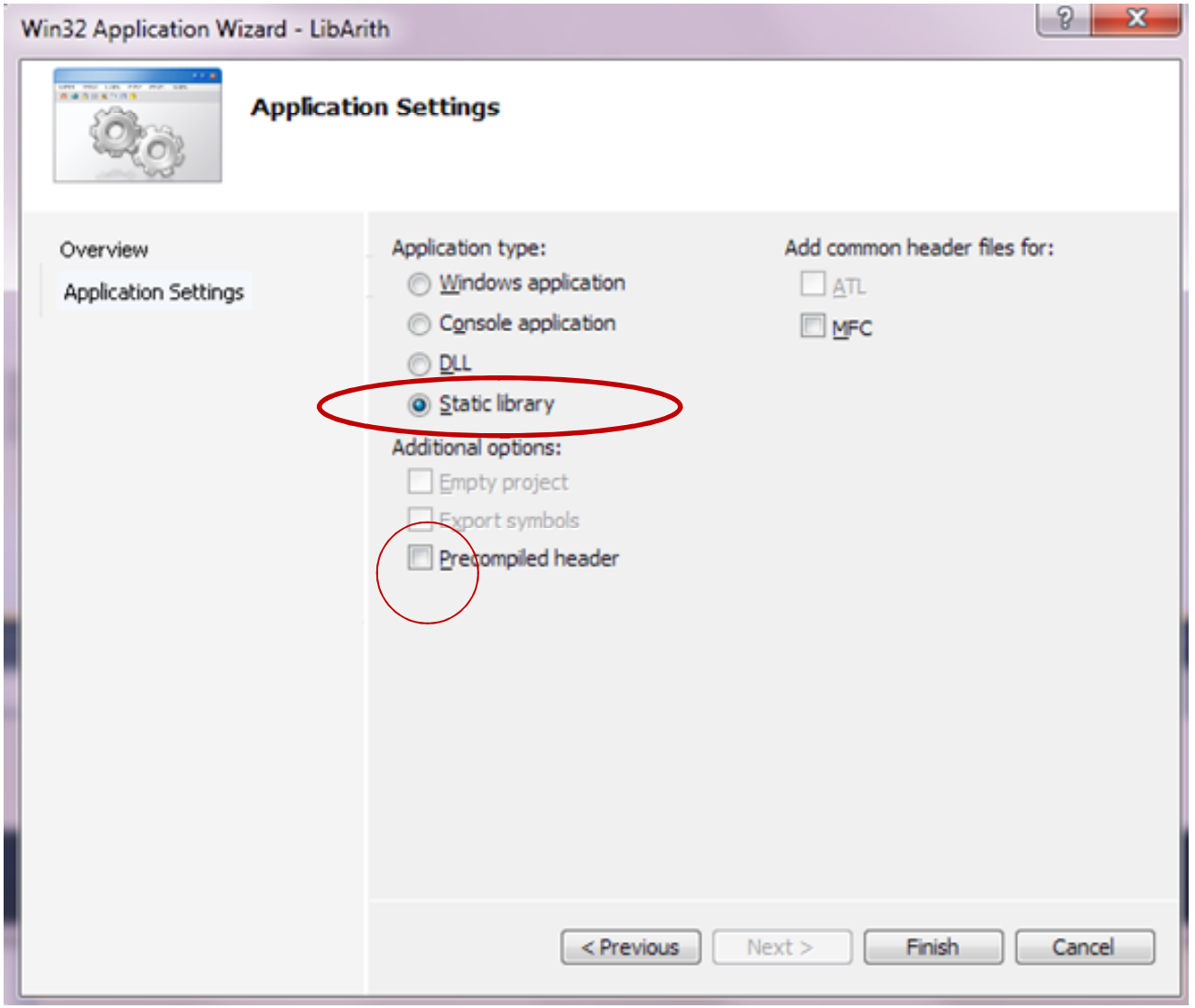
Just a testing console application, not supposed to be used in real application

```
int main()
{   Arithmetic Op;
    Op.SetNum(3,4);
    int Num1 = Op.GetNum1();
    int Num2 = Op.GetNum2();
    cout << Num1 << " + " << Num2
    << " = " << Op.Add() << endl;
    return 0;
}
```

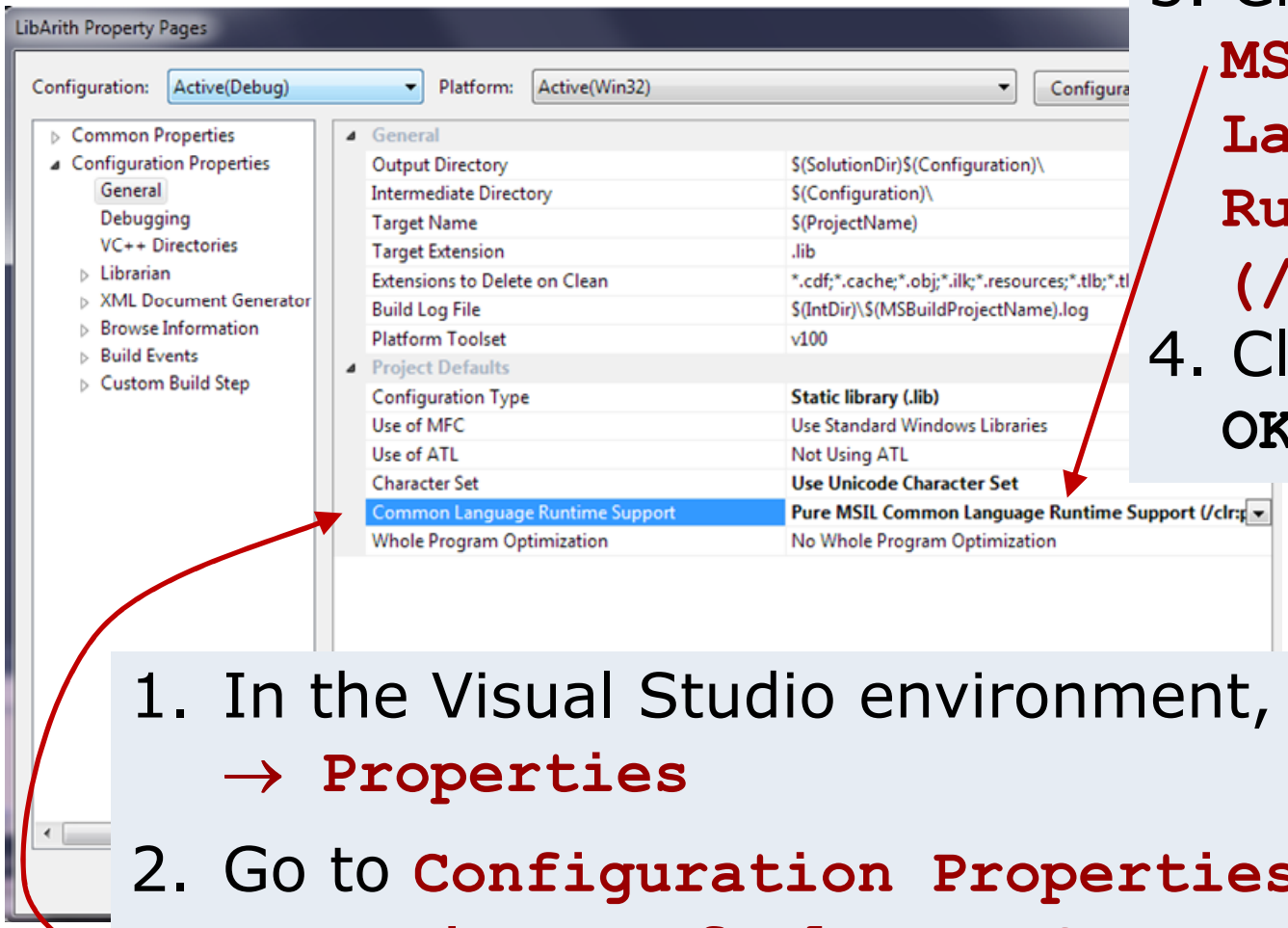
Step 4: Create the library with Visual C++

- Assume your class is fully tested. You can create your library using Visual C++
  - In Visual C++ 2010, start a new Win32 Console Application as usual
  - Set the location to store your **library** project, e.g. **`E:\VS_Proj`**
  - Set the project name, e.g. **`LibArith`**
  - However, on choosing the options in the **`Application Settings`** window, check **`Static library`** and uncheck the option **`Precompiled header`**.

9 Building Graphical User Interface



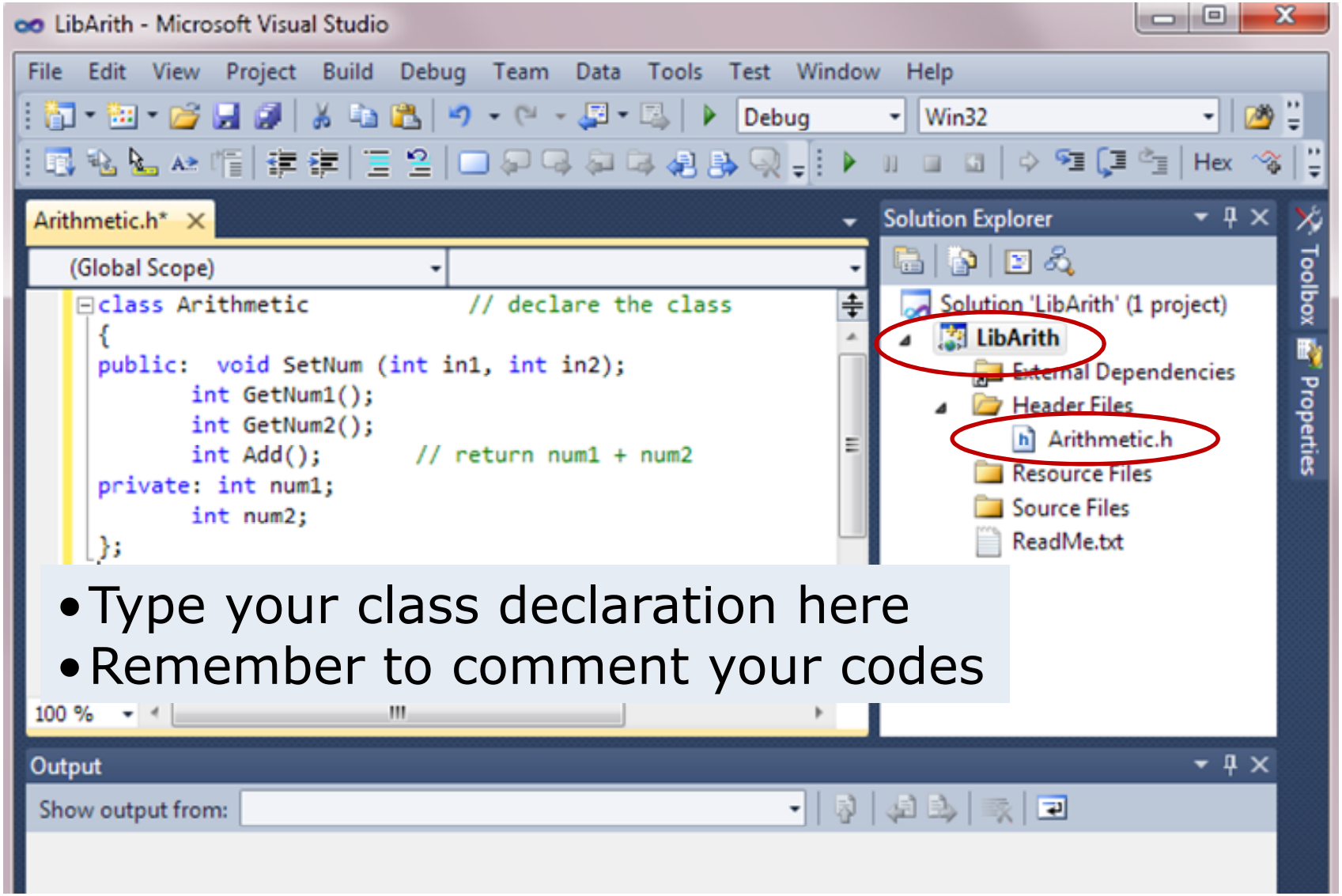
9 Building Graphical User Interface



3. Choose **Pure MSIL Common Language Runtime Support (/clr:pure)**
4. Click **Apply** and **OK**

1. In the Visual Studio environment, click **Project → Properties**
2. Go to **Configuration Properties → General → Project Defaults → Common Language Runtime support**

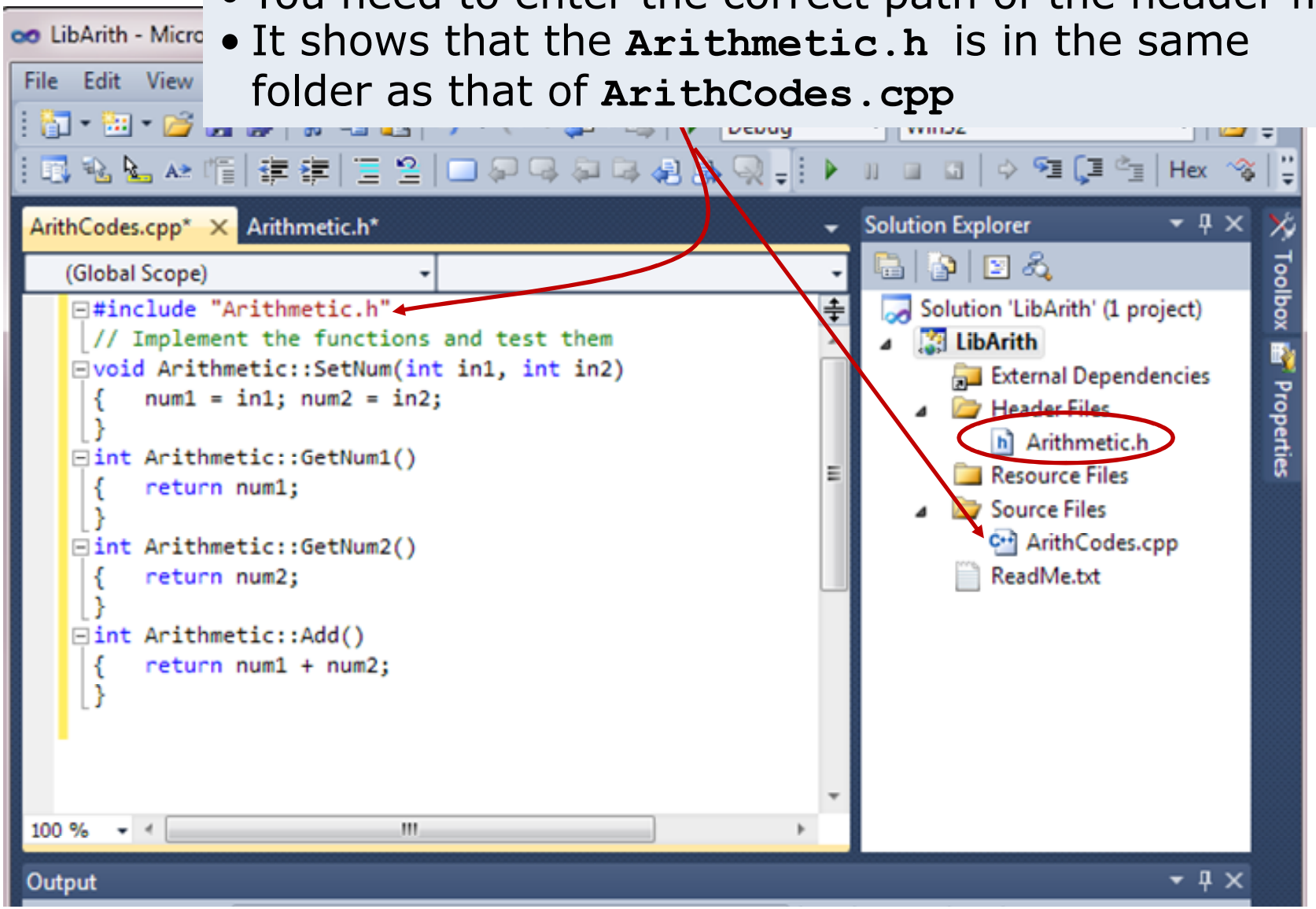
- In the Visual Studio environment, click **`Project →Add New Item...`** and choose **`Header File (.h)`**
- Set the Header File name to, e.g. **`Arithmetic`**
- Click **`Add`** and the header file **`Arithmetic.h`** will be added to the project **`LibArith`**
- Type the class declaration.


LibArith - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Win32
Hex
Arithmetic.h*
(Global Scope)
class Arithmetic // declare the class
{
public: void SetNum (int in1, int in2);
int GetNum1();
int GetNum2();
int Add(); // return num1 + num2
private: int num1;
int num2;
};
Solution Explorer
Solution 'LibArith' (1 project)
LibArith
External Dependencies
Header Files
Arithmetic.h
Resource Files
Source Files
ReadMe.txt
Toolbox
Properties
• Type your class declaration here
• Remember to comment your codes
100 %
Output
Show output from:

- In Visual Studio, click **`Project→Add New Item...`** and choose **`C++ File (.cpp)`**
- Set the C++ source file name, e.g. **`ArithCodes`**
- Click **`Add`** and the C++ source file **`ArithCodes.cpp`** will be added to the project **`LibArith`**
  - You may verify it by checking the **`Solution Explorer`**
- Type the codes for the implementation of the member functions
  - Remember to include your **header file** and note its location
- Build the library by clicking **`Build Solution`**.

9 Bui

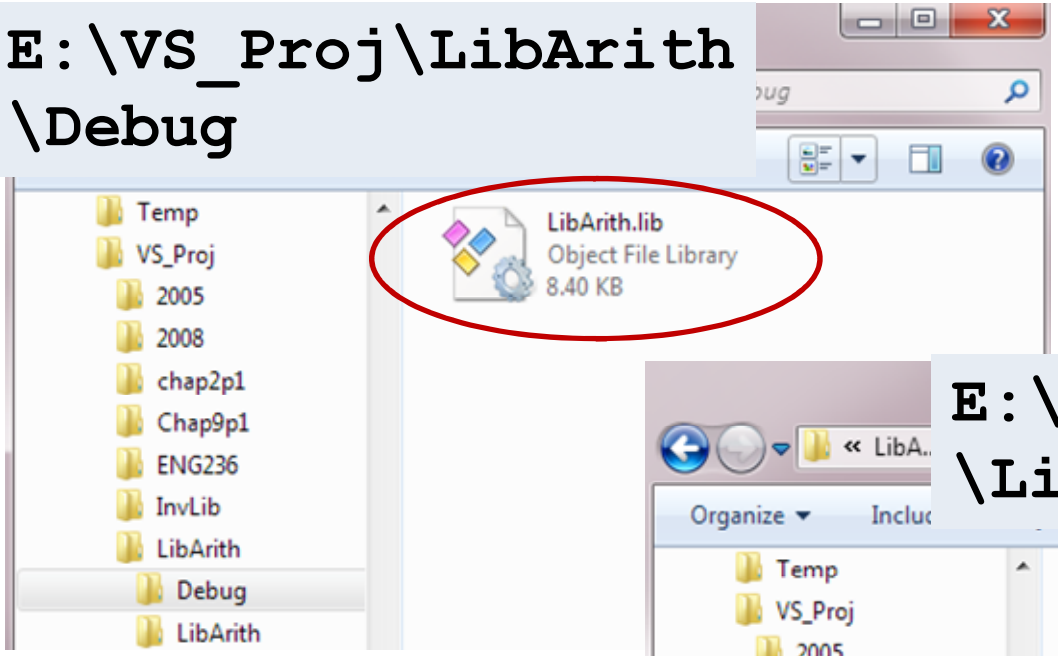
- You need to enter the correct path of the header file
- It shows that the **`Arithmetic.h`** is in the same folder as that of **`ArithCodes.cpp`**



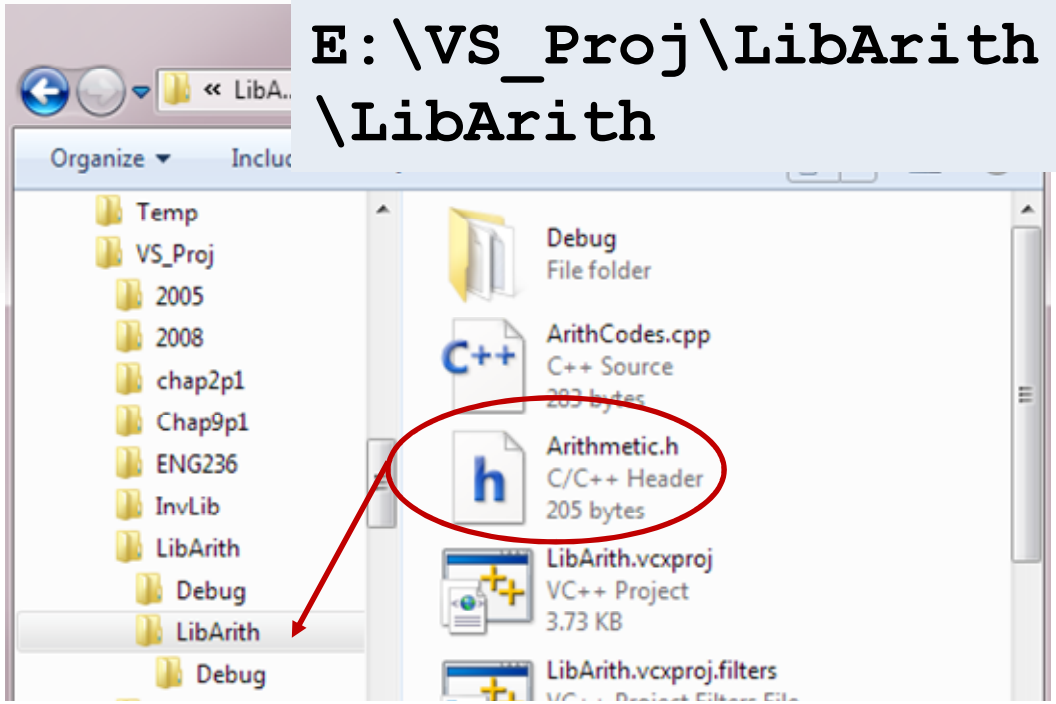
Step 5: Locate the library and header files

- Locate the files **`Arithmetic.h`** and **`LibArith.lib`**
  - **`Arithmetic.h`** shows the class definition
    - From its contents, we know how to use the class
  - **`LibArith.lib`** contains the codes for the implementation of the member functions
    - Note that it is in object code form. No one can see or modify the source code.

E:\VS_Proj\LibArith\Debug

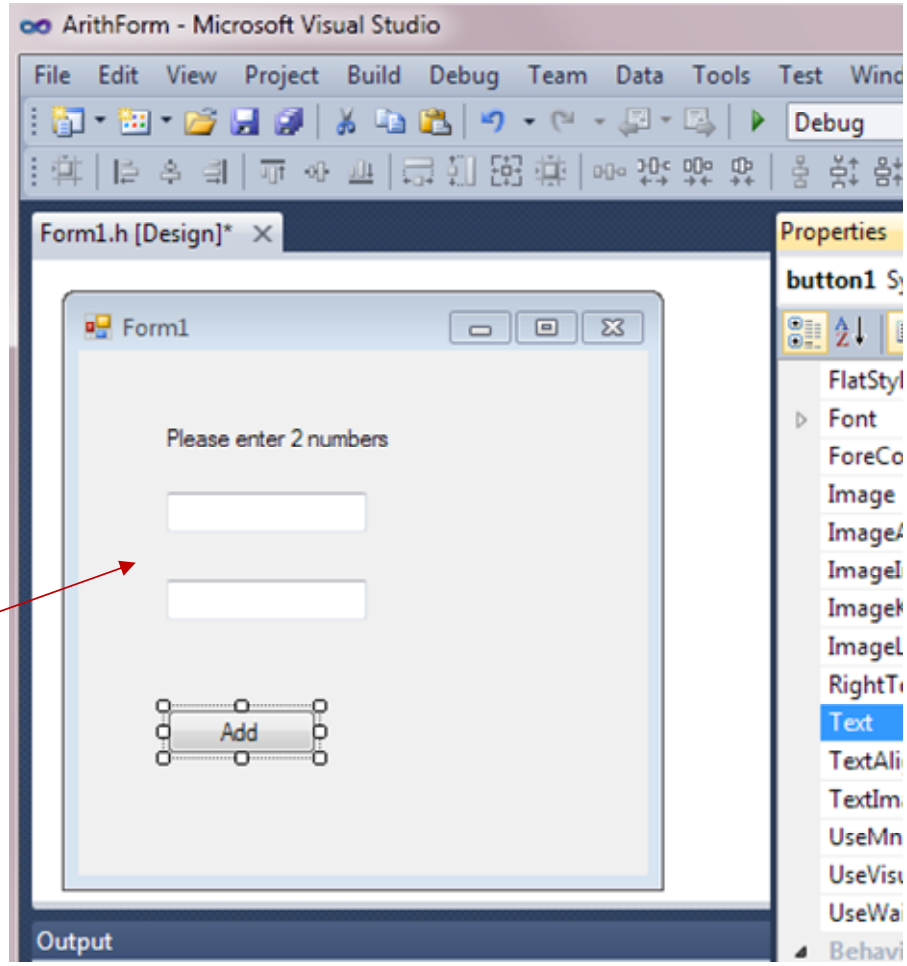


E:\VS_Proj\LibArith\LibArith



Step 6: Integrating into the application

- Open a project of **Windows Forms Application**, e.g. **ArithForm** in the folder e.g. **E:\VS_Proj**
- Design the form as shown:



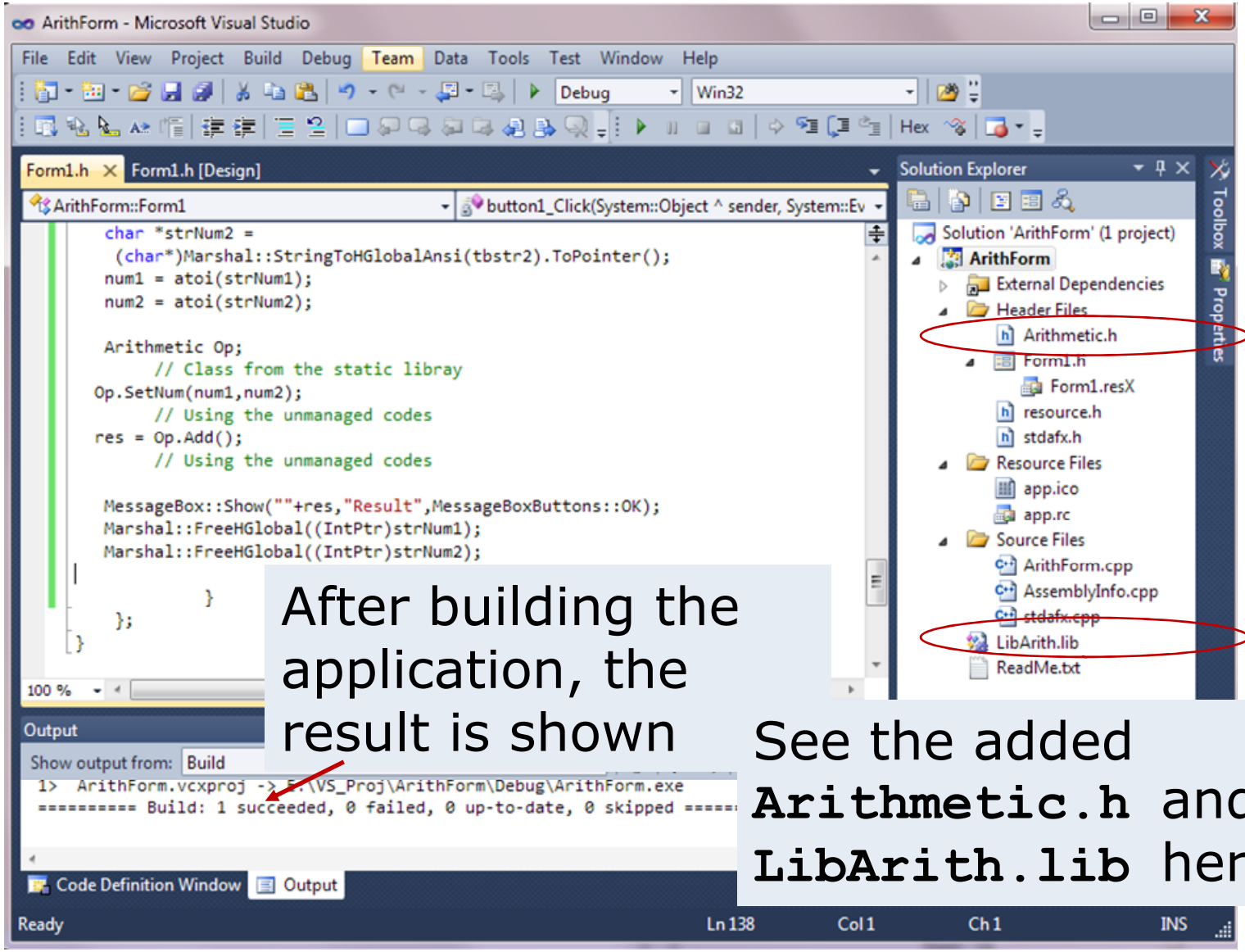
- **Copy** `Arithmetic.h` and `LibArith.lib` to the current folder `E:\VS_Proj\ArithForm\ArithForm`
- In the Visual Studio environment, click `Project→Add Existing Item...` under `ArithForm`. Select `All Files (*.*)` to show all the files in the folder
- Add `Arithmetic.h` and `LibArith.lib` to the project.

- **Double-click** the **Add** button and add the following codes to **button1_Click()**

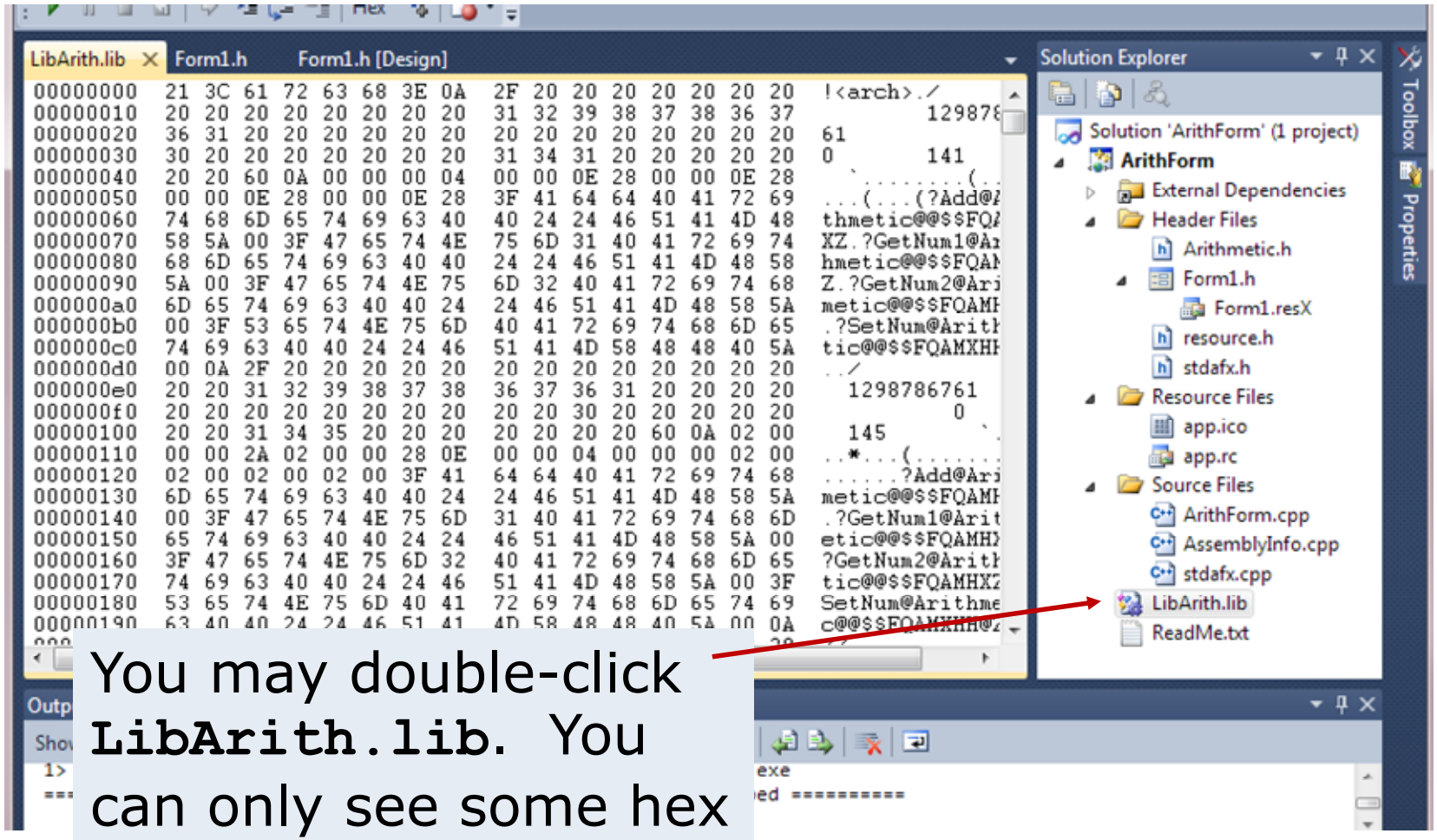
```
#include "Arithmetic.h"
#include <stdlib.h> // For atoi()
// Must be added at the beginning of Form1.h

using namespace System::Runtime::InteropServices;

private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
{  int num1, num2, res;  // variables used by unmanaged code
   String ^ tbstr1 = textBox1->Text;
   String ^ tbstr2 = textBox2->Text;
   char *strNum1 =
    (char*)Marshal::StringToHGlobalAnsi(tbstr1).ToPointer();
   char *strNum2 =
    (char*)Marshal::StringToHGlobalAnsi(tbstr2).ToPointer();
```

```
    num1 = atoi(strNum1);
    num2 = atoi(strNum2);

    Arithmetic Op;
            // Class from the static libray
    Op.SetNum(num1,num2);
            // Using the unmanaged codes
    res = Op.Add();
            // Using the unmanaged codes

    MessageBox::Show(""+res,"Result",MessageBoxButtons::OK);
    Marshal::FreeHGlobal((IntPtr)strNum1);
    Marshal::FreeHGlobal((IntPtr)strNum2);
}
```

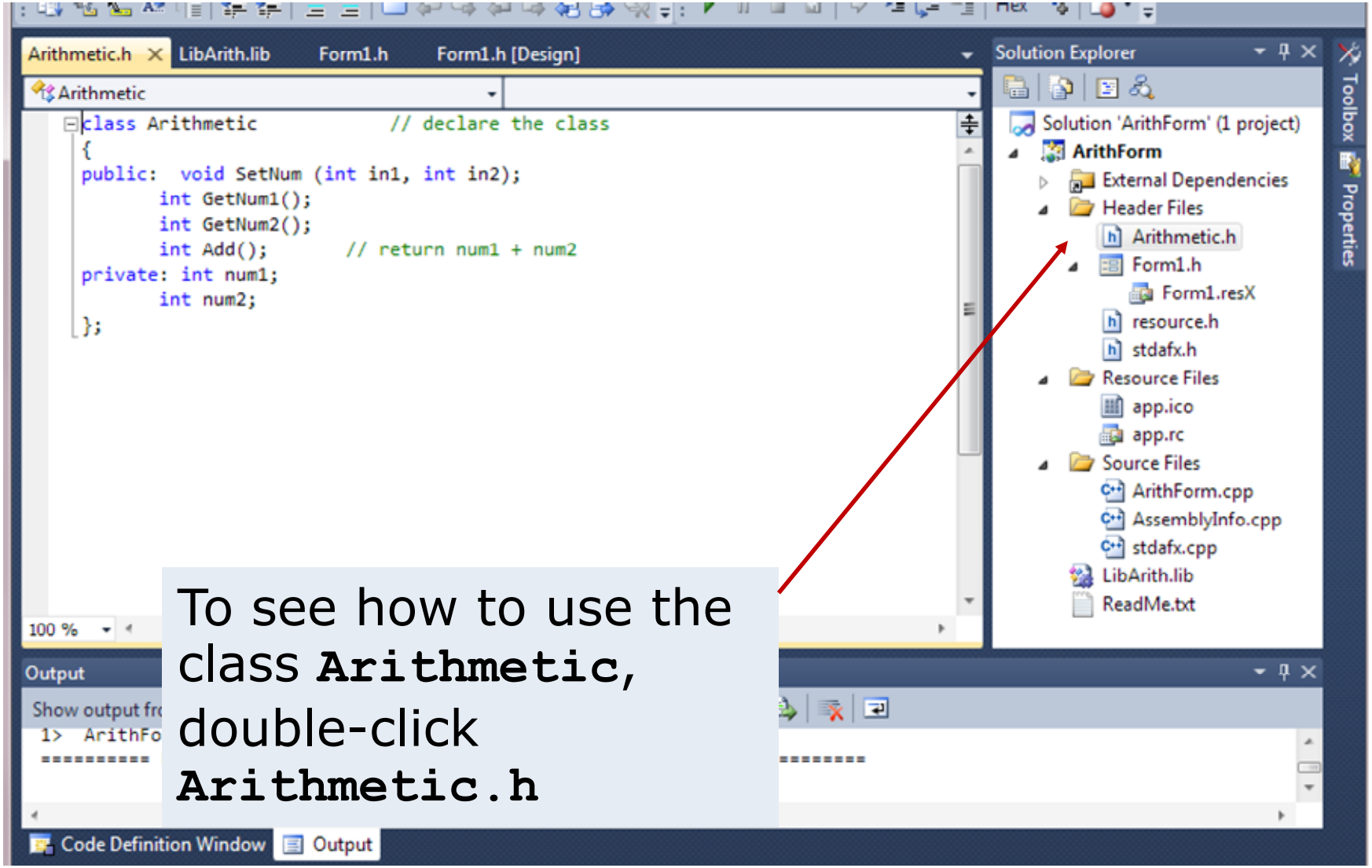
9 Building Graphical User Interface



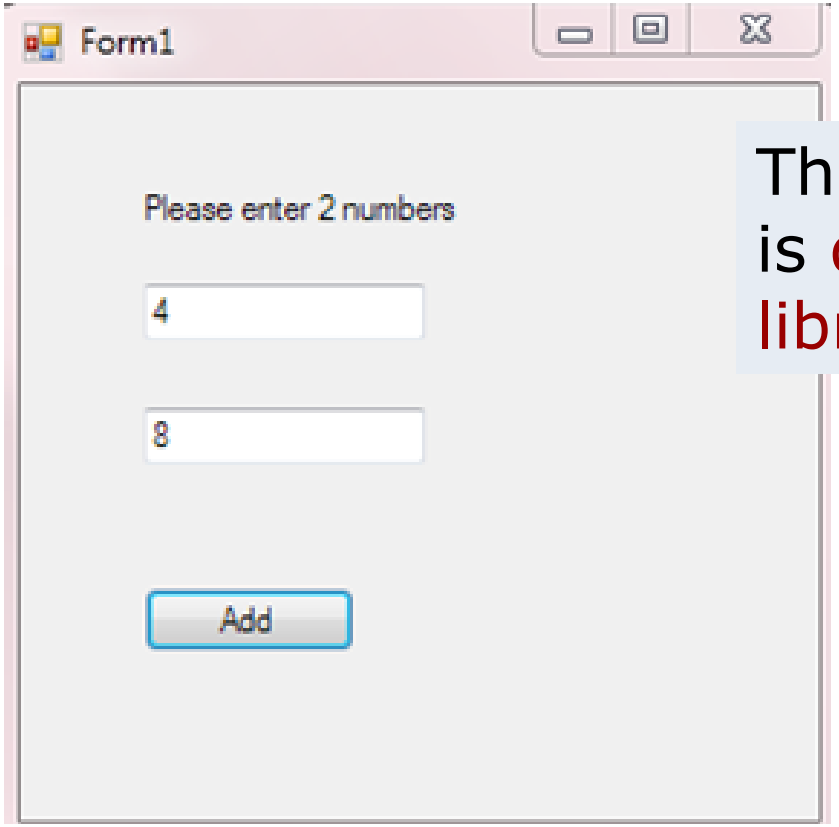
9 Building Graphical User Interface



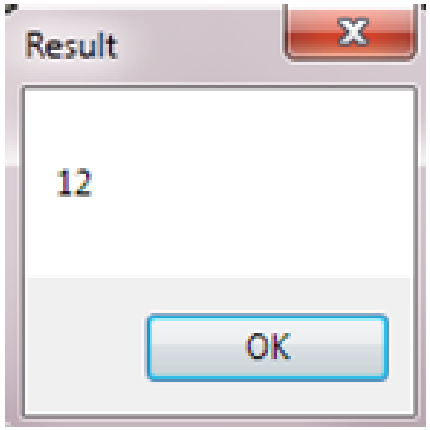
You may double-click **`LibArith.lib`**. You can only see some hex codes. No source codes can be found


Arithmetic.h
LibArith.lib
Form1.h
Form1.h [Design]
Arithmetic
class Arithmetic // declare the class
{
public: void SetNum (int in1, int in2);
int GetNum1();
int GetNum2();
int Add(); // return num1 + num2
private: int num1;
int num2;
};
100 %
Solution Explorer
Solution 'ArithForm' (1 project)
ArithForm
External Dependencies
Header Files
Arithmetic.h
Form1.h
Form1.resX
resource.h
stdafx.h
Resource Files
app.ico
app.rc
Source Files
ArithForm.cpp
AssemblyInfo.cpp
stdafx.cpp
LibArith.lib
ReadMe.txt
Toolbox
Properties
Output
Show output fr
1> ArithFo
Code Definition Window
Output
To see how to use the class Arithmetic, double-click Arithmetic.h

Step 7: Execute the application



The add operation is done by the library function



Exercise 9.2

Modify your static library such that if the inputs are x and y, the member functions for finding x^y and x/y are implemented; all of them in integer arithmetic.

Your GUI should include 2 buttons, namely "**`Power`**" and "**`Division`**".

On clicking the "**`Power`**" button, detect if y is negative. On clicking the "**`Division`**" button, detect if y is 0. In both cases, do not do the operation but show an error message box; otherwise, the corresponding function of the static library will be called and the result will be shown in a message box.

9.3 Creating a Multiple-Form Interface

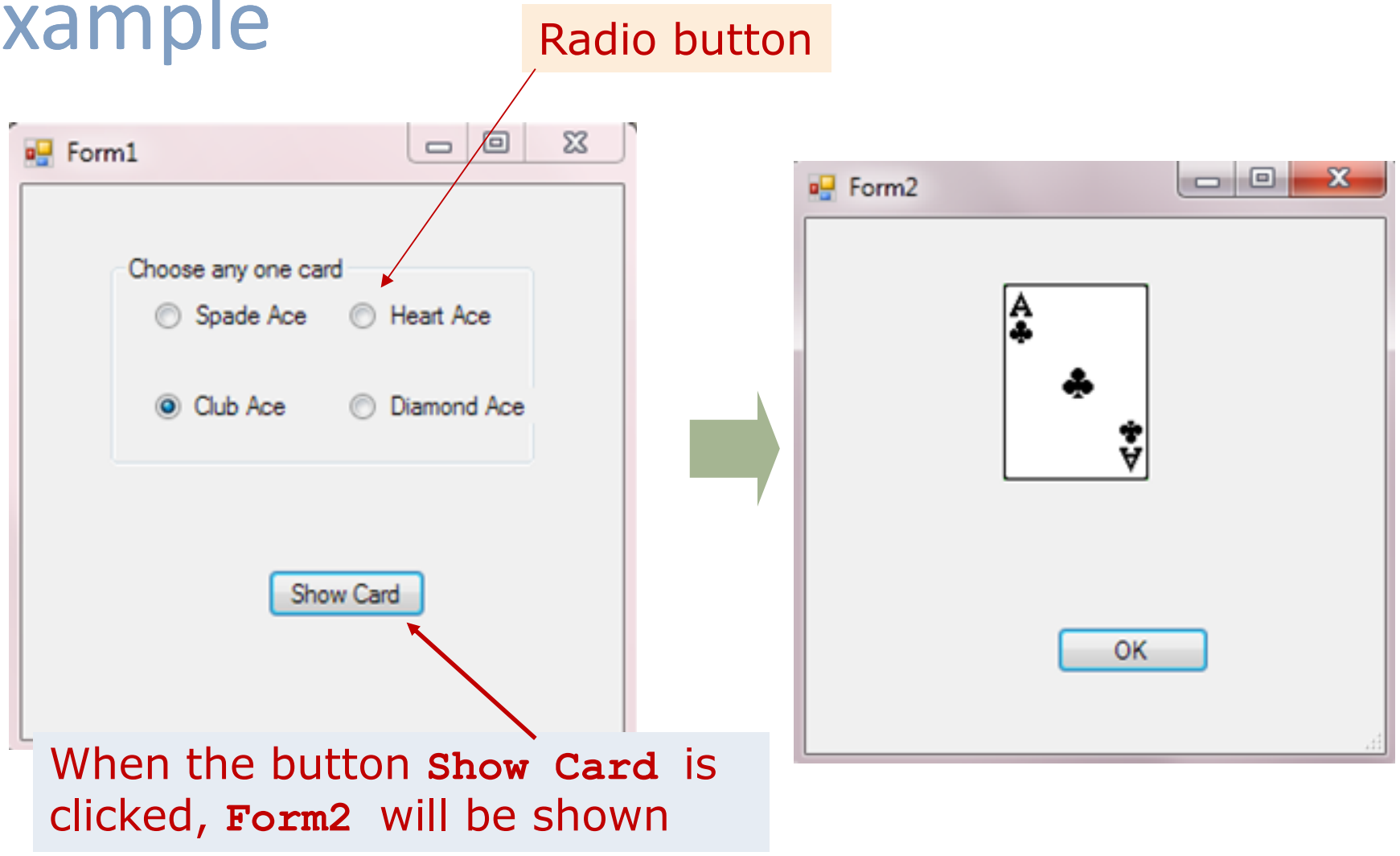
Creating a Multiple-Form Interface

- It is common that an application may have more than one form
  - In addition, one form may be invoked by another form

Each form has its own GUI

- To add one more form to a project, we need to follow the steps below:
  1. Add one more form class to your project
  2. Design the form layout
  3. Set the properties of the GUIs in the new form
  4. Update the program code of the original form.

Example


Radio button
Form1
Choose any one card
Spade Ace
Heart Ace
Club Ace
Diamond Ace
Show Card
When the button Show Card is clicked, Form2 will be shown
Form2
OK

Step 1: Specification of the program

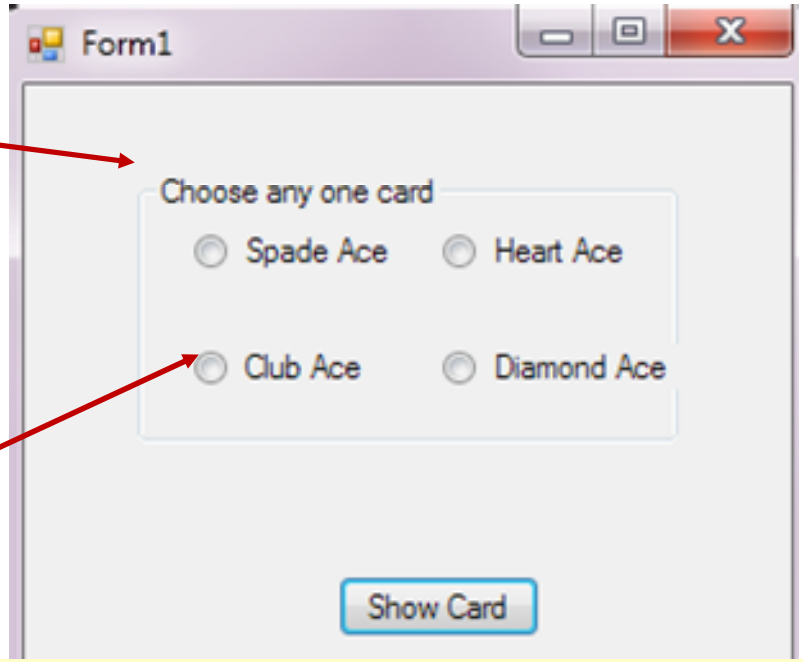
1. A user is asked to select one and only one of the four cards
   - If the user initially does not check any one of the radio buttons, no card will be shown
2. If the user clicks the **`Show Card`** button, another form (**`Form2`**) will be shown with the selected card
3. The user cannot access the original form (**`Form1`**) until he clicks the **`OK`** button of **`Form2`**. **`Form2`** is known as the **modal form**.
4. If the user clicks on the picture, the card will **toggle** between face-up and face-down.

Project name: **Chap9p3**

Step 2: Develop Form1

GroupBox
All controls inside a group box will be grouped together

RadioButton
For the radio buttons inside a group box, only **one** of them can be selected



Make sure the **TabStop** property of the radio buttons (under **Behavior**) are **False**
TabStop enables the user to give the focus to this control using the TAB key

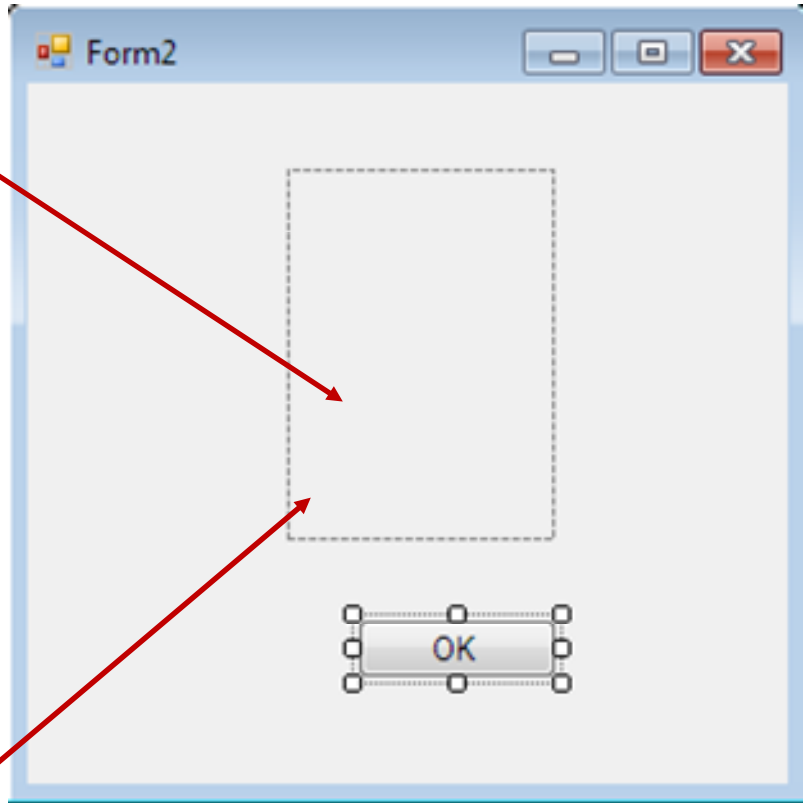
Step 3: Develop `Form2`

- In the **`Solution Explorer`**, right-click your project name and select **`Add → New Item...`**
- Select **`Windows Form`**. Give a name, say **`Form2`**, to your new form (**`Form2`** will be the class name of the form)
- You will see another set of files added to your project, e.g. **`Form2.cpp`**, **`Form2.h`**.

PictureBox

allows you to put a picture into it.

The picture can be obtained from a file



We can add the control we want to here

View → Code

Base class

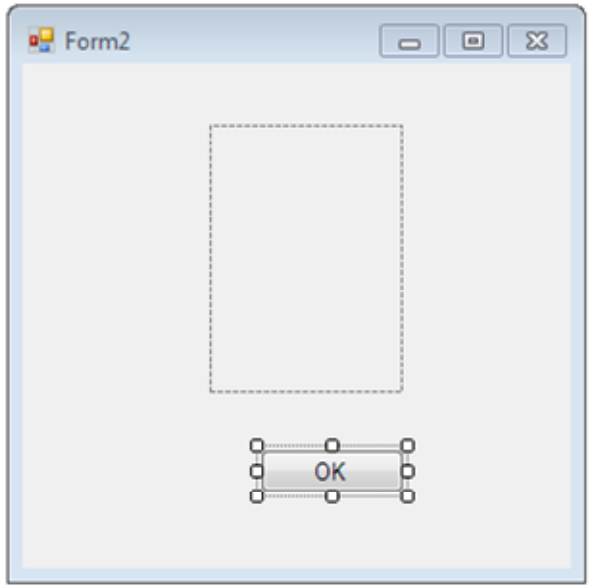
Derived class

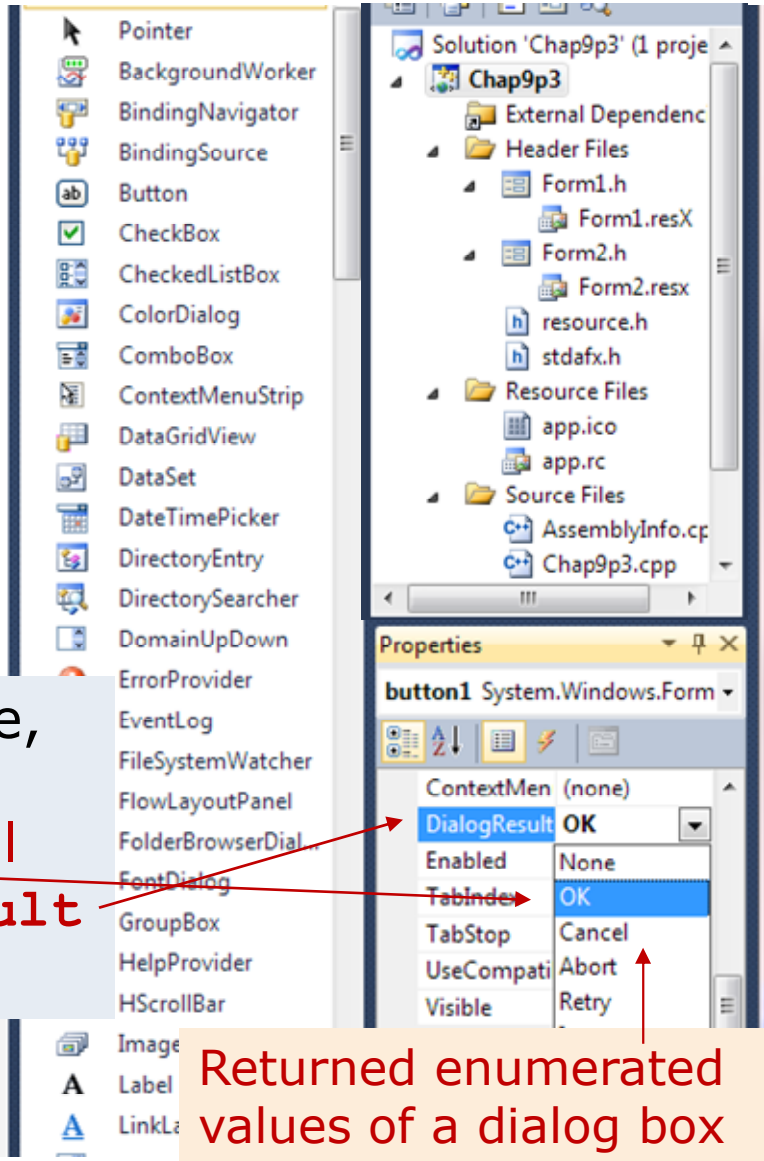
```
namespace chap9p3 { // define the namespace of the class
   :
  public ref class Form2 : public System::Windows::Forms::Form
  {    // The class name is Form2.
 public:
       Form2(void)   // The constructor. Define the things
       {      :      // that will be done when the class
       }             // is instantiated. TO BE WRITTEN LATER.
             :
  private: System::Windows::Forms::PictureBox^  pictureBox1;
  private: System::Windows::Forms::Button^  button1;
             :      // The Button object is defined above
  private:
       :
       void InitializeComponent(void)
       {      :      // Record the properties of the controls
       }
  };
}
```

p. 72

Form2.h is used mainly for defining the class structure of this form

9 Building Graphical User Interface





Rather than writing many lines of code, we can simply define the function of a button as **`OK`** (i.e. the **modal** form will close if it is clicked) in the **`DialogResult`** of the **`Properties`** window

Returned enumerated values of a dialog box

double-click **Show Card**

Step 4: Implement the handler of Form1

```
#include "Form2.h" // Add to the beginning of Form1.h

private: System::Void button1_Click(System::Object ^  sender,
System::EventArgs ^  e)
{      // Called when button1 of Form1 is clicked
  int card = 0;      // 0 = no card, 1 = sA, 2 = hA
                     // 3 = cA, 4 = dA
  if (radioButton1->Checked)
       card = 1;   // spade ace selected
  if (radioButton2->Checked)
       card = 2;     // heart ace selected
  if (radioButton3->Checked)
       card = 3;     // club ace selected
  if (radioButton4->Checked)
       card = 4;     // diamond ace selected
  Form2 ^ F2 = gcnew Form2(card); //Form2 defined in Form2.h
  F2->ShowDialog(); // Show modal form, return DialogResult value
}
```

An object of **Form2** is created in the CLR heap pointed by **F2**
Use **gcnew** because **Form2** is a reference type under Common Type System (CTS)

Since we want to tell **Form2** which card a user has selected, the constructor of **Form2** has to be modified to accept one input parameter that indicates the selected card

```
Form2 ^ F2 = gcnew Form2(card);
F2->ShowDialog();
```

To show **Form2**, we need to call the member function **ShowDialog()** of the class **Form2**

- If **ShowDialog()** is called, **Form2** is a **modal form**
  - we cannot access other forms unless we close **Form2**
- If **Show()** is called instead, **Form2** is a modeless form
  - we can access other forms without closing **Form2**.

In `Form2.h`

Step 5: Implement the constructor and other functions of `Form2`

```
Form2(int c)// The constructor is modified by adding
{           // an input parameter
  InitializeComponent();
  cd_no = c;  // 0 = no card, 1 = sA, 2 = hA
              // 3 = cA, 4 = dA
  face = 0; // By default, the card is face up (0).
              // When the card is clicked, the
              // state will toggle from face-up to
              // face-down or vice versa
  ShowCard(); // Show card based on the value of cd_no
}
// Two private variables are added
private:
  int cd_no;// To keep the info of the card selected
  int face; // See the comment above
```

p. 68

Add a member function inside the `Form2` class

```
void ShowCard()
{   if (cd_no == 0)
      pictureBox1->Visible = false;
    if (cd_no == 1)
    { pictureBox1->Visible = true;
      pictureBox1->Image = Image::
      FromFile("c:\\temp\\cards\\s1.gif");
    }
// codes for other cards follow
```

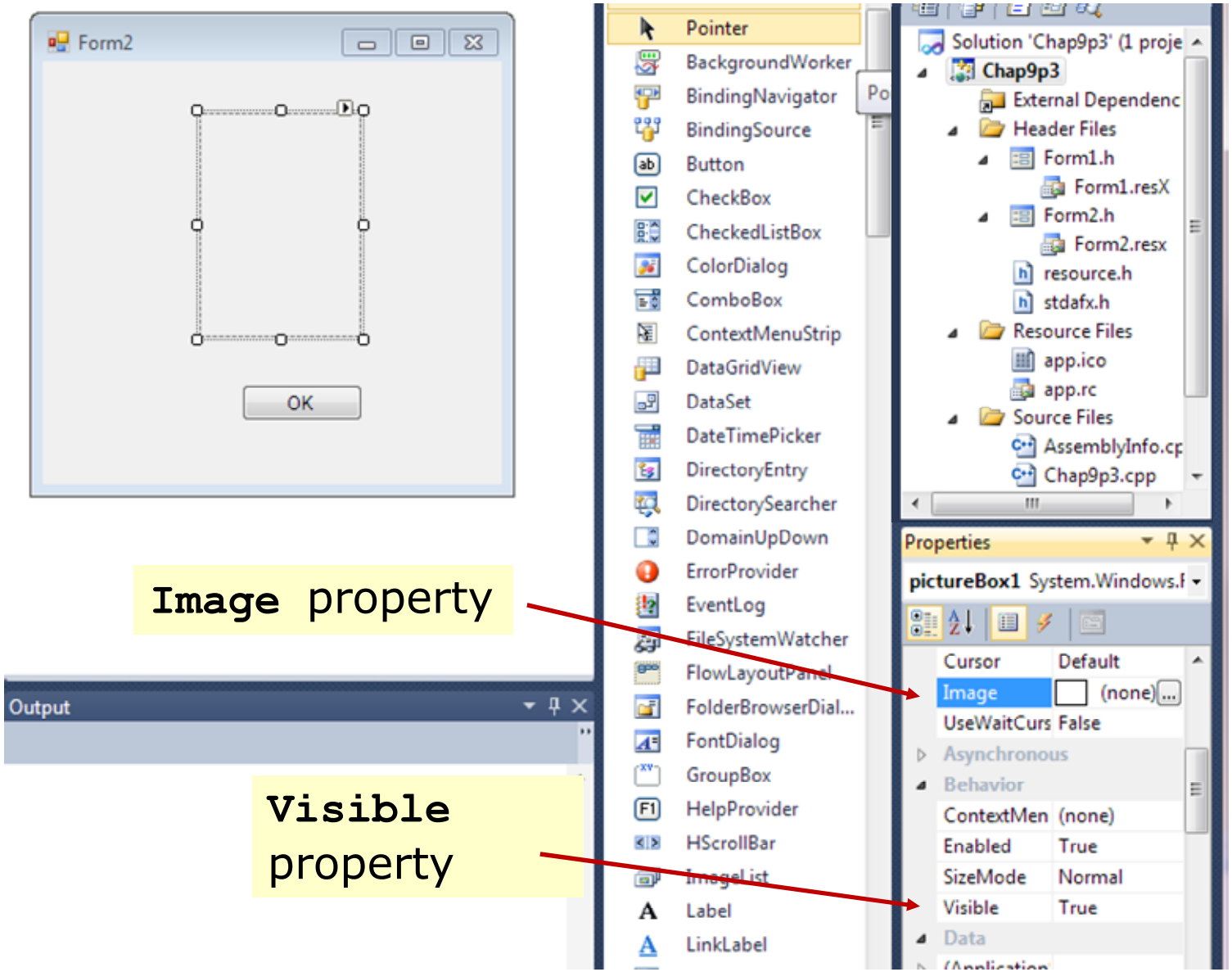
`Visible` is a property of `PictureBox`. If it is `false`, the `PictureBox` object will not be visible.

The file for Spade Ace

`FromFile` is a static member function of the `Image` class. It returns a file handler of an image stored at the specified location

`Image` is another property of `PictureBox`. It defines the location where the picture can be found

9 Building Graphical User Interface



```
    if (cd_no == 2)
    {   pictureBox1->Visible = true;
        pictureBox1->Image =
                Image::FromFile("c:\\temp\\cards\\h1.gif");
    }
    if (cd_no == 3)
    {   pictureBox1->Visible = true;
        pictureBox1->Image =
                Image::FromFile("c:\\temp\\cards\\c1.gif");
    }
    if (cd_no == 4)
    {   pictureBox1->Visible = true;
        pictureBox1->Image =
                Image::FromFile("c:\\temp\\cards\\d1.gif");
    }
}
```

The file for Heart Ace

The file for Club Ace

The file for Diamond Ace

\ is a special character. If we want to use it as just the \ character, we have to use the escape sequence \\.

Double-click **pictureBox1**

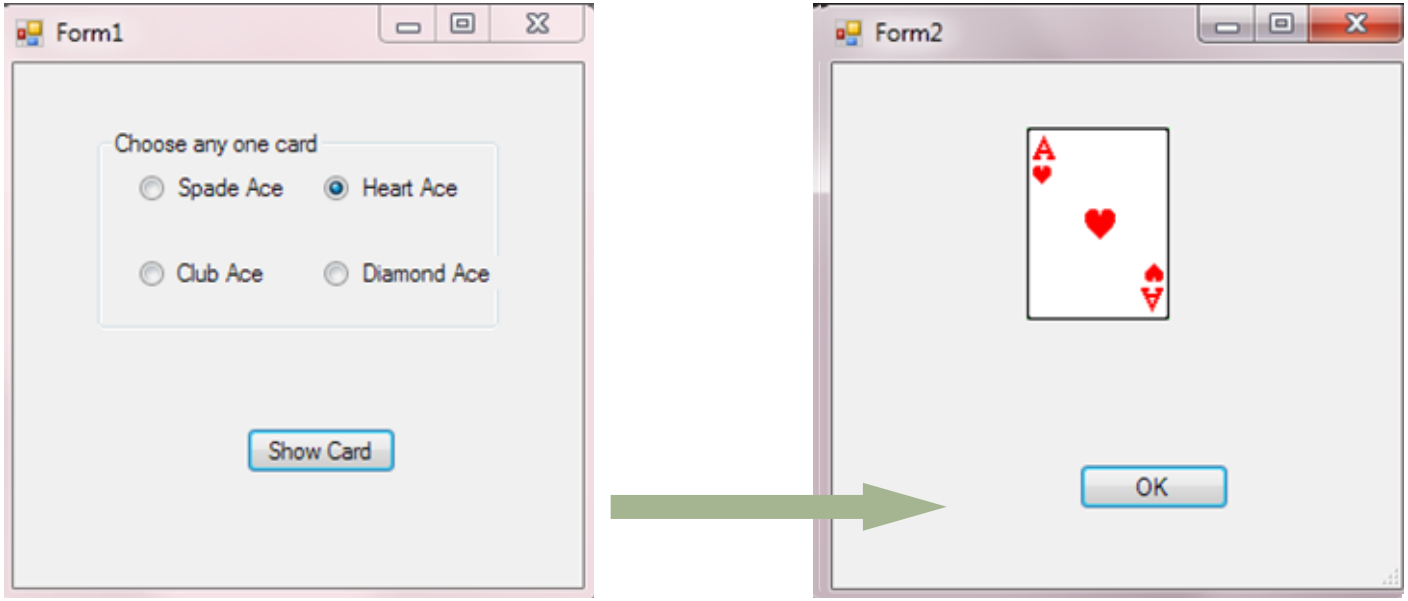
```
// Define the thing to be done when pictureBox1 is
clicked
private: System::Void pictureBox1_Click(System::Object
^  sender, System::EventArgs ^  e)
{
   if (cd_no != 0) //  A card is selected
   {  if (face == 0)  // face = 0 means face-up
      {  pictureBox1->Image = Image::
            FromFile("c:\\temp\\cards\\b1fv.gif");
            // Show the card as facing down, actually a
            // different file is shown
         face = 1; // Currently face-down
      } // codes for face==1 follow
```

If user clicks on the **PictureBox**, it will change from face-up to face-down or vice versa

It is controlled by the member variable "**face**"

```
        else          // It is face-down
        {   ShowCard();
            face = 0; // currently face-up
        }
    }
}
```

Step 6: Build and run the project



Exercise 9.3

Follow the above steps to build the project.
Change the **`Form2`** object to a modeless form.
Add an event-handler for the **`OK`** button in **`Form2`** such that on clicking the button, the picture-box will randomly display one of the four cards, irrespective of the selection in **`Form1`**.

Hint: We can generate a random number by using the function **`int rand(void)`**, which returns a random integer between 0 to 0x7FFF. Remember to include **`stdlib.h`** before you use this function.

Use **`Help`** of **`Visual Studio`**

Remark: On using **`rand()`**, the random sequence is repeatable on every execution of the program. If we don't want it to be repeatable, what should we do? (Hint: learn **`srand(unsigned int)`**)

9.4 A Brief Introduction to GDI+

S.R.G. Fraser, *Pro Visual C++/CLI and the .NET 3.5 Platform. Berkeley, CA:* Apress, 2009 - Ch. 12. [Electronic Resources]

http://msdn.microsoft.com/en-us/library/ms533798

What is GDI+?

`System::Drawing`

- Using the .NET framework class library's Windows Form controls is NOT the only way to graphically present data
  - Windows Form controls sometimes are too restrictive that make the displaying of graphic data difficult, e.g.
    - Drawing a line on the screen
    - Putting a circle on a form
    - Showing a sequence of images as an **animation**, etc.
  - Graphics can be put into a program only if a related control is provided, hence it restricts creativity.

API vs GUI

- To programmers, GDI+ is just a set of namespaces that provides library functions for the rendering of 2D graphics An API
  - It provides support for colors, pens, fonts, image transformations, etc.
    - However, it does not have advanced animation and 3D rendering features found in DirectX
  - GDI+ is originated from the Windows Graphics Device Interface (GDI) with several improvements
    - GDI+ is device independent – output of the graphics can be shown on screens and printers.

Programmer makes calls to methods provided by GDI+ classes and those methods in turn make the appropriate calls to specific device drivers.

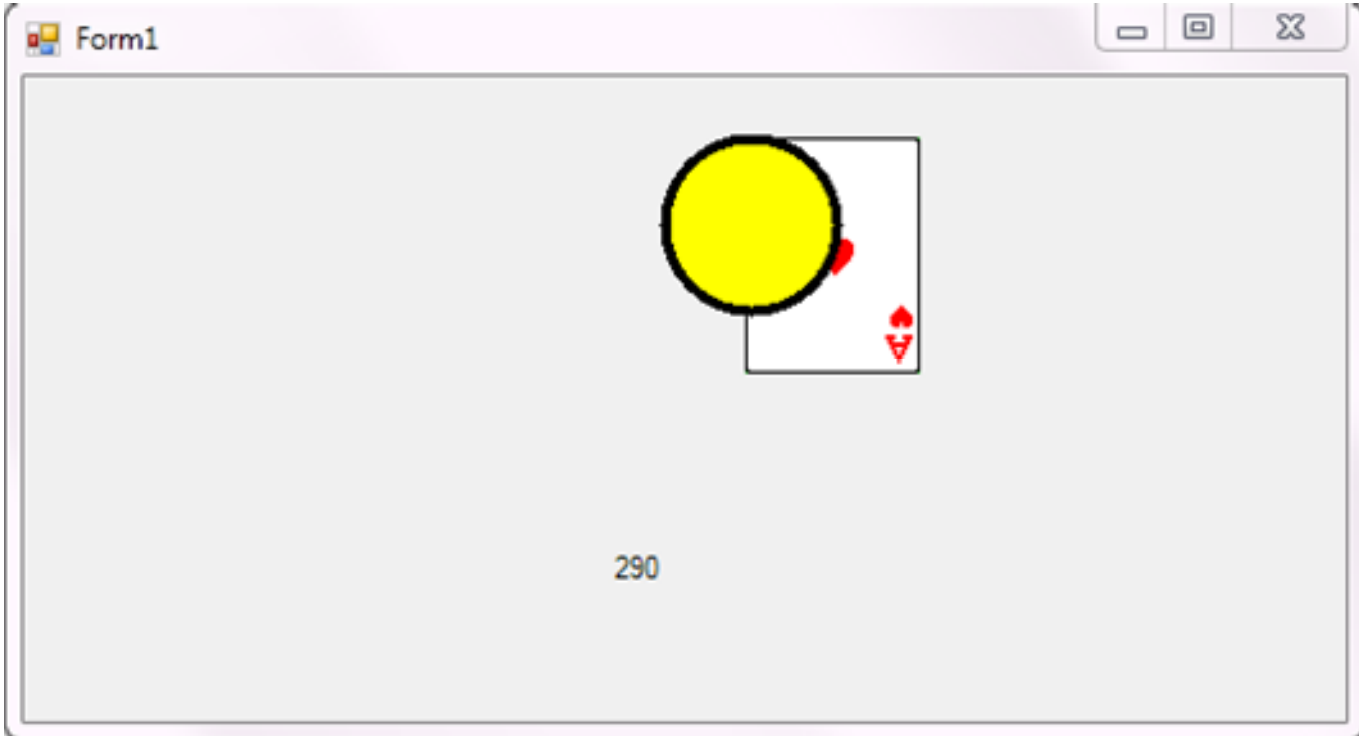
- When starting a Windows Form project in Visual C++ 2010, GDI+ has been included by default
  - One can verify this by checking if the Form class contains the following statement
    - `using namespace System::Drawing;`
    - and if the **`References`** part of the solution contain
    - `System.Drawing`

Right-click project name and select **`References...`**

  - For detailed discussion of GDI+, one can refer to
    - Microsoft MSDN home page:

      `http://msdn.microsoft.com/en-us/library/ms533798`

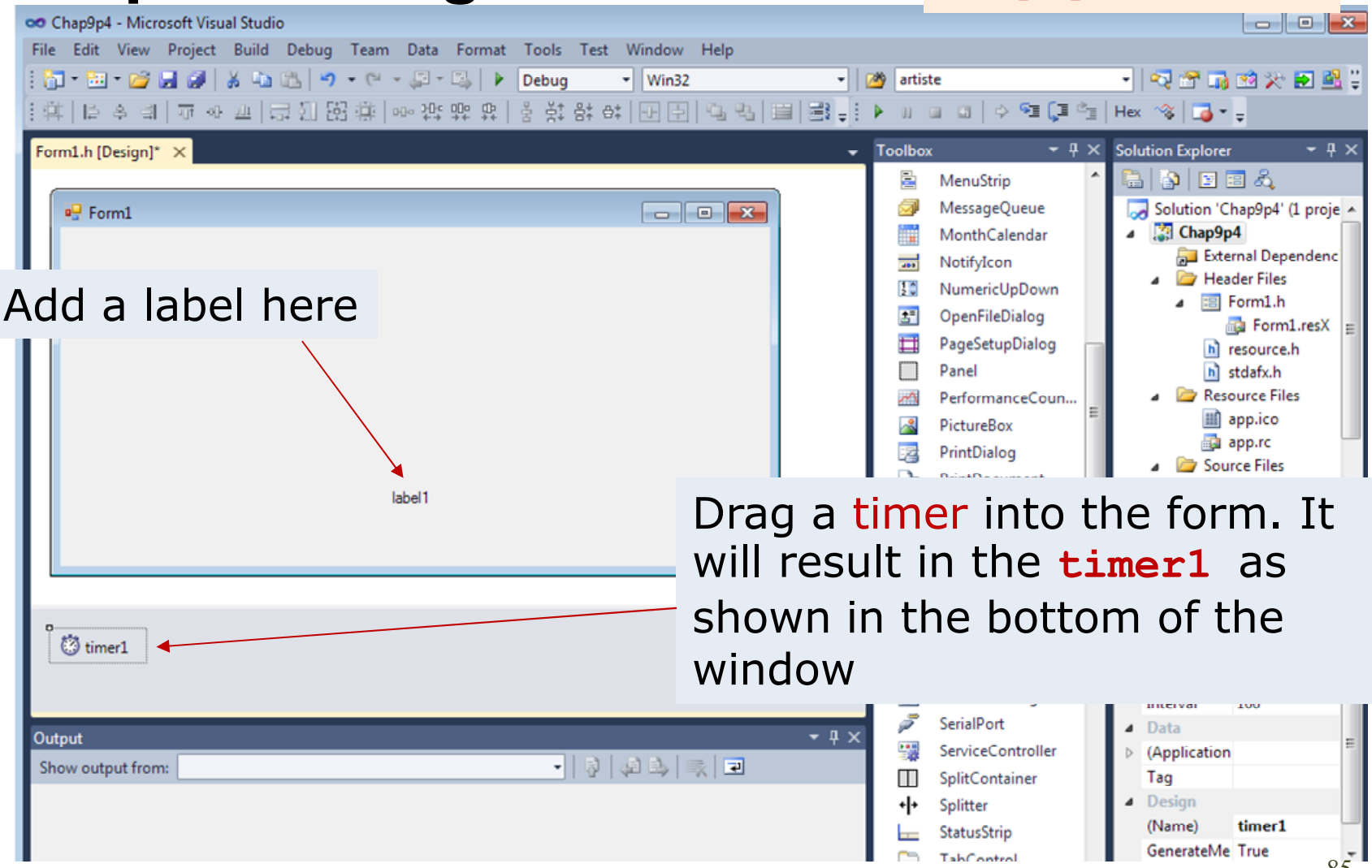
Example: Moving Graphic and Picture



- Requirements:
  1. The card will fly from left to right and the circle (graphic) will fly from right to left
  2. A counter is put at the bottom to show the current horizontal position of the card
  3. When clicking on the counter, the motion of both the circle and the card will stop. When clicking on the counter again, the motion restarts.

Step 1: Design the Form

Project name:
Chap9p4

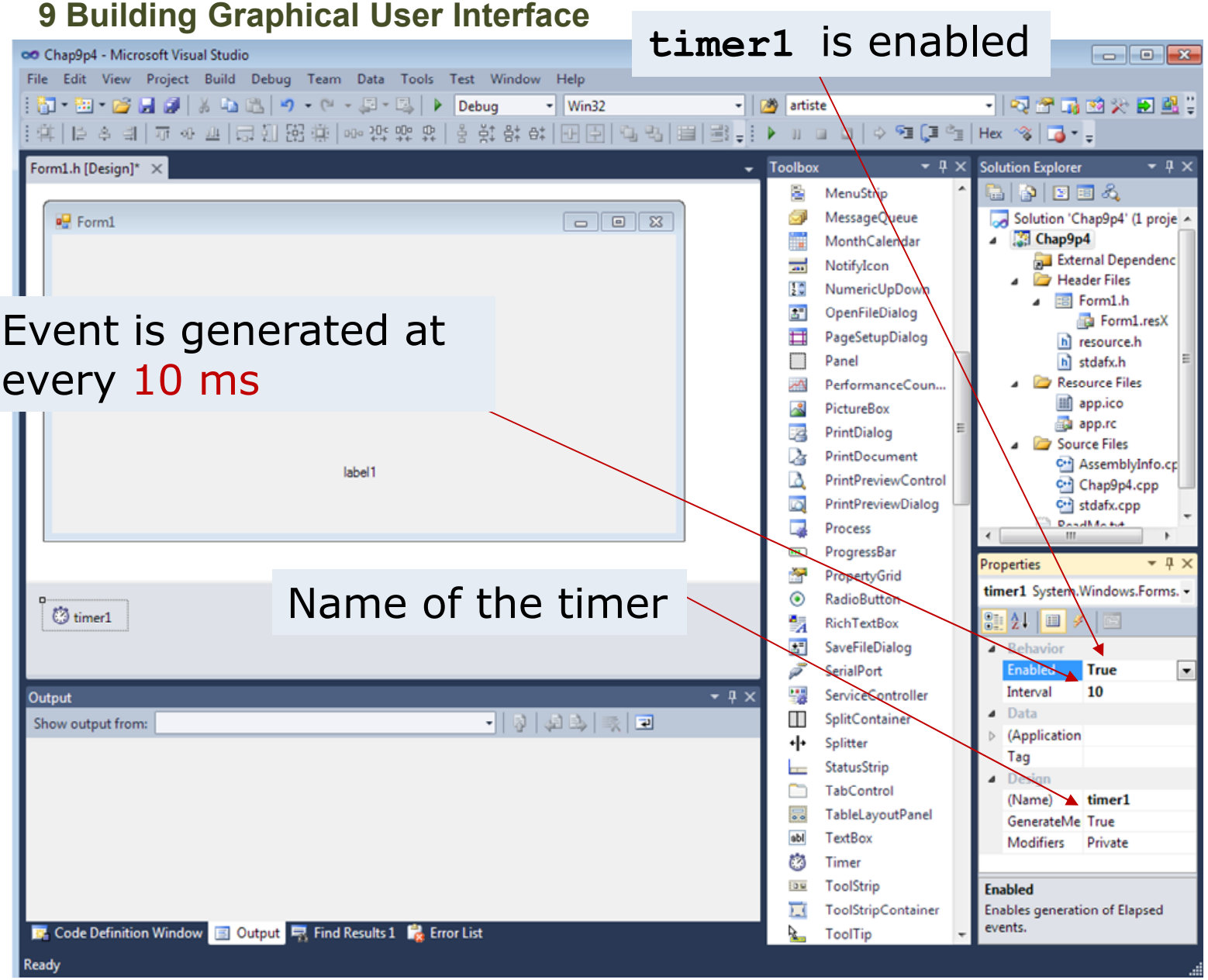


Add a label here

Drag a timer into the form. It will result in the **timer1** as shown in the bottom of the window

- **`Timer`** is one of the .NET GUI controls
  - When enabled, it generates an event for every clock tick, by default, of 100 ms
  - An event handler can be designed in response to every clock tick; hence, it is useful to the tasks that need to be done periodically in time,
    - e.g. to show an **animation**
  - By default, timers are disabled
    - All defaults can be changed in the **`Properties`** window.

9 Building Graphical User Interface



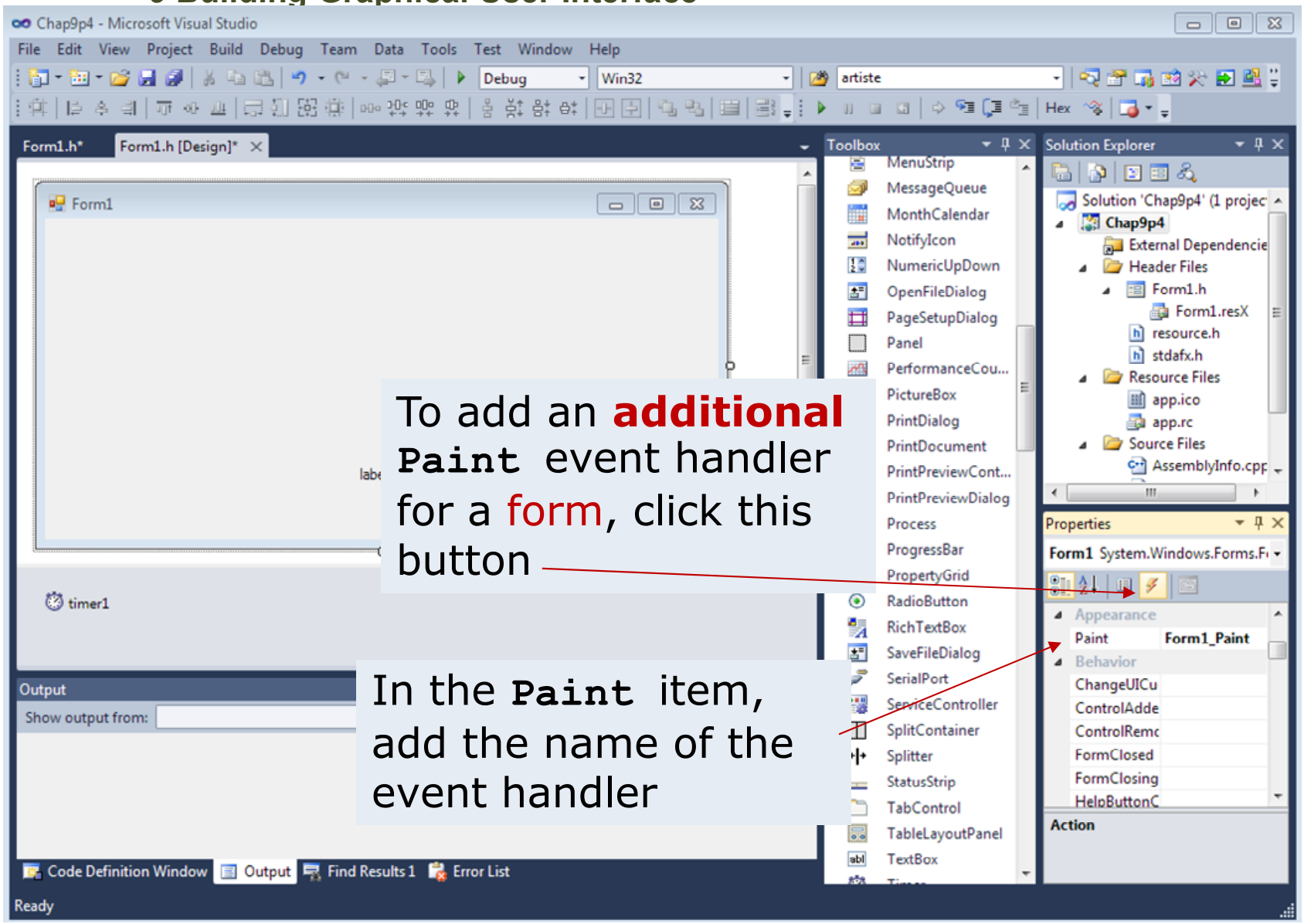
Paint Events to the Form

- All controls (e.g. a graphic) to a form generate a paint event when they determine that the form needs to be updated, e.g. at every clock tick of the timer
  - More specifically, a paint event is triggered when
    - a control is created, resized, or restored; or
    - when another control that had overlaid it is moved
  - By default, the paint event of a control is handled by one of its member functions called `OnPaint()`

Repaint every time

- **Additional** event handler can be added if one wants to do additional thing whenever the control is updated.

9 Building Graphical User Interface



Step 2: Develop the Event Handlers

- From Requirements 1 & 2, a picture and a graphic will move across the form; and the position of the picture is shown on the label
  - It can be implemented by changing the positions of the picture and graphic at every clock tick. To achieve this, we need to have
    - a handler for the **clock tick event** that
      - will show the position of the picture on the label;
      - generate a paint event for the whole form at every clock tick
    - a handler for the **paint event** that will draw the picture and graphic on the form at the new positions.

- From Requirement 3, the picture and the graphic will stop moving when one clicks on the label
  - To achieve this, we need to have a handler for the **label click event** that
    - toggles the state of the timer between "`Enabled`" and "Not `Enabled`" every time the label is clicked.

Step 2a: The Event Handler for `timer1`

```
public: Form1(void)          // Form1 constructor
{ X = 0;                     // Initialize the counter to 0
  InitializeComponent();
}  // X is the x-coordinate of the picture

private: int X;    // Serve as a counter of the
                   //   position of the picture
private: System::Void timer1_Tick(System::
Object ^ sender, System::EventArgs ^ e)
{ if (X%10 == 0)             // If X is a multiple of 10,
      label1->Text =""+X; // show its value on label1

  this->Invalidate();    // Generate a Paint message to
}                        //    Form1
```

```
this->Invalidate()
```

- Equivalent to `Invalidate()`
- Just to indicate we are not talking about other `Invalidate()`, but the `Invalidate()` of `Form1`
- Cannot use `Form1->Invalidate()` since `Form1` is the name of a class, not an object's address

- `Invalidate()` method is the **manual** way of triggering a paint event
- Since we are calling the `Invalidate()` of `Form1`, hence the paint event handler of `Form1` will be called (which is `Form1_Paint()`.)

`Invalidate()`: Invalidates the entire surface of the control and **causes the control to be redrawn**.

Step 2b: The Event Handler `Label1_Click()`

```
private: System::Void
label1_Click(System::Object ^  sender,
System::EventArgs ^  e)
{
    timer1->Enabled = !(timer1->Enabled);
    // Whenever the label is clicked, the
    // state of timer1 toggles between enable
    // and disable
}
```

Step 2c: The Event Handler `Form1_Paint()`

Already there

```
private: System::Void Form1_Paint(System::Object ^
sender, System::Windows::Forms::PaintEventArgs ^  e)
{
    Graphics ^g = e->Graphics;
    if(X < this->Size.Width)    { X += 1;}
       else  { X = 0;}
    Bitmap ^bp = gcnew Bitmap("c:\\temp\\cards\\h1.gif");
    g->DrawImage(bp, X, 25); //same as e->Graphics->DrawImage

    Drawing::Rectangle Head =
    Drawing::Rectangle(this->Size.Width-X, 25, 70, 70);
    g->FillEllipse(Brushes::Yellow, Head);
    Pen ^b4pen = gcnew Pen(Color::Black, 4);
    g->DrawEllipse(b4pen, Head);
}
```

It is a member property of **`PaintEventArgs`**, and is a handle to the class **`Graphics`**

Update position

Draw image

Draw circle

```
Graphics ^g = e->Graphics;
```

- The **Graphics** class is the heart of all rendering activities of GDI+
  - It's a device-independent representation of the drawing surface that you plan to render graphics on
  - The above statement indicates that a tracking handle **g** of a **Graphics** object is created to the drawing surface on **Form1** (i.e. **g** stores the **Graphics** object of **Form1**'s address returned by **e->Graphics**)
  - **Graphics** has many member functions
    - **DrawLine()**
    - **DrawRectangle()**
    - **DrawImage()**
    - **:**

x co-ordinate

```
if(X < this->Size.Width)
      { X += 1;}
else { X = 0;}
```

- For each paint event, the value of **`X`** will be incremented by 1
  - It will get back to 0 when it is equal to the width of **`Form1`**
- Since **`Form1`** can be resized during run-time, we need to check the actual width of **`Form1`** in the program
  - It is returned by **`this->Size.Width`**.

The height of the form is returned by **`this->Size.Height`**

```
Bitmap ^bp =
   gcnew Bitmap("c:\\temp\\cards\\h1.gif");
g->DrawImage(bp, X, 25);
```

y co-ordinate

- The class **Bitmap** is for the temporary storage of image data
  - The constructor of **Bitmap** requires a string that indicates the path of the image file
  - The location of the **Bitmap** object created (in the CLR heap) is stored by a tracking handle **bp**
- We can render the data in the **Bitmap** object on the drawing surface using the **DrawImage()** function of the **Graphics** object of handle **g**
  - The 2nd and 3rd parameters of **DrawImage()** refer to the *x* and *y* co-ordinates at which the image is drawn.

Top-left corner

Top-left corner co-ordinates

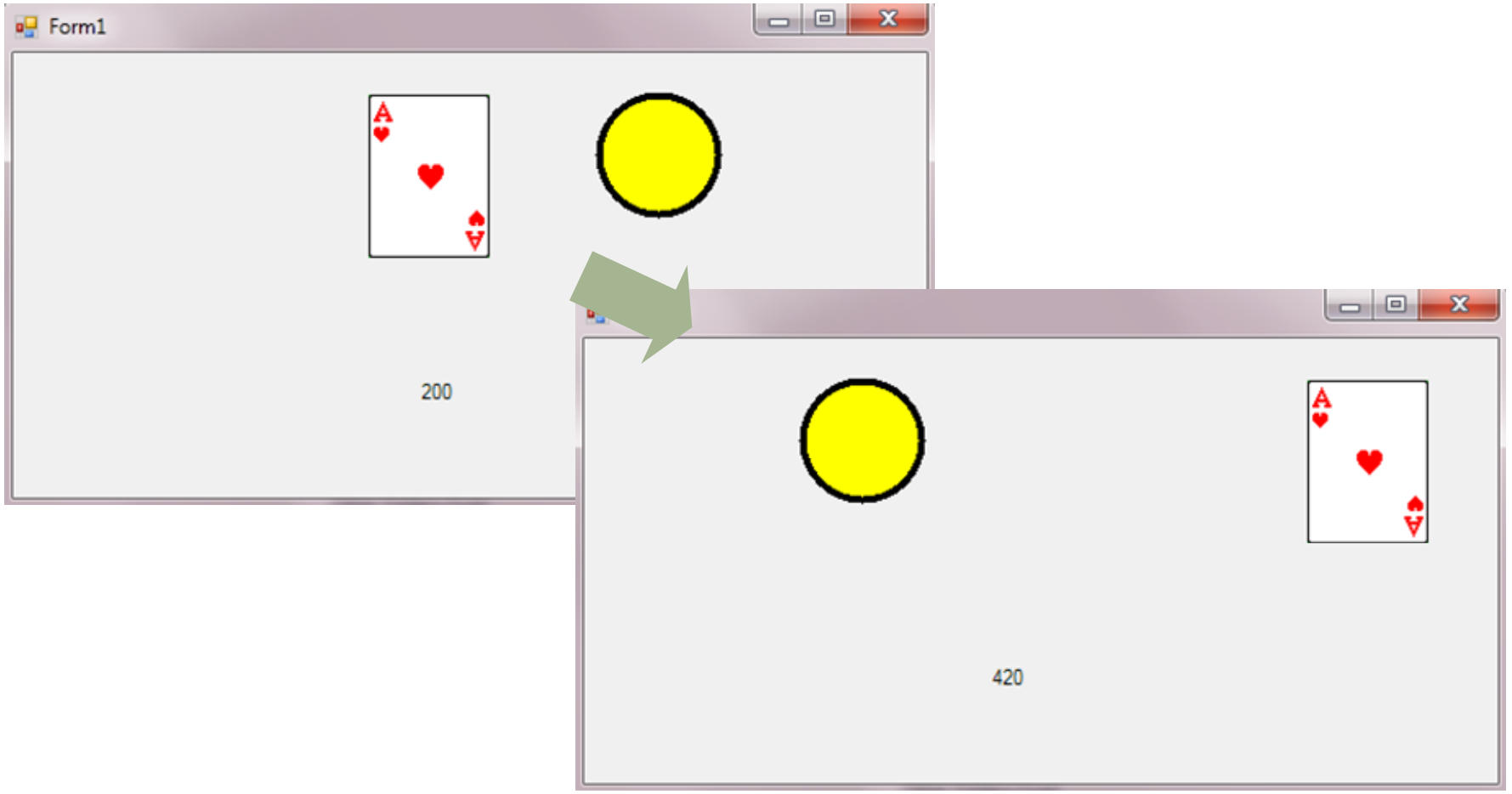
```
Drawing::Rectangle Head =
     Drawing::Rectangle(this->Size.Width-X, 25,
     70, 70);//Drawing::Rectangle Head(this->Size.Width-X,25,70,70)
g->FillEllipse(Brushes::Yellow, Head);
```

- The class **`Rectangle`** is a subclass of **`Drawing`**
  - The **constructor** **`Drawing::Rectangle()`** allows one to create a rectangular region **`Head`** with specified size and position. It requires 4 parameters
    - 1st and 2nd – the *x* and *y* co-ordinates of the rectangle
    - 3rd and 4th – the width and height of the rectangle
  - It will return the rectangular region created
  - **`g->FillEllipse(Brushes::Yellow, Head)`** allows one to create a filled ellipse
    - The ellipse is filled using the **`Yellow`** brush and has a size the same as that defined by the rectangular region **`Head`**.

```
Pen ^b4pen = gcnew Pen(Color::Black, 4);
g->DrawEllipse(b4pen, Head);
```

- Besides **Brushes**, GDI+ provides the **Pen** class for one to draw a graphic structure
  - The class **Pen** let us define the pen we can use for drawing
- The constructor of **Pen** requires two parameters
  - 1st parameter – the color of the **Pen**
  - 2nd parameter – the size of the **Pen**
- **g->DrawEllipse(b4pen, Head);** enable one to draw a circle on the drawing surface with a size as specified in **Head** using the pen **b4pen.**

Step 3: Build and Run the Project



Comparison with using standard form control

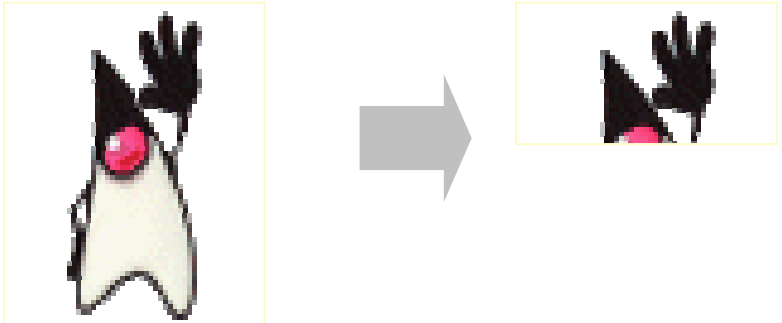
- The above can also be achieved using standard form controls, however, with much difficulty
  - E.g. if we want to show an animation of images, we can create many **`PictureBox`**'s on the way of the motion; and display the **`PictureBox`** one by one
    - ⇒ Very time consuming when developing
  - We can show a moving circle by converting it to an image and use **`PictureBox`**'s the way before
    - ⇒ Even more time consuming
  - GDI+ has simplified the job of graphic and image manipulation and provided much flexibility.

Double buffering

pixel or block of pixels

Memory → CPU ⇢ Screen

- Very serious flickering effect in the image and graphic may be seen
  - Rendering graphic information and transfer them onto the screen can be a relatively slow process
  - It is preferable to do it in a block rather than pixel by pixel

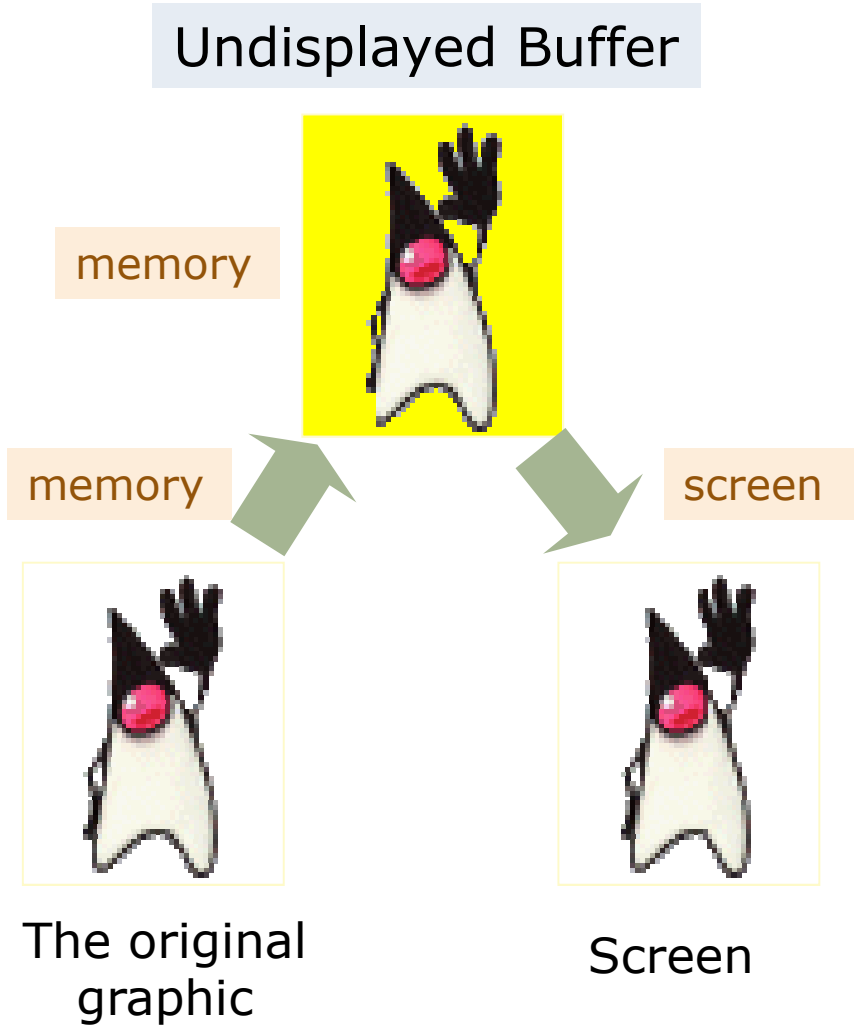


Traditional way

screen

- Graphic is rendered pixel-by-pixel directly onto the screen
- It so happens only half of the picture is drawn when it needs to be updated.

- When using double buffering, graphic is first rendered into a buffer (memory) first
- When the rendering process finishes, the whole buffer data are transferred to the screen in a block
  - Result in much faster and smoother graphic and image drawing.



- The easiest way to enable double buffering is to use default double buffer for forms and the controls, which is provided by the .NET Framework, i.e.
  - Enable default double buffering for our Windows Forms and authored Windows controls by using the **`SetStyle`** and **`UpdateStyles`** methods

```
public:Form1(void)   // Modified constructor
{                    // with double buffering
  X = 0;
  InitializeComponent();
  this->SetStyle(static_cast<ControlStyles>(
      ControlStyles::AllPaintingInWmPaint |
      ControlStyles::DoubleBuffer |
      ControlStyles::UserPaint), true);
  this->UpdateStyles();
}
```

Specify which 3 bits

- The **`SetStyle()`** method allows one to modify the **`ControlStyles`** of the control
- The **`UpdateStyles()`** method allows the modified **`ControlStyles`** to be effective
- When the control style **`DoubleBuffer`** is set to true,
  - Drawing is performed in a buffer, and after it completes, the result is output to the screen
  - Double-buffering prevents flickering
  - If **`DoubleBuffer`** is set to true, one should also set **`UserPaint`** and **`AllPaintingInWmPaint`** to true.

Learn these enumerations using Help

Exercise 9.4

Follow the above steps to build the project.

- See the difference between with and without Double-buffering

Modify the program such that

- The circle will move at 45° to the horizontal instead
- It will bounce when it touches the bottom and top of the window
- When the circle touches the left side, it will re-appear in the right side of the window.

Hint: Test whether the circle is rising or falling, and whether the top or bottom is reached.

Slides

- The powerpoint slides are provided by Dr Frank Leung, EIE.