An Introduction to Visual Studio 2012
(Visual C++) – Part I

Author: Huang Da
Contributors: Low WeiLin, Xie Kai, Zeng Yong
Last modified: 2014 Jan 12

This document describes how to build a simple windows app using Visual C++ and Visual Studio Ultimate 2012 (VS for short). The app takes in a non-negative integer from the user and outputs all primes less than the number. The final appearance looks like this:

Those who already know how to use VS can download the sample solution from the module website (same place as you download this document) play around with it. The final executable is in Release folder. Others, don’t worry. This tutorial will guide you step-by-step.

Tips appear in boxes like this. They aren’t usually necessary to follow this guide. But they can be helpful at times.

Highlights appear in boxes like this. Pay extra attention to them.

Extra info appears in boxes like this. These are additional things you can safely skip if you are not interested to learn more.
## CONTENTS

**Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRELUDE. DOWNLOADING AND INSTALLING</td>
<td>3</td>
</tr>
<tr>
<td>DOWNLOADING</td>
<td></td>
</tr>
<tr>
<td>INSTALLING</td>
<td>3</td>
</tr>
<tr>
<td>CHAPTER 1. GETTING FAMILIAR</td>
<td>5</td>
</tr>
<tr>
<td>CUSTOMIZE</td>
<td>6</td>
</tr>
<tr>
<td>VS WINDOWS</td>
<td>8</td>
</tr>
<tr>
<td>VS MENUS</td>
<td>8</td>
</tr>
<tr>
<td>CHAPTER 2. STRUCTURING THE SOLUTION</td>
<td>12</td>
</tr>
<tr>
<td>FILE ORGANIZATION IN VS</td>
<td>12</td>
</tr>
<tr>
<td>CREATING NEW PROJECTS</td>
<td>12</td>
</tr>
<tr>
<td>CONFIGURE THE PROJECTS AND SET AN ENTRY POINT</td>
<td>15</td>
</tr>
<tr>
<td>BUILDING AND RUNNING</td>
<td>20</td>
</tr>
<tr>
<td>CHAPTER 3. DESIGNING THE UI</td>
<td>23</td>
</tr>
<tr>
<td>OVERVIEW OF THE DESIGNER AND USEFUL WINDOWS</td>
<td>23</td>
</tr>
<tr>
<td>ADD COMPONENTS TO THE UI</td>
<td>24</td>
</tr>
<tr>
<td>CONFIGURE THE COMPONENTS</td>
<td>25</td>
</tr>
<tr>
<td>BIND EVENT HANDLERS TO EVENTS</td>
<td>28</td>
</tr>
<tr>
<td>CHAPTER 4. INTERACTING WITH THE LIBRARY</td>
<td>31</td>
</tr>
<tr>
<td>DEFINING THE LIBRARY INTERFACE</td>
<td>31</td>
</tr>
<tr>
<td>INTERACTING WITH A LIBRARY</td>
<td>32</td>
</tr>
<tr>
<td>CHAPTER 5. DEBUGGING</td>
<td>35</td>
</tr>
<tr>
<td>SINGLE STEP TRACE</td>
<td>36</td>
</tr>
<tr>
<td>SETTING BREAK POINTS</td>
<td>36</td>
</tr>
<tr>
<td>TERMINATING THE DEBUGGING SESSION</td>
<td>37</td>
</tr>
</tbody>
</table>
Prelude. Downloading and installing

This chapter is about downloading and installing Visual Studio 2012 Ultimate.

Downloading

We can download Visual Studio 2012 Ultimate from Dream Sparks at this link. On the main page, you should see this:

Now simply click “Sign in” at top-right corner of the page and sign in using NUSNET account:

If you cannot login, double check if you are in the correct Dream Spark sub-server. You must be in Dream Spark NUS, not other institute.
Then in the search box, type in “visual studio 2012 ultimate” and then press enter key, after a short while you should see a list of “visual studio 2012 ultimate” related products:

If you see “no result found”, you probably didn’t login using your NUSNET account successfully.

Although Visual Studio 2012 Ultimate on Dream Sparks is for 32bit only, you still can install it on a 64bit windows machine without any problems since 64bit is backward compatible (for more discussion, check here). If you are using OSX, you are recommended to install Parallel Desktop (or otherwise, e.g. Virtual PC on Mac) and run a virtual Windows 7 (or higher version) on it. For your information, Windows 7 is also available on Dream Sparks.

You can choose either installer based on your preference and click Add to Cart button beside the item. After checking out, you should see a confirmation page:
Note down the product key shown on this page, and click “Start Download” to proceed. You should see a guide page like this:

Get Your Software in 4 Easy Steps

1. Download the Secure Download Manager (SDM) installation file
2. Locate the file from step 1 and run it to install the SDM
3. Download the .SDX file for your order
4. When the SDM launches, start downloading your software

Follow the steps on this page, and you should be able to start downloading Visual Studio 2012 Ultimate.

Installing

After the .iso file is downloaded, open it, and run the installer:

If you encounter any notifications about missing files during the installation, just choose “download from online”.

If you install VS2012 alongside VS2010, you may get into trouble when converting a VS2010 solution to VS2012. You can solve the problem according to Andy Rushton’s suggestion given here.
Chapter 1. Getting familiar

This chapter will provide introductions on customizing Visual Studio 2012 Ultimate, referred to as VS from this point.

Customize

VS lets us choose for a selection of pre-defined environment setting combinations that is optimized to building different types of applications.

We suggest choosing Visual C++ Development Settings, which minimize the settings you should configure.

💡 You can always reset your current setting or set to other pre-defined settings through TOOLS  Import and Export Settings.

💡 You are able to do additional customizations to Visual Studio, such as changing the color theme, font and size of the code, foreground and background color of the editing area, etc. All these preferences can be adjusted by selecting TOOLS  Options.
Changing VS color theme

First we need to open **Options** dialog box:

Then in the **Environment** section, we choose **General**, and you should be able to see a drop-down list for available color theme. Default color theme is **Light**, and now we are going to change it to **Dark**:

After clicking on **OK** to confirm the changes, our IDE should look like this:
VS windows

After you open VS, you can identify the tool windows, the menus and toolbars, and the main window space. Tool windows are docked on the left and right sides of the application window, with Quick Launch, the Menu bar, and the standard toolbar at the top. In the center of the application window is the Start Page. When a file is loaded, text editor and/or designer will appear in this space. When you develop an application, you’ll spend most of your time in this central area.

VS menus

Visual Studio has docked most of the options you may use frequently at the standard toolbar, such as Create a new solution, Open an existing solution, Copy and Paste, Debug type, etc. You are able to add or remove buttons docked on the standard toolbar by clicking the dropdown button at the right-most end of the toolbar:

Generally, if you don’t know which menu or submenu the desired option is located, just type the key words in Quick Launch, or press Ctrl + Q. For example, if we want to create a new project but have no idea where the option “New project” is, we can search using Quick Launch:
- **FILE** menu contains most of the file manipulation options, such as **New** or **Open** project solution, **Save** file or files, **Recent** files, **Print** options, etc.
- **EDIT** menu contains text editing options, such as **Copy** and **Paste**, **Undo** and **Redo**, **Find** and **Replace**, etc.
- **VIEW** menu contains various **Explorer** and **Windows**, which may help you manage your project better. Here we suggest you to keep two windows docked on application window: **Solution Explorer, Class View**.

---

**Solution Explorer and Class View**

Solution Explorer provides an overview of all **projects** under the current **solution** (these two terms are defined here). It will be very helpful when your solution contains multiple projects. You are able to navigate down the project, from high-level to the lower-level details:

![Solution Explorer](image)

This explorer is docked at the left side of the application by default. In the event you close it by accident, you can re-open it by selecting **VIEW → Solution Explorer**.

![Solution Explorer](image)

Unlike **Solution Explorer** which gives a high-level view in terms of files, **Class View** provides a hierarchical view of all projects in terms of classes. You can view all namespaces and classes you created in each of the project, and check its member functions and variables and their accessibility efficiently.

![Class View](image)
The class view is docked at the left side of the application window by default. If you close by accident, you can re-open it by selecting **VIEW → Class View**.

- **PROJECT menu** and **BUILD menu** are a bit useless since all the options listed in the menus can be found by right clicking on a target project in Solution Explorer.
- **PROJECT menu** provides methods to structure our solution and configure a project. The options in this menu will be elaborated more in the next chapter, in which we are going to structure and configure our solution.
- **BUILD menu** provides functionalities to compile the code and build the executable. We are able to configure the mode, namely **Debug Mode** and **Release Mode**, the solution will be built. This will be covered in another
document.

- **DEBUG** menu provides functionalities to debug the whole project. We are able to **Step Into** or **Step Over** an instruction, or **Step Out** a chunk of code. We can set a **Break Point** so that we are able to jump directly to that point. However, using hot keys and shortcuts docked at the toolbar can equally do all these functionalities. More details will be covered in Chapter 5.

- **TOOLS** menu provides some extensions to Visual Studio 2012 Ultimate. But this is out of our discussion, so we will skip this menu.

- **TEST** menu provides some options you are able to manipulate with test cases. However all the options can be found on another useful explorer called **Test Explorer**. As for how to call the explorer and make use of it, we will cover it in another document.

For other menus, we can ignore them for now, as they are not that helpful in our project.

Now we have gone through all necessary menus and it's time to start our first project in Visual Studio. In the next chapter, we will set up a project and structure the solution step by step.
Chapter 2. Structuring the solution

This chapter is about structuring your solution.

File Organization in VS

In Visual Studio, we are building a **Solution** to solve a problem. This solution consists of one or more **Projects** to solve sub problems. We will set a project as the **Startup Project** and set an **entry point** so that VS knows where to start executing the code. Each project involves one or more **Source** files and **Header** files, in which we define routines to solve the specific sub problem.

Let's see how this organization works in our sample project. Ultimately, we need to build a solution for our product, a prime generator. We need at least two components to achieve the functionalities: a **UI**, which handles the interaction between the user and the program, and a **back-end library**, which handles calculations. Accordingly, we can divide our solution into two projects.

Note that different sub problems may require different types of projects. In the next few sections, we look at how to configure projects.

Creating new projects

From the VS main window, select **FILE → New → Project**, or press hot key combination **Ctrl + Shift + N**, you should be able to see **New Project Dialog** given below:

Since we have chosen **Visual C++ Development Settings**, Visual C++ is our default language preference. In the expanded list, we can see several choices. The most general option is **General → Empty Project**, which provides a project with
default settings. Choosing other project types will give you a configuration optimized for the respective project type.

First, let us create a project for the UI part of our solution. Let us use Windows Forms to build the UI. Note that there are other third-party UI libraries such as Qt that you can use.

A windows form is based on a runtime environment called Common Language Runtime, or CLR for short. Therefore, we should navigate to the CLR page and select CLR Empty Project as our UI project prototype.

![Image](image.png)

More specifically, we are using C++/CLI to build the windows form application. You are recommended to read this page and this document to get necessary knowledge on C++/CLI.

To finish creating a project, you need to specify the location you want to store the project and the name you want to give. All projects will be listed under a Solution, and you should specify the solution’s name also. In our project, we give the current project a name “UI” to identify its functionality, and name our solution “PrimeGenerator”. After clicking on OK button, you should be able to view the current solution and projects under it in Solution Explorer.

![Image](image.png)

The second project, Library, is no more than a container for functions we want to
use. Therefore, an empty general project will serve the purpose. To add the project into the current solution, right-click on the current solution, select **Add → New Project:**

![Add New Project](image)

Then navigate to **General** page and select **Empty Project:**

![Create Empty Project](image)

Now you should see two projects under our solution **PrimeGenerator:**

![Solution Explorer](image)

You may notice that **UI** is in bold font but **Library** is not. This is because **UI** is
automatically set as the **Startup Project**, which is what we want. We will explain more on this topic in the next section.

Now we have created a solution with two standalone sub projects. In the next section, we will illustrate how to configure each project to do their specific job, get the necessary links up, and set an entry point.

**Configure the projects and set an entry point**

First we need to apply necessary extra settings on each of the project. Let’s start from configuring the **Library** project.

Based on our design, Library provides all necessary functional classes and static functions. It is independent from all other components such as the UI. In other words, it does not care about other classes’ functionalities. For example, a Test method can also call Library’s functions, and Library does not even care about who is the caller but just answers the call. Therefore, we’d better set our Library project as a **Static Library** project to enable this kind of independence.

To set as a Static Library, right click on the project in **Solution Explorer**, and select **Property**.

![Configuration settings](image)

In **General** page, you should see an option called **Configuration Type**. By default, this option is set as **Application (.exe)**. Select the dropdown list, and you will see the wanted type, **Static library (.lib)**. Select **Static library (.lib)**, then click **OK** to apply the change.

Now, let’s configure the **UI** project. We want our **UI** to be a **Windows Form Application**, so we need to do some extra settings to suit this purpose.

Right-click on **UI** project in **Solution Explorer**, and select **Property**. Navigate to **Linker** section, and select **System page**.
In this page, we are going to configure our subsystem as a Windows system. Select the dropdown list of **SubSystem**, and in the list select **Windows**, then click OK to apply changes.

![SubSystem dropdown list]

Now it's time to connect the two components: UI and the Library. According to our design, **UI** depends on the **Library**. Therefore we need to set a reference and dependency of Library in UI.

Let's set the reference first. Right click on **UI** project, and select **References**. Then click **Add New Reference** button.

![Add Reference dialog]

In **Add Reference** dialog, check the **Library** project, and click OK to apply changes.
Now we have settled the reference side, but we still need to set the additional **Include Path** so that we don’t need to specify functions headers with long relative paths. Normally, if we do not set the Additional Include Path, we will have to type something like: `#include "../AnotherProjectDirectory/SomeHeader.h"` in the .h/.cpp files, instead of simply `#include "SomeHeader.h"`. If we set the include path incorrectly, VS may fail to compile your program and then produce **Fatal Error C1083: Cannot open include file**. So if you encounter this error, please check your include path settings.

In **UI property page**, select **Configurable Properties**, and then navigate to **C/C++** tab, then select **General**.

Click the dropdown button of **Additional Include Directories**, and in the dropdown list select `<Edit...>`, then you will see **Additional Include Directories** edit dialog. Click on the “folder” icon at the top, then you should be able to add new include directories.
Click “…“ to edit the directory. By right, you should choose ..\Library as the path.

Then click OK to apply the changes.

Until now, all necessary configurations are done except one thing: setting an Entry Point. An entry point refers to the first function (Entry Function) that the program will execute. For example, main function in an ordinary single-class C++ project is an entry function, or entry point. In our project, we want our GUI to start up when we run the project, so our entry function should be the main function of UI project. To achieve this, we need to configure two things: set UI project as a Startup Project, and set Main function of UI as entry function.

Luckily, UI project is set as Startup Project by default because it was the first project we crated in this solution. You can set a project as the Startup Project by right clicking on the project in the Solution Explorer and choosing Set as Startup Project.

To set the main function, we need to create our main function first. Right click on...
the **Source Files** filter, select **Add  New Item**. In the **Add New Item** dialog, navigate to **UI** tab, and choose **Windows Form**. We should give the GUI a meaningful name, say **PrimeGeneratorGUI**.

Then click on Add to create the file.

Then you may notice that not only one .h file has been created, but also a **.cpp** file. The .h file defines the properties and event handlers of all the components of the Windows Form, in the form of a class. Then, this class is instantiated in the .cpp file, and run in an infinite loop. Therefore, the .cpp file is responsible for constructing the GUI, and that's why we need to set it as our entry point.

In .cpp file, insert the following code:

```cpp
#include "PrimeGeneratorGUI.h"

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

[STAThread]
void main(array<String^>^ args) {
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    UI::PrimeGeneratorGUI mainWindow;
    Application::Run(%mainWindow);
}
```

💡 You can make line number visible by check **Line Numbers** in **TOOLS  Options  Text Editor  All Languages  General  Display  Line Numbers**.

📖 Quick pick-up of C++/CLI

Basically, all **C++/CLI** object should be managed, unlike native **C++** objects,
which are ‘unmanaged’. For managed objects, ‘^’ is similar (but not equivalent) to ‘*’ in native C++, and ‘%’ is similar (but not equivalent) to ‘&’ in native C++. In addition, if we want to “new” a managed object in CLR, we should use gcnew keyword, instead of new.

[STAThread] Single Thread Application Attribute
The thing we put in the square brackets before main is called an attribute. STAThread attribute indicates the current application runs on a thread that contains a Single Thread Application. You can learn more on STAThread attribute on MSDN or this blog.

Next, we need to set this main function as our entry function.

Right click on the UI project in the Solution Explorer, select Properties, navigate to Linker tab, and then select Advanced. In Entry Point box, put in our entry function’s name, which is main.

Then click OK to apply the changes.

Building and running

Now that the setting up is complete, let’s run the project. Right click on the solution in Solution Explorer and select Build. The progress of building shows up in the output window docked at the bottom of main application window.

Oops, we got a link error (as expected, if only one link error is reported).
Don’t worry, this is because we didn’t put anything in our Library project. We can fix this problem by adding a dummy class in Library.

Right-click on Source Files under Library project, select Add → Class.

In Add Class dialog, navigate to C++ tab, and select C++ Class.

Then you should see a Generic C++ Class Wizard. In the wizard, specify the class name, and then click OK. Here we name our class PrimeGenerator.

Now you should see a PrimeGenerator Class has been added to our Library project, with separate .h header file and .cpp implementation file.

To rebuild, right-click on the solution in Solution Explorer, and select Rebuild Solution.
Now you should be able to see 2 projects built successfully.

If the build was successful, we can run it. VS provides two modes of running: with Debug mode and without Debug mode (we will discuss more on this topic in later chapter). This time we choose run without Debug. To run it, click on DEBUG menu, and select Start without Debugging.

Then you should see an empty GUI pops out.

Now we have finished the basic steps and obtained a skeletal solution.
Chapter 3. Designing the UI

The UI acts like a ball-passer. It passes the input from the user to the back-end library, and displays the results returned from the back-end. This chapter explains how to use the native Windows Form Designer to flesh out the UI.

Overview of the Designer and useful windows

Double clicking the PrimeGenerator.h under UI project produces the Designer page in main editing area.

Note: Alternatively, you can right click on PrimeGenerator.h, and select View Designer.

There are two important windows that may assist you in designing the UI: Toolbox window and Property window. By default, they should appear at the right side of the main application window.
If you accidentally close Toolbox, you can reopen the window through VIEW → Toolbox. For opening Property window, simply right click the target object in the designer view and select Property.

Toolbox window provides necessary components you may need in a GUI, such as Textbox, Label, Button, etc. You can drag and drop the component you want into the Designer and configure the properties and/or events of that component.

In Property window you can edit the property of a target object, and bind an event handler function to a certain event. For your information, it’s possible to initiate another event handler inside an event handler. In other word, it’s possible to trigger an event by another event, or what we call, triggers a chain effect. We will explain more on handling event in later section.

Add components to the UI

To design the UI, simply drag and drop the components you want into the designer and do some basic adjustments, like resizing the components, finding a good place for it, etc. At this step, we focus on “getting the components” first. Tweaking the look-and-feel can be done later.

For simplicity, we will demonstrate to build up a very simple GUI.
Here is the GUI with all default settings. In this GUI, we dragged in two **Labels**, two **Textboxes**, and one **Button**.

**Configure the components**

To change the text contents of labels and button, we need to configure the **Text** property of the component. For example, if we want to change the text of upper label to “Primes less than upper bound:”, right click on the upper label and select **Property**. Then in the **Property** window, **Text** property is highlighted:

Here, we can change its **Text** into something we want. You are also able to change the font and size of the text. You can try it and find the best combination. In this example, we leave all the other settings unchanged.

Right after you leave the focus, the latest change will be reflected in the designer:

Repeat the same procedure, we can configure the text of all other components.
Next we will configure the text boxes. For the display box, we want it to be a multiple-line display. This is controlled by Multiline property. Right click the upper text box and select Property, in Behavior section you should find Multiline property.

By default, the text box is configured as a single-line box, to enable multi-line, we should change Multiline property to true. After you made the change, the text box will be resizable in 8 directions. Resize it to what you want it to be.

You can look into other properties of a text box, and modify them to suit your design. Here we want to modify two more properties, Readonly and Scrollbar. Readonly property controls whether the text box is write enabled, meaning if you are allowed to key in something in the box.
Scrollbar property controls scrollbars at side and bottom.

Below is the final appearance after configuration.

Now you can build and run the project and see the outcome.

We have finished all necessary configurations. We can view the code by right-clicking PrimeGenerator.h under UI project and select View Code.

Note how the code for initializing components has been created automatically. However, we can change the names for better code quality. We can change the names in code directly, but here we suggest you to change the names in designer using Property window. In the designer, choose the component you want to change the name, right click on it and select Property. In the Property window, navigate to Design section, and (Name) property is the one you need to configure.

We can give the label a proper name, say outputBoxInstructionLabel.

We can repeat this step until all components are given a proper name. Now, view code again and you will see the changes have been applied.

The UI looks fine but has a potential layout problem: if you resize or maximize the window, the layout gets messed up. The most obvious solution is to disable the Maximize function of the window and set the border as un-resizable.

To set the form a fixed border and disable maximize button, you need to configure
FormBorderStyle property and MaximizeBox property.

Right click on the form, and select Property. In Property window, navigate to Appearance section and set FormBorderStyle to FixedSingle.

Then navigate to Window Style section and set MaximizeBox to false.

Now try to build and run again, and you will find it can no longer be resized and maximized.

Customize Tab Index

TabIndex property defines the order when you use Tab key to navigate.

TabIndex equals to 0 indicates it’s the very initial focus of the program. In our project, we want the input box as the very first focus, and then the Generate button, then the output box.

To achieve this, we need to configure TabIndex property of target components.

The process of configuring this property is almost the same as configuring previous properties, so details are omitted here and left as an exercise.

You may find labels also have TabIndex property, but they are dummy values since labels cannot be focused in the Windows Form.

Bind event handlers to events
An event is something that happens to the UI, for example, clicking a button, a textbox getting focus, and focus leaving a text box. When an event occurs, an event handler is triggered. It defines the reaction of the program to that event. Let’s take clicking Minimize Box button as an example. When you click on the button, a system event handler is called. That event handler contains the code to hide the main window. As a result, you will see your window is minimized. More formally, event handler defines a routine that the program should take when a specific event occurs. In our project, we should define an event handler for clicking on the Generate button. We want to get a prime list right after we clicked this button.

To bind an event handler to Click event, right click on the Generate button and select Property. In the property window, you will see a “lightning” icon at the top of the window, click on it and it will lead you to the Events page.

We want to bind an event handler to the Click event. In Events page navigate to Action section and double click the editing area. Then, an event handler called generateButton_Click will be created and bound to the event automatically.

**Note:** MouseClick event is slightly different from Click event. MouseClick event will be triggered only if the button is clicked by a mouse, but Click event will be triggered as long as the button is pressed (either by mouse or keyboard enter).

In the code, you will see the prototype of the event handler. You can add your event handler code in this function. For now let’s just insert a very simple “hello world” code to test the functionality:

```c++
private: System::Void generateButton_Click(
    System::Object^ sender,
    System::EventArgs^ e) {
    MessageBox::Show("hello world!");
}
```

Now build and run the project again, and you will find the Generate button can be clicked and show a message “hello world!” in a message box!
However, UI itself cannot generate a prime list, it needs support from the backend library. In the next chapter, we will discuss more on interacting with backend library.
Chapter 4. Interacting with the Library

This chapter describes how to get the UI to interact with the backend library.

Defining the library interface

UI and the library should interact through the public functions provided by the library. We should define these functions’ prototype in the .h header file, and the implementation should be written in the .cpp source file.

Double click on PrimeGenerator.h to edit the code. According to our design, we should create only one public function and one private prime list.

```cpp
#pragma once
#include <vector>
#include <algorithm>
#include <exception>

class PrimeGenerator {
private:
    std::vector<int> primeList;

public:
    PrimeGenerator(void);
    PrimeGenerator(int);
    ~PrimeGenerator(void);

    std::vector<int> GetPrimeLessThan(int);
};
```

You may notice the coding standard provided by Visual Studio is a bit different from our coding standard. However, there is no way to change coding style for C++ in Visual Studio, so we need to change them manually.

We should include all necessary header files in .h file instead of .cpp file. Usually it’s NOT a good practice to do “using namespaces”, especially “using namespace std”. This is because omitting namespace may result in ambiguous function when the two functions in two namespaces share the same name and parameter list. You can check this page for more discussions.
After defining the prototype, we can go into the source file and do the implementation. Double click on PrimeGenerator.cpp to edit the code. For now, we put in some dummy stuff for testing only.

```cpp
#include "PrimeGenerator.h"

PrimeGenerator::PrimeGenerator(void) {
    primelist.push_back(2);
}

PrimeGenerator::~PrimeGenerator(void) {
}

std::vector<int> PrimeGenerator::GetPrimeLessThan(int upperBound) {
    return primelist;
}
```

**Interacting with a library**

To use functions from another project, we need to include its header file first. Since we have already specified the dependencies and reference path, we can include the `PrimeGenerator.h` without specifying the full path.

```cpp
#include "PrimeGenerator.h"

namespace UI {
...
}

Now we can use all the functions defined in the header file.

```cpp
#pragma once

#include "PrimeGenerator.h"
#include <vector>

namespace UI {
...
/// <summary>
/// Summary for PrimeGeneratorGUI
/// </summary>
public ref class PrimeGeneratorGUI : public System::Windows::Forms::Form{
private:
    PrimeGenerator* prime;

public:
    PrimeGeneratorGUI(void) {
```
InitializeComponent();

prime = new PrimeGenerator();
}

protected:
    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    ~PrimeGeneratorGUI() {
        if (components)
        {
            delete components;
            delete prime;
        }
    }

    ...
private:
    System::Void generateButton_Click(System::Object^ sender, System::EventArgs^ e) {
        std::vector<int> primeList = prime->GetPrimeLessThan(2);
        ...
    }
}

Now, try to build the project again, and you should see both projects build successfully. Then we can proceed to write some code to test the functionality of the library.

System::Void generateButton_Click(System::Object^ sender, System::EventArgs^ e) {
    std::vector<int> primelist = prime->GetPrimeLessThan(2);
    int totalPrime = primelist.size();
    String^ s = "";

    for (int I = 0; I < totalPrime; I++) {
        s = s + primelist[i].ToString();

        if (I != totalPrime - 1) {
            s = s + ",";
        }
    }

    outputBox->Text = s;
}
Build and run the solution, now clicking on Generate button and you will see the hard-coded value is displayed in the output box.

You can handle the rest of coding according to the design. In the next chapter, we look into debugging.
Chapter 5. Debugging

This chapter describes some useful debugging features provided by VS.

To go into the debugging mode, click to DEBUG \(\rightarrow\) Step Into. When you are in the debug mode, you will see two new panels docked at the bottom of main window.

![Debugging Panels](image)

Usually, Autos and Watch window in the left panel, and the Call Stack window in the right panel are sufficient to do debugging.

**Autos** will display the current value for all active objects and variables. **Watch** allows you to define which variables and objects you want to trace. **Call Stack** window will display function stack information.

![Autos and Watch Window](image)

You can also view the current value stored in a variable by hovering your mouse on the variable’s name, as shown below.

![Variable Value](image)

A yellow pointer at the code side bar will point to the current instruction. Although you can drag the pointer around from a point to another statement, doing so is not recommended because it can sometimes mess up the execution order.
Single step trace

The *single step trace* mode allows you to trace your code line by line. You can choose to **step into**, **step over** or **step out** a function.

**Step into** will bring you into the implementation a function. Consider the following code segment:

```c
double ExampleFunction() {
    return 2.13;
}
```

```c
int main() {
    double a = ExampleFunction();
    int b = 213;
}
```

If we step into at the red line, we will start tracing the green line.

**Step out** is opposite to **step into** instruction. Using **step out** will go out of current function with whole function executed. In the code example above, if we step out at the green line, we will get back to the red line but variable a has already been assigned with 2.13.

**Step over** is like a **step into** and **step out** combo. It will not go into the detailed implementation of current statement and go to the next statement (with the current statement executed). In the code example above, if we step over at the red line, we will go to the blue line with variable a set to the result of `ExampleFunction()`.

All these three single step functions can be found on the toolbar under *Debug* mode. Alternatively, you can find them in *DEBUG* menu.

Setting break points

We may want to skip a big chunk of code and jump to some statement that we are interested in. In this case, we can set a break point beside that statement and then use **Continue** button to continue execution until the break point.

To set a break point, single click on the side bar of editing area beside the line you are interested in. A red dot will show up to indicate a break point.

**Continue** button can be found on the toolbar under *Debug* mode, or in *DEBUG* menu.
Terminating the debugging session

To terminate a debugging, you should click on Stop (●) button on the toolbar under Debug mode.

You are not recommended to change the code when you are in Debug mode. Always remember to terminate debugging before you start changing your code.

Now this is the end of the first half of tutorial. You can use the debugging skill introduced in this chapter to trace your own project or the given sample project.