Table of Contents

Introduction – How Programs are Created and Run

The diagram below (Figure 1) illustrates the typical process of creating and running a program:

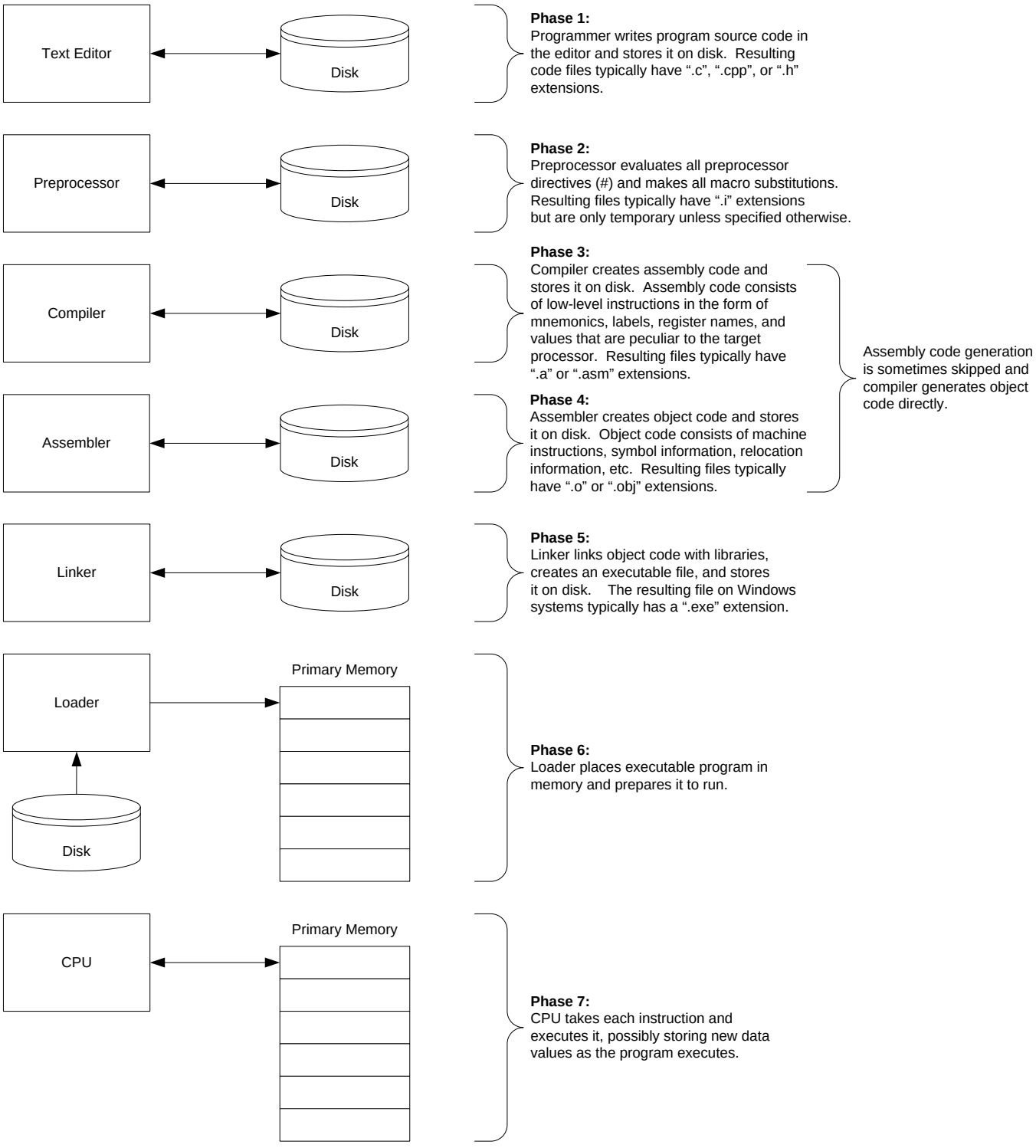

Figure 1

The IDE (Integrated Development Environment)

What is an IDE?

An IDE (Integrated Development Environment) is an all-in-one suite of tools that can be used to perform all of the steps outlined in the previous diagram (Figure 1). This allows a programmer to conveniently develop, run, and debug a program without ever leaving the IDE and without the need for any additional tools. Unless you just want the experience of developing your programs without an IDE, I don't recommend it simply because of the additional and unnecessary complexity (and pain) involved.

Microsoft Visual Studio

All IDEs have their own peculiarities and set up requirements and this document is intended to explain and resolve some of the more common issues students may encounter when using them. The Microsoft "Visual Studio 2017 Community – Update 3" IDE installed on Windows 10 is used as an example but other versions should be similar enough that this information will work well for them too. While most of the topics covered also apply to products other than Visual Studio, the step-by-step details can differ significantly and often require students to determine the exact procedures for themselves, typically by referring to online resources.

Getting Visual Studio

Unless you simply want to pay for Visual Studio I recommend that you get the "Community" edition. **It's free forever if you sign in to your Microsoft account from within the IDE, but will expire in 30 days if you don't**. Microsoft accounts are totally free. Other editions of Visual Studio are not free but do come with a "90-day free trial". Go to https://www.visualstudio.com/ to find what you want.

Changing the IDE Defaults

Although some students begin using their IDEs for the course assignments without carefully reading and following the suggestions in this document, my experience has been that most of them end up with much more wasted time and frustration than if they had just taken the time to make the changes in the first place.

Installing the IDE (Visual Studio 2017 Community Edition)

1. When you start the "Visual Studio 2017 Community" installer you will be notified of the license terms and usual legal stuff. Simply click the ***Continue*** button if you agree. The next window you see should be similar to that in Figure 2. Place a check in the checkbox for "Desktop development with C++" then click the ***Install*** button. This will display the window shown in Figure 3 on the next page.

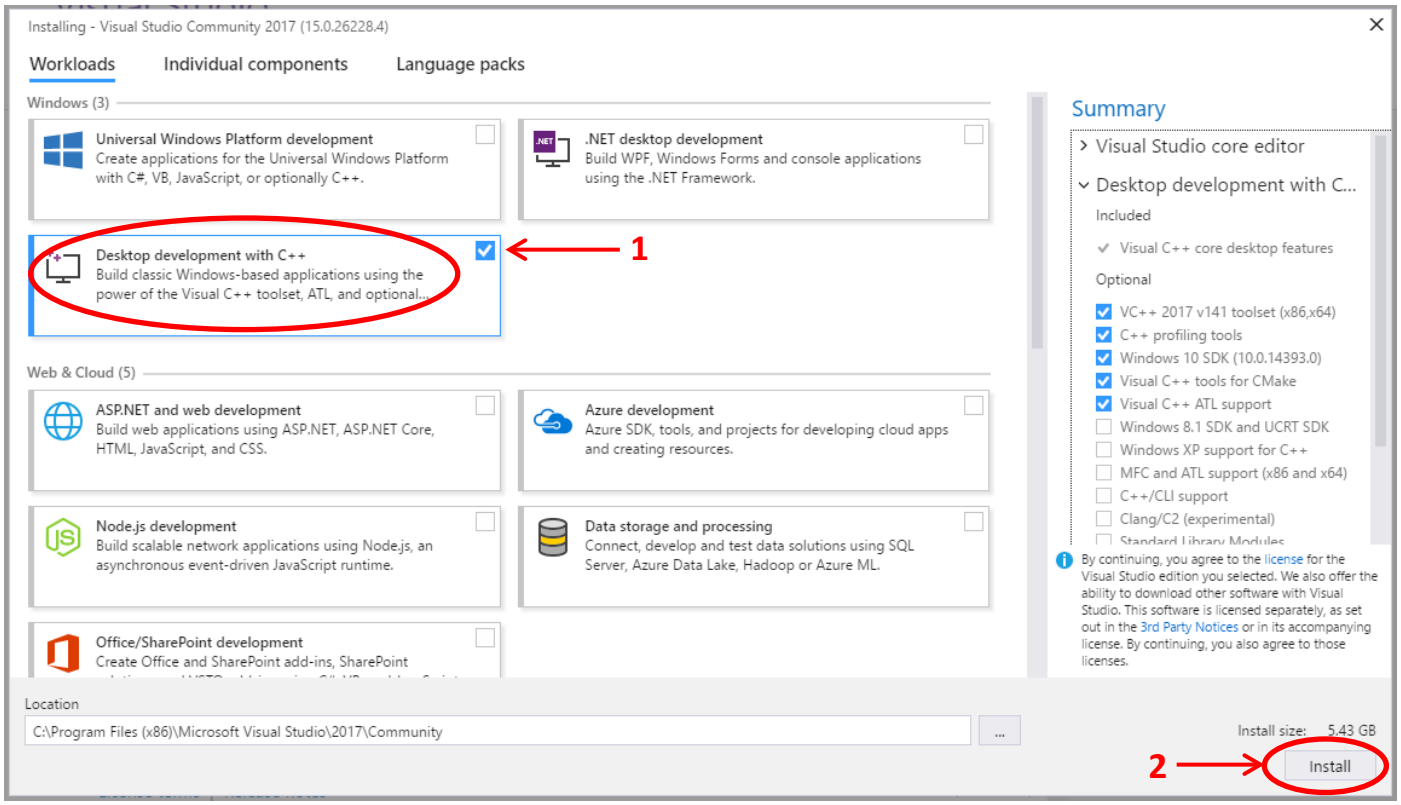

Figure 2

Continued on the next page...

Installing the IDE (Visual Studio 2017 Community Edition), *continued*

2. The installation process (Figure 3) will acquire and apply various different packages and will take some time so please be patient. When it is complete the window shown in Figure 4 will be displayed. Click the ***Start Visual Studio*** button, which will display the window shown in Figure 5 on the next page.

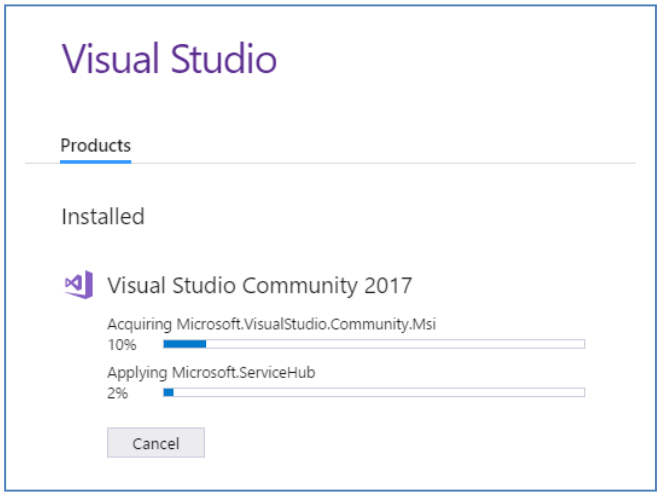

Figure 3

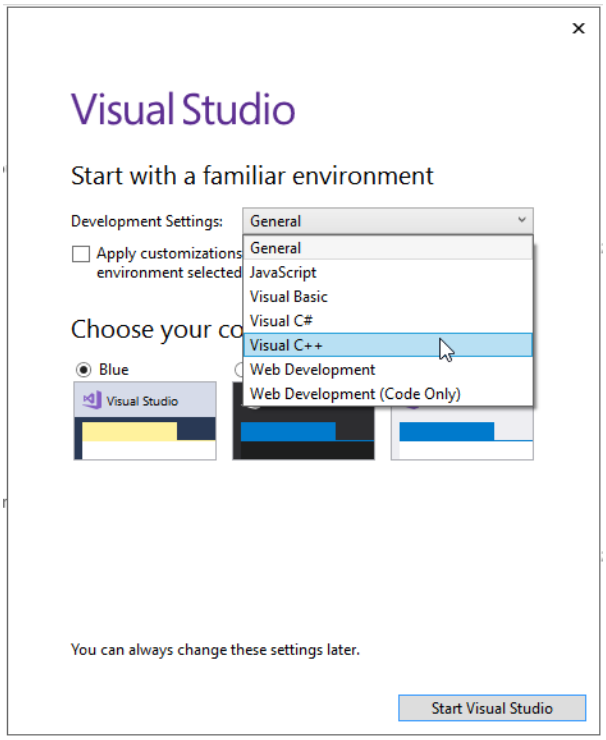

Figure 4

Continued on the next page...

Installing the IDE (Visual Studio 2017 Community Edition), *continued*

3. If you have a Microsoft account the window in Figure 5 lets you sign into it. **This is not mandatory but Visual Studio 2017 Community will expire in 30 days if you don't**. Click the ***Sign in*** button to sign in or the ***Sign up*** hyperlink below it to create a new free account. To skip signing in click the ***Not now, maybe later*** hyperlink. Regardless, the environment selection window shown in Figure 6 will be displayed next. Select **Visual C++** from the **Development Settings:** dropdown list then click the ***Start Visual Studio*** button to start the IDE and display the main IDE window shown in Figure 7 on the next page. The steps just shown in Figures 5 and 6 will be remembered and will not be needed again when the IDE is opened in the future.

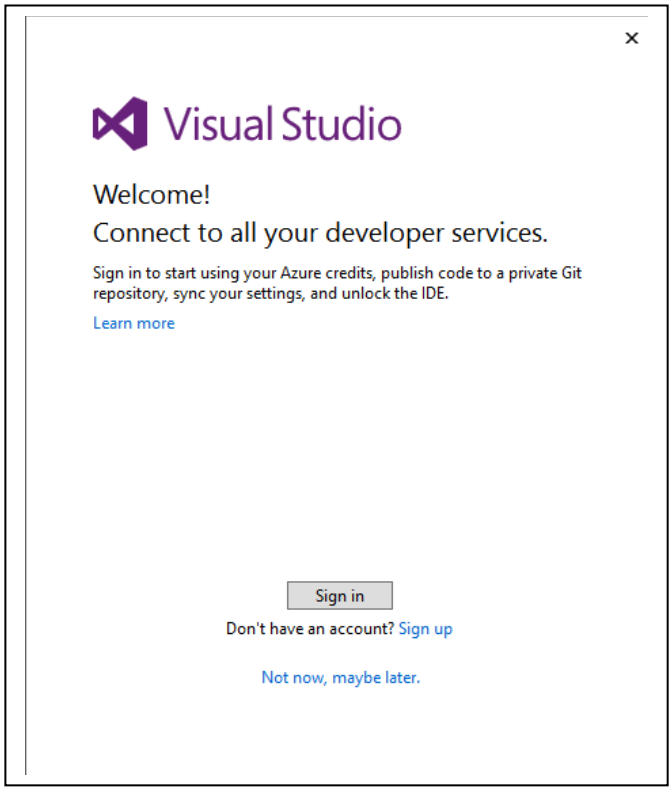

Figure 5

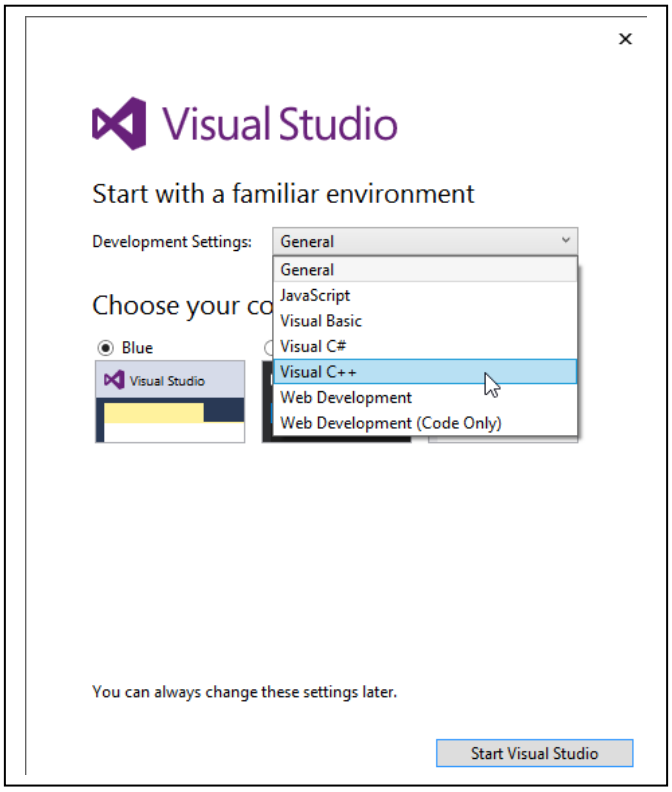

Figure 6

Continued on the next page...

Installing the IDE (Visual Studio 2017 Community Edition), *continued*

4. This is the main IDE window from which all future operations will be performed. It is currently displaying the **Start Page** and the **Solution Explorer** frame but will be used to display various other things as needed during program development. The solution explorer frame can be resized and/or drug to a new position if desired:

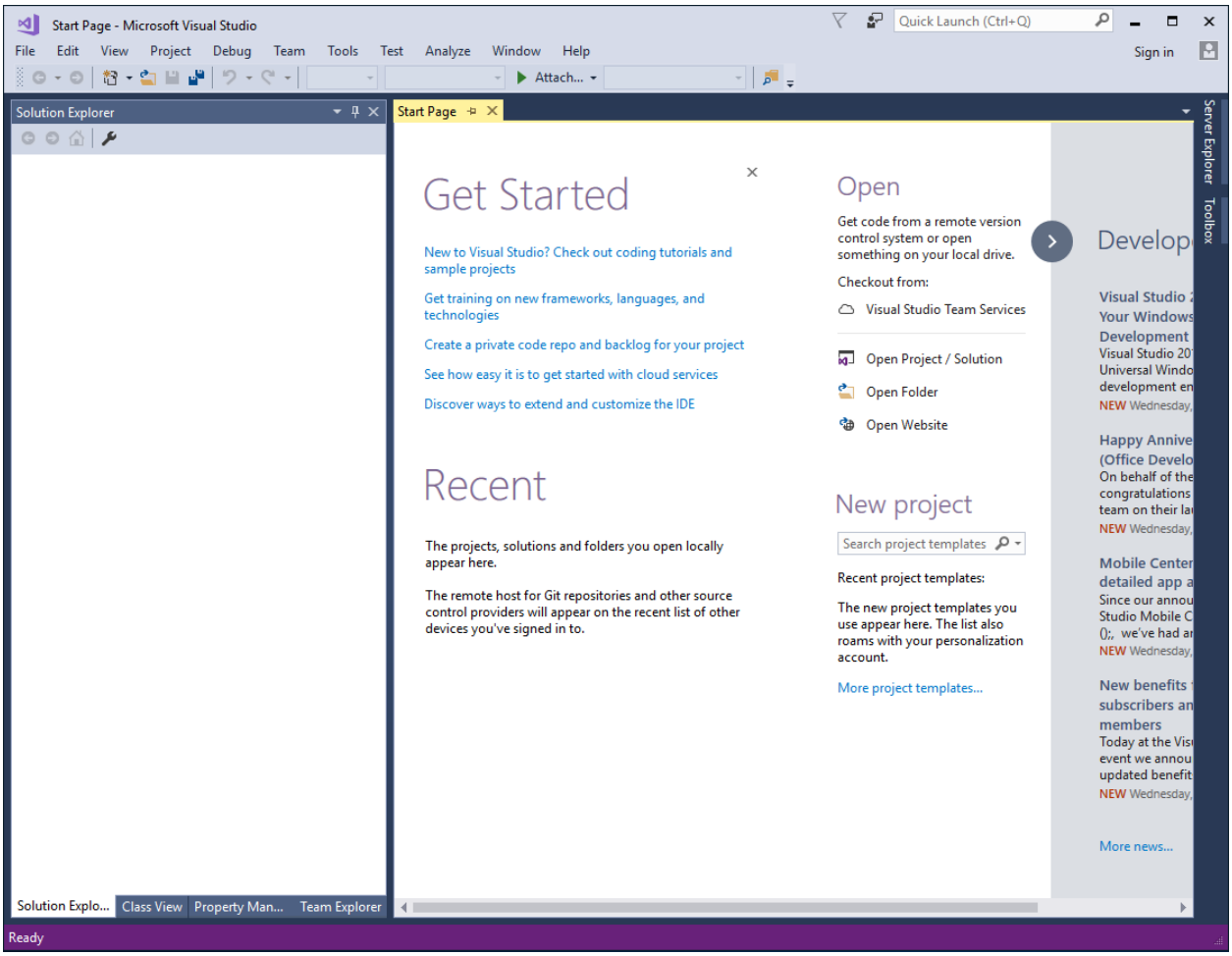

Figure 7

Changing the IDE Defaults

Although some students begin using their IDEs for the course assignments without carefully reading and following the suggestions in this document, my experience has been that most of them end up with much more wasted time and frustration than if they had just taken the time to make the changes in the first place.

Change the IDE's Text Editor Tab Settings

...affects the entire IDE. It is recommended that the text editor built into the IDE be used for editing all source code files. By default it is set to provide 4-column tab stops and to insert a "hard" tab character each time the keyboard Tab key is pressed. Although 4-column tab stops are fine (my personal preference is 3), the use of "hard" tabs is not allowed in this course and is discouraged in general because the actual size of a "hard" tab is interpreted differently by different editors and printers. Thus, files containing them have an undesirable editor/printer dependency that can result in some ugly surprises. It is usually preferable to have the text editor substitute an appropriate number of spaces whenever the tab key is pressed. Use the following procedure to make changes to the tab characteristics of the IDE's built in text editor:

1. From the IDE's main window menu select **Tools → Options...** This opens the *"Options"* dialog box (Figure 8);
2. In the left frame select **Text Editor → C/C++ → Tabs**
3. In the right frame set the **Tab size:** and **Indent size:** fields to the desired values and select the **Insert spaces** radio button;
4. Click ***OK*** to close the dialog box and accept the changes.

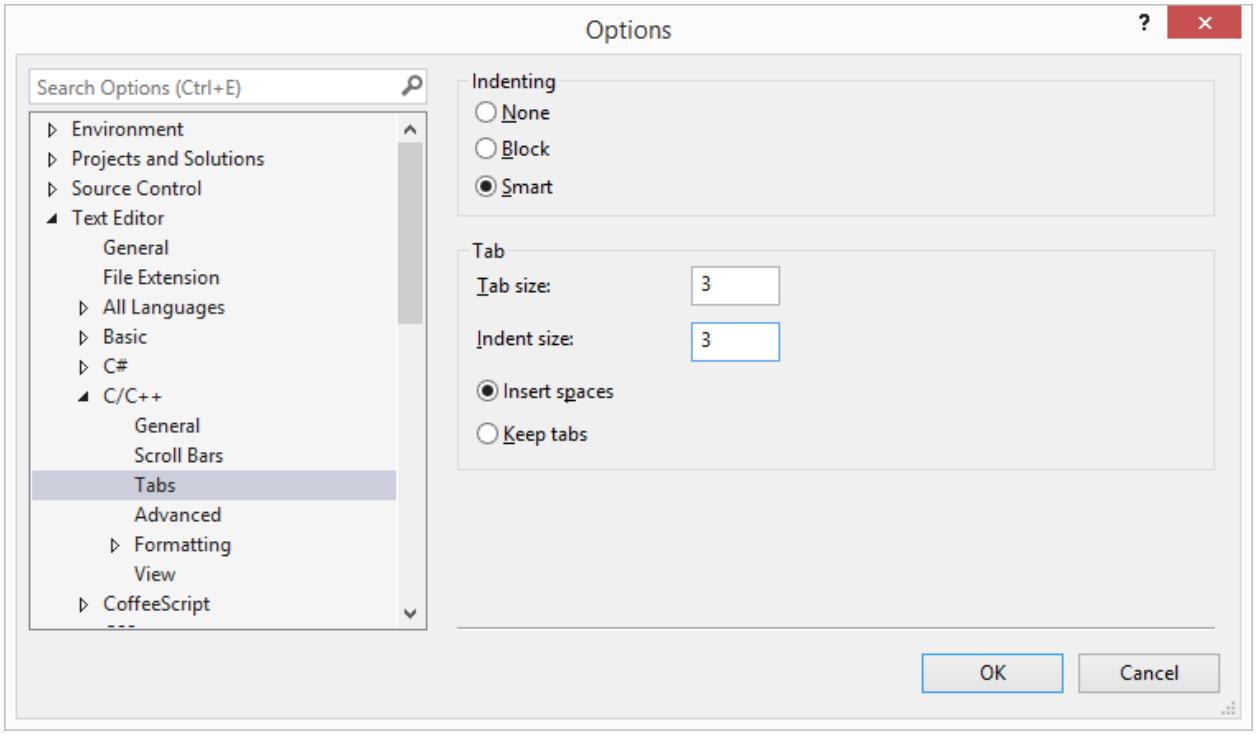

Figure 8

Removing Existing "Hard" Tabs

The previous procedure will not affect any "hard" tabs that are already in a file. These must either be replaced one-at-a-time manually, or by using the instructor-supplied "Hard Tab Removal Tool", or by selecting all lines in the file (Ctrl+A) and doing

Edit → Advanced → Untabify Selected Lines

or by some other means.

Enable the IDE's Text Editor Line Numbering

...affects the entire IDE. It is recommended that the text editor built into the IDE be used for editing all source code files. By default the editor does not display line numbers in such files, but line numbers are very useful when trying to match compiler error/warning messages to the code causing them and when referring to code in general. Use the following procedure to enable the display of line numbers. This change only affects how files are displayed in the text editor and does not affect the contents of the files themselves:

1. From the IDE's main window menu select **Tools → Options...** This opens the *"Options"* dialog box (Figure 9);
2. In the left frame select **Text Editor → C/C++ → General**
3. In the right frame check the **Line numbers** check box to enable the display of line numbers;
4. Click ***OK*** to close the dialog box and accept the changes.

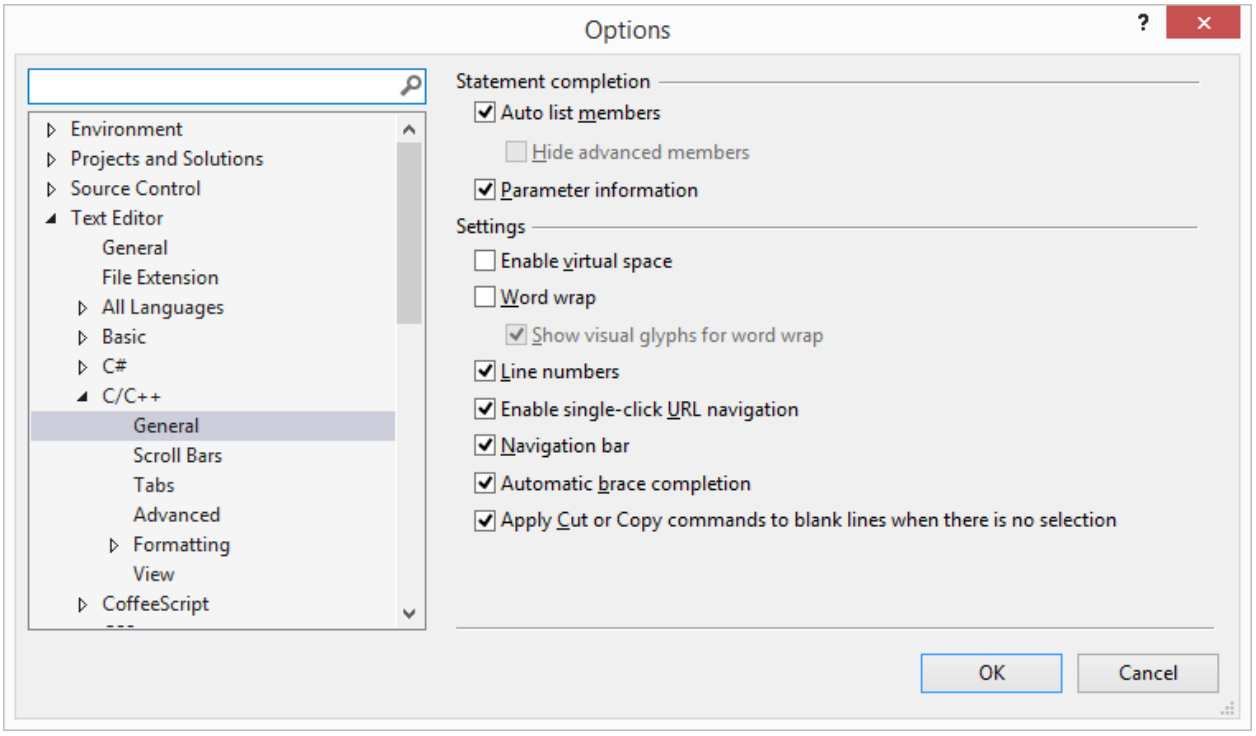

Figure 9

Change the IDE's Text Editor Default "case" Indenting

...affects the entire IDE. It is recommended that the text editor built into the IDE be used for editing all source code files. While the most common format for indenting the contents of `switch` statements is shown in Figure 10A, the format in Figure 10B is also fairly common:

```
switch (switchExpression)
{
    case 1:
        statement;
        etc;
        break;
    case 2:
        statement;
        etc;
        break;
    etc;
}
```

Figure 10A – Most Common

```
switch (switchExpression)
{
case 1:
    statement;
    etc;
    break;
case 2:
    statement;
    etc;
    break;
etc;
}
```

Figure 10B – Also Acceptable

Although Figure 10B represents Visual Studio's default setting it is easy to change if you prefer the more common format in Figure 10A. Making this change only affects new code. Existing code must be reformatted manually:

1. From the IDE's main window menu select **Tools → Options...** This opens the *"Options"* dialog box (Figure 11);
2. In the left frame select **Text Editor → C/C++ → Formatting → Indentation**
3. In the right frame ensure that both the **Indent case contents** and **Indent case labels** check boxes are checked;
4. Click ***OK*** to close the dialog box and accept the changes.

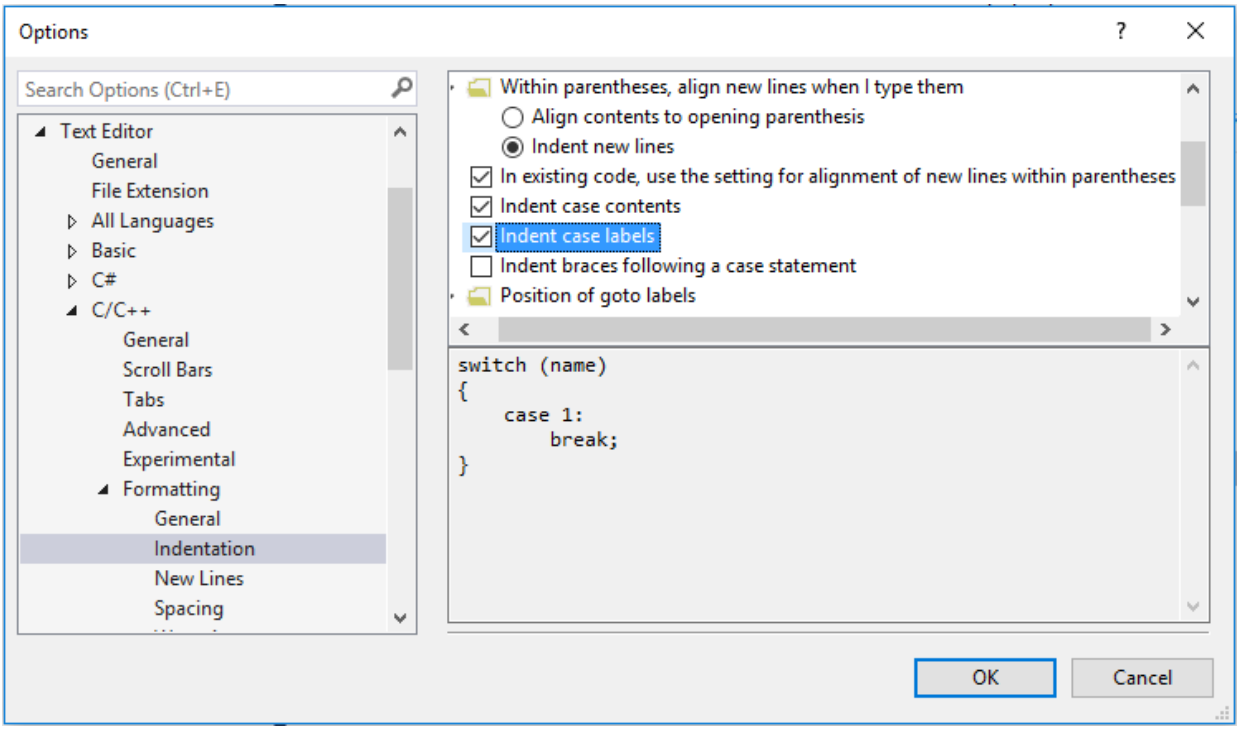

Figure 11

Creating a New Project

Most IDEs use the concept of a "project", which consists of all source code files, settings, and other resources necessary to create a working program. In addition, some IDEs such as Visual Studio also incorporate the concept of a "solution", which merely acts as a container in which multiple projects may be kept. This example assumes that you wish to create a new empty project named **MyPrograms** in folder **C:\Projects**, but you may use any legal name and location desired.

Unless you simply want to do unnecessary extra work do not create a new project for each exercise in this course. Instead, use the technique described in the section titled "Reusing the Same IDE Project for Every Exercise" on page 24 of this document.

1. From the IDE's main window menu select **File → New → Project...** This opens the *"New Project"* dialog box (Figure 12);
2. In the leftmost frame select **Installed → Visual C++ → General**
3. In middle frame select **Empty Project**
4. In the **Name:** field (near the bottom of the page), enter **MyPrograms**
5. In the **Location:** field, enter **C:\Projects**
6. Uncheck the *"Create directory for solution"* and *"Add to source control"* checkboxes;
7. Click the ***OK*** button. This closes the *"New Project"* dialog box and returns to the IDE's main window, which should look similar to what is shown in Figure 13 on the next page.

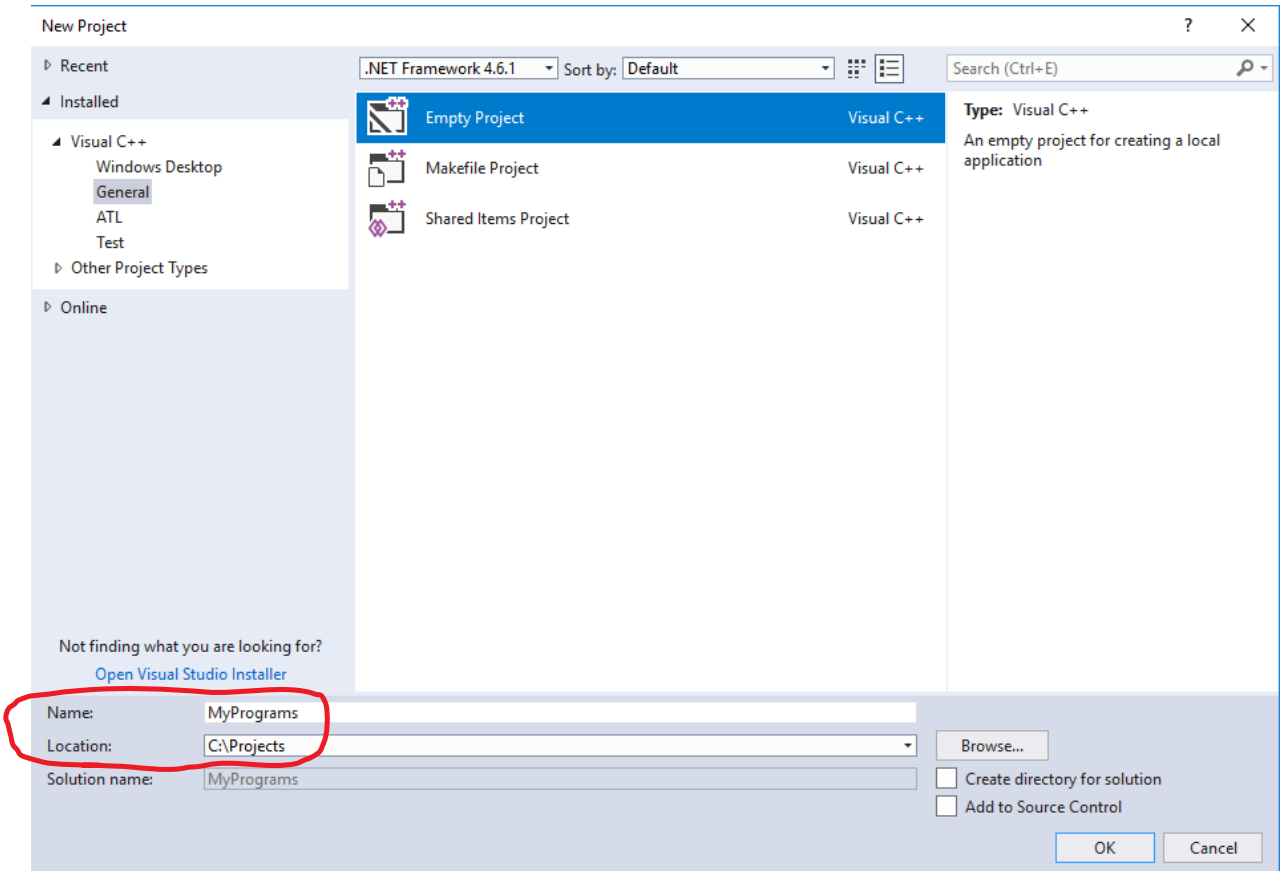

Figure 12

Continued on the next page...

Creating a New Project, continued

8. In this example the IDE's main window contains 4 child windows named "Class View", "Properties", "Output", and "Solution Explorer", **but** your main window may contain different child windows or possibly none at all.
9. The only child window we care about at this point is the one named "Solution Explorer" (if present), so merely close any others that are present.
10. If the "Solution Explorer" window is not present open it by doing **View → Solution Explorer**. That window is shown enlarged in Figure 14 on the next page.

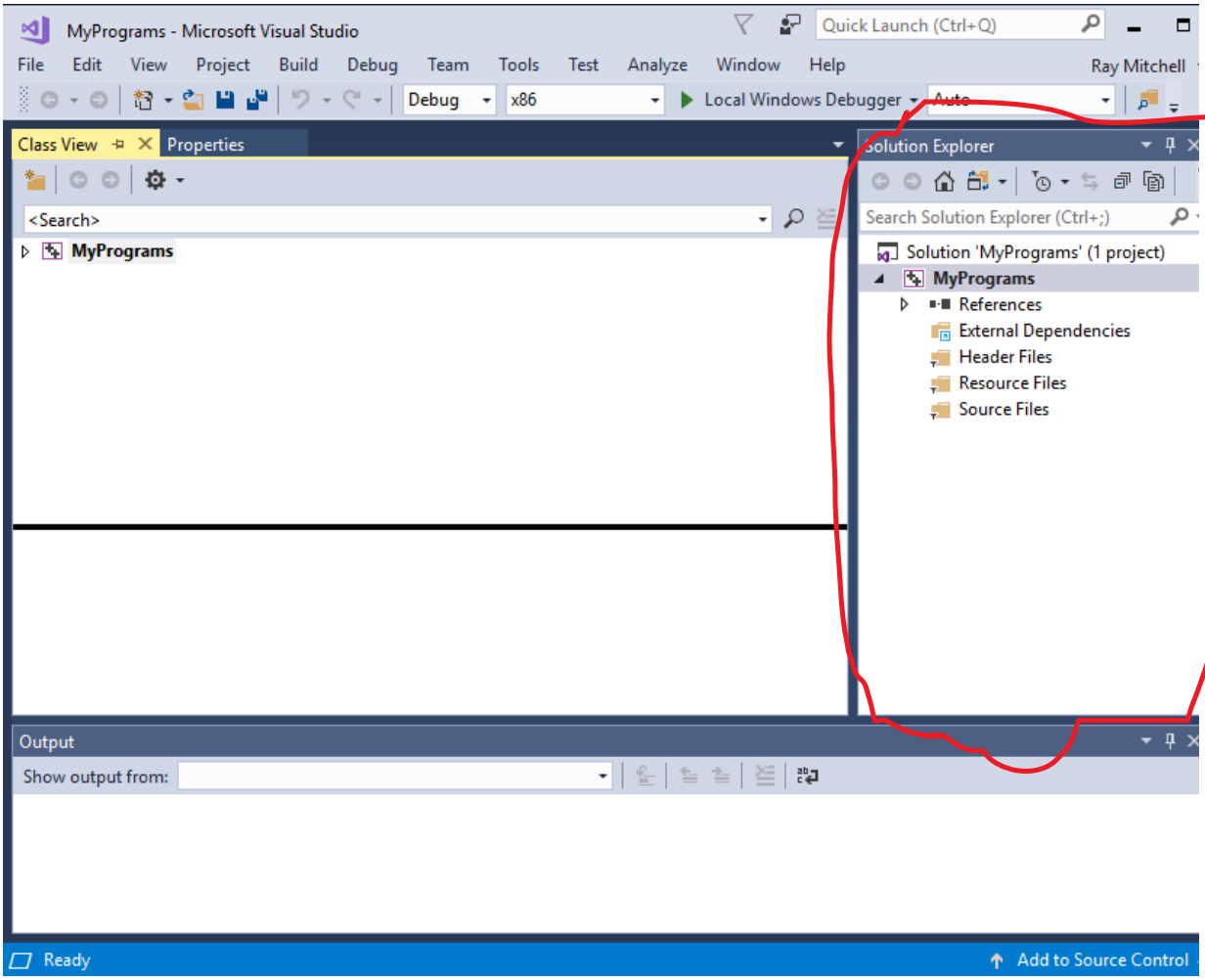

Figure 13

Your new empty project has now been created and you can proceed to the next page, which provides directions for adding one or more source code files to a project.

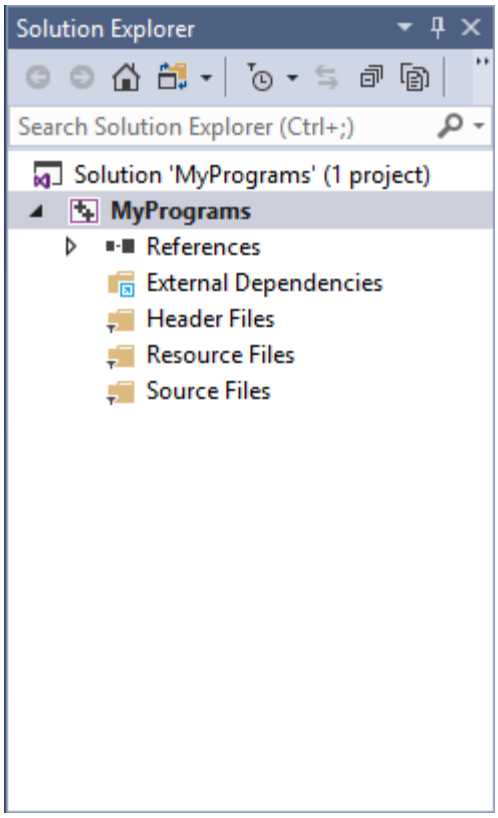

Figure 14

Adding One or More Source Code Files to a Project

Use this procedure to add any number of source code files to an existing project. The IDE will then be able to automatically compile and link these files together as appropriate to produce an executable program file. It is assumed that you have already followed the directions on the previous page and have created an appropriate project.

1. From the IDE's main window menu select **File → Open → Project/Solution...** This opens the *"Open Project"* dialog box.
2. Navigate to the directory containing your project;
3. Select the desired **.sln** file (i.e., **MyPrograms.sln**);
4. Click the **Open** button to open the project.

   Recommended Shortcuts:

   The IDE keeps a list of recently used projects/solutions and selecting from this list is usually quicker than the procedure described above. This list is available in **File → Recent Projects and Solutions** as well as on the left side of the *"Start Page"*. If the *"Start Page"* is not already open select **File → Start Page**. If not subsequently closed it will always be displayed first whenever the IDE opens and provides the quickest way to open an existing project.
5. Once the project is open the *"Solution Explorer"* window should be visible (Figure 14 on the previous page). If not, select **View → Solution Explorer**
6. In the *"Solution Explorer"* window, click the black arrowheads (if any) adjacent to the *"Source Files"* and *"Header Files"* folders to expand them and view the files that are already part of your project. Both of these will be empty in a new project if you have configured it properly. Ignore the *"External Dependencies"*, *"External Dependencies"*, and *"Resource Files"* folders.
7. Repeat the steps on the next page for each source code file you wish to add to your project, using an appropriate file name and extension for each:

Continued on the next page...

Adding One or More Source Code Files to a Project, continued

A. To create and add a new empty file right-click the *"Source Files"* or *"Header Files"* folder, as appropriate, in the *"Solution Explorer"* window, then:
   1) Select **Add → New Item...** This opens the *"Add New Item"* window (Figure 15).
   2) In the leftmost frame select **Installed → Visual C++ → Code**
   3) In the middle frame select **C++ File (.cpp)** or **Header File (.h)**, as appropriate. (**C++ File (.cpp)** is used for **.c** as well as **.cpp** files.)
   4) In the **Name:** field near the bottom of the page type the desired file name and extension (.c, .cpp, or .h). The name "Test.c" has been chosen for this example.
   5) Click the ***Add*** button.

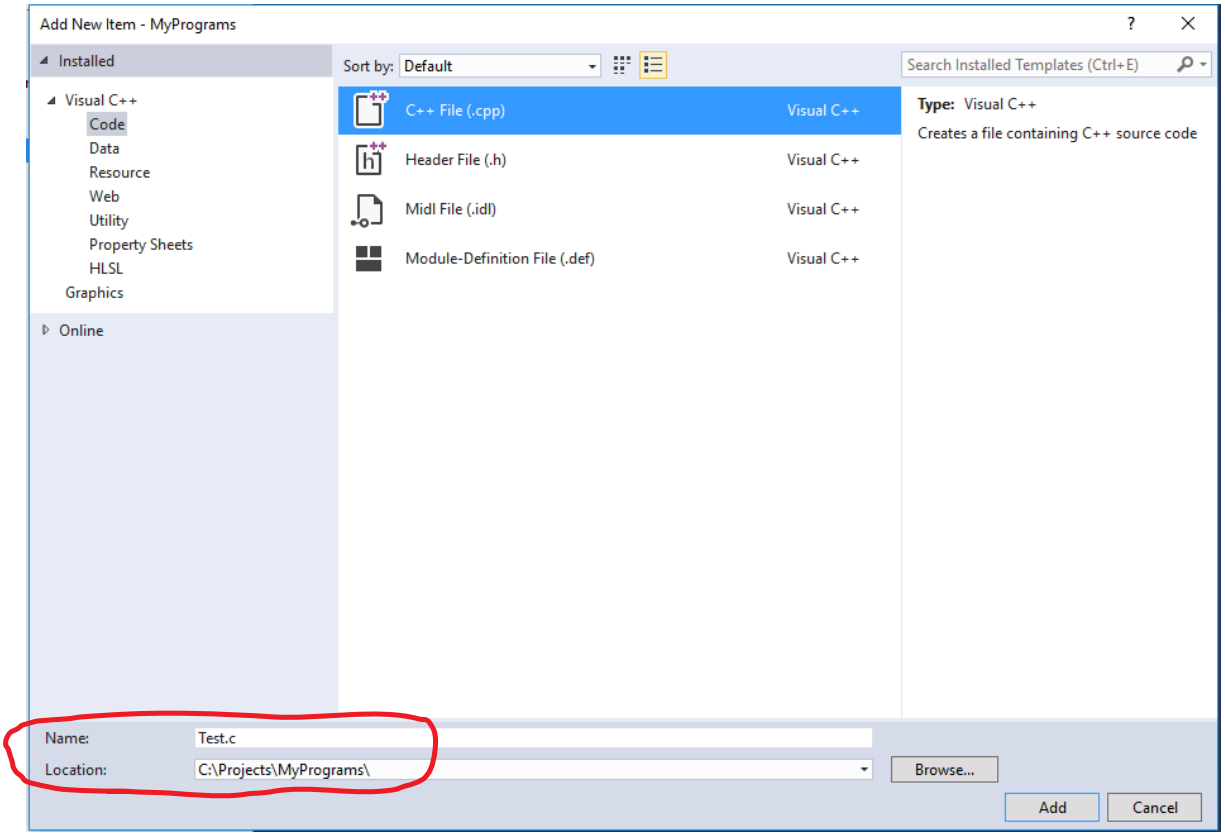

Figure 15

B. To add one or more existing files right-click the *"Source Files"* or *"Header Files"* folder, as appropriate, in the *"Solution Explorer"* window, then:
   1) Select **Add → Existing Item...** This opens the *"Add Existing Item"* window.
   2) In the *"Add Existing Item"* window, navigate to the directory containing the desired file and select that file. Multiple files may be selected by holding the *Ctrl* key depressed as you select them.
   3) Click the ***Add*** button.

You are now ready to compile and link your files. The compiler will automatically use all of the implementation files (.c and .cpp) listed in the *"Solution Explorer"* window. Header files are not compiled but are solely for inclusion in other files as needed. See page 23 for information on how to remove files from a project.

Changes to the Project Settings

While some settings affect the entire IDE, those on the following pages only affect an individual project. The examples assume that the Solution and Project are both named

MyPrograms

are located in

C:\Projects\MyPrograms

and there is a C implementation file in the project named

Test.c

which is located in

C:\Projects\MyPrograms

To view/change IDE settings that only affect a specific project:

1. Make sure the *"Solution Explorer"* window is visible in the IDE's main window (Figure 16). If it isn't, select **View → Solution Explorer** to expose it. (Note that I arbitrarily moved it to the right side of the window.)
2. Highlight the project (not the solution) you are using ;
3. Select **Project → Properties** (Figure 16) to open the *"Property Pages"* window as shown in Figure 17 on the next page.

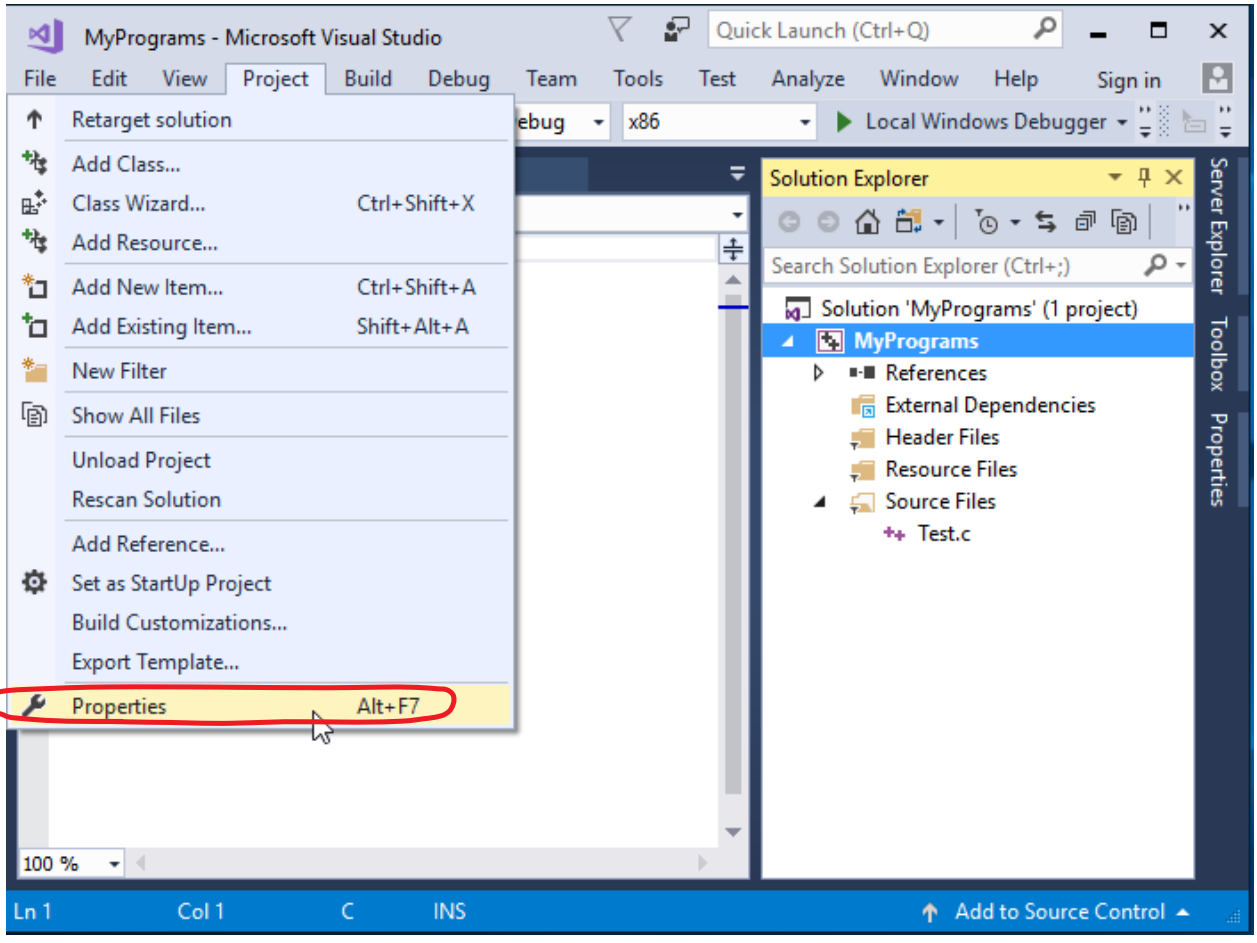

Figure 16

Continued on the next page...

Changes to the Project Settings, continued

The left frame of the *"Property Pages"* window (Figure 17) lists the various categories for which project settings changes can be made. Expand the **Configuration Properties** item if necessary to see them. I recommend you look through them just to get a feel for what's there, even if you don't understand most of it.

4. Select ***All Configurations*** from the ***Configuration:*** dropdown list in the upper-left. This configuration will be used for all remaining settings changes.

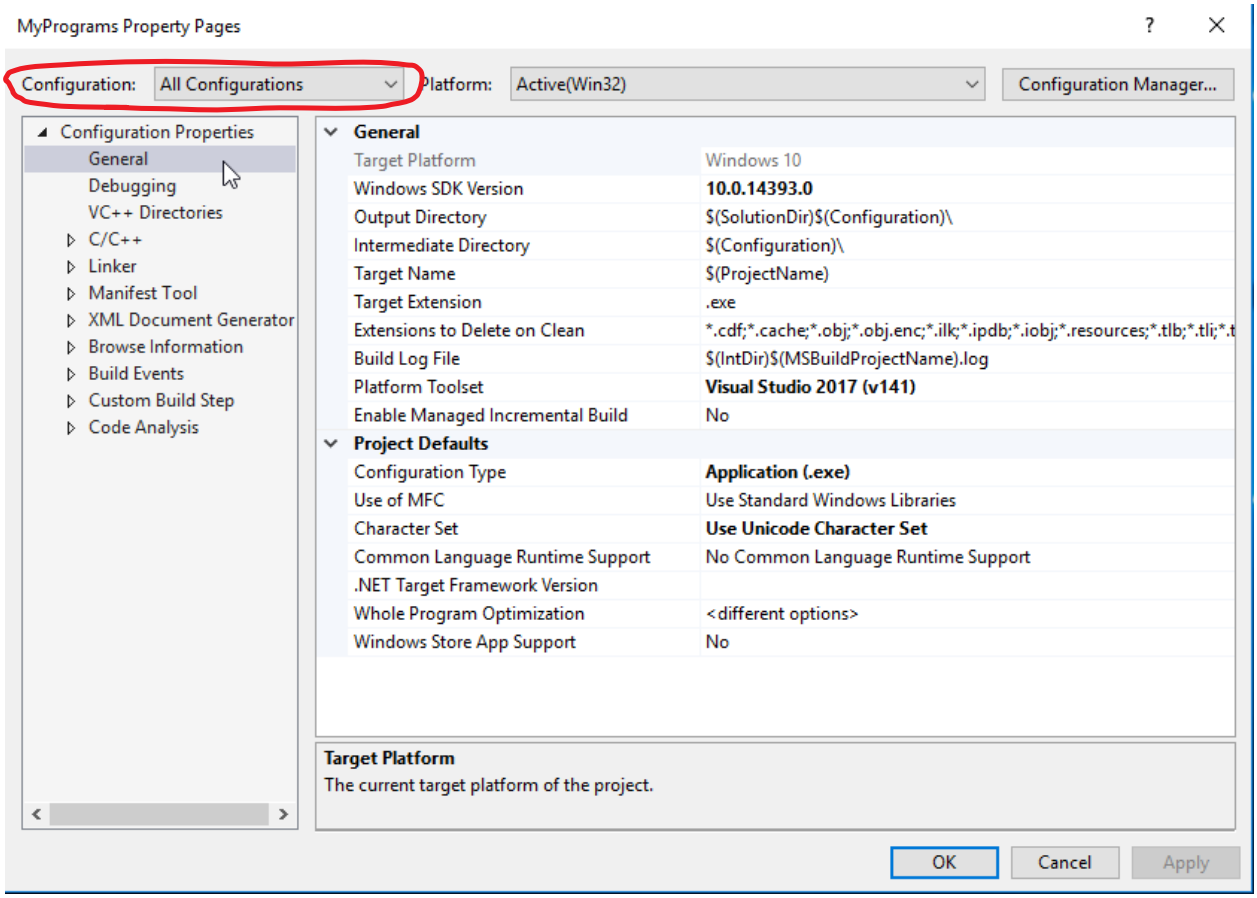

Figure 17

Determining/Changing the "Working Directory"

You will not need this information until you write a program that uses an instructor-supplied data file.
...affects an individual project, not the entire solution.

What is a "Working Directory": A program's "Working Directory" is the directory it uses for any files it opens or creates if their names are specified without a path. You must place any instructor-supplied data file(s) (.txt or .bin extensions) your program needs in that directory. Its default location differs between IDEs and operating systems and it's important to know where it is and how to change it.

Determining the Working Directory: If you have created your project by following the instructions previously given in this document a project file named **MyPrograms.vxcproj** will have been automatically created in a directory named **MyPrograms**. That directory is known as the "project directory" and by default is used as the working directory for any program run by that project. If you can't find it or putting files there doesn't seem to work, a simple way to empirically determine any program's working directory is to place the single statement:

```
puts(_getcwd(0, 1234));   // Remove before assignment checker submission
```

in the program's **main** function and ensure that the following inclusions are present:

```
#include <direct.h>       // Remove before assignment checker submission
#include <stdio.h>        // Remove if/when no longer needed
```

When the program executes **puts(_getcwd(0, 1234))** it will display the current working directory path on your screen and that's where you must put any needed instructor-supplied data file(s). If you are not satisfied with that location see the section on page 19 titled **Changing the Working Directory** for information on changing it.

Continued on the next page...

Determining/Changing the "Working Directory", continued

You will not need this information until you write a program that uses an instructor-supplied data file.

Changing the Working Directory: To change your project's working directory:

1. Open the *"Property Pages"* window using the technique shown on page 16.
2. Select **Configuration Properties → Debugging → Working Directory** (Figure 18). By default the working directory is **$(ProjectDirectory)**, which is a Visual Studio macro that represents the project's "project directory".
3. To change the directory you may either directly type the desired path into the entry field or browse for it by clicking the down-arrow to the right of that field and selecting **<Browse...>**.
4. Click ***OK*** to close the window and accept the change.
5. Your program will now use this directory for any "pathless" files it opens or creates.

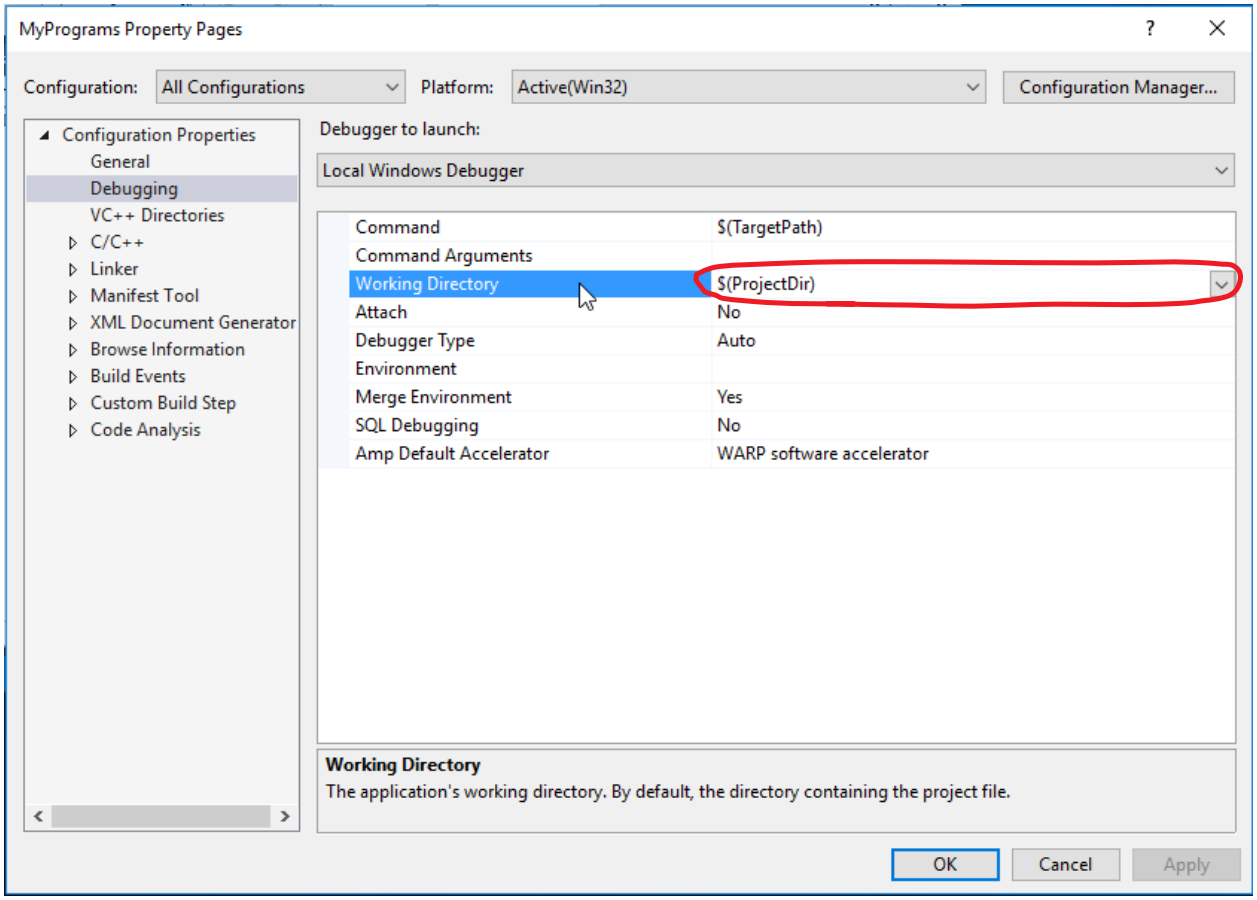

Figure 18

Setting the Warning Level

...affects an individual project, not the entire solution. The compiler's warning level should be set as high as possible. This will ensure that you receive notification of the maximum number of potential problems that the compiler is capable of detecting. The highest warming level is currently 4 and the following steps may be used to set it:

1. Open the *"Property Pages"* window using the technique shown on page 16.
2. Select **Configuration Properties → C/C++ → General → Warning Level** from the *"Property Pages"* window (Figure 19). **There must be at least one *.c* or *.cpp* file in the project for the C/C++ item to be visible.**
3. Click the down-arrow to the right of that field and select ***Level4 (/W4)***.
4. Click ***OK*** to close the window and accept the change.

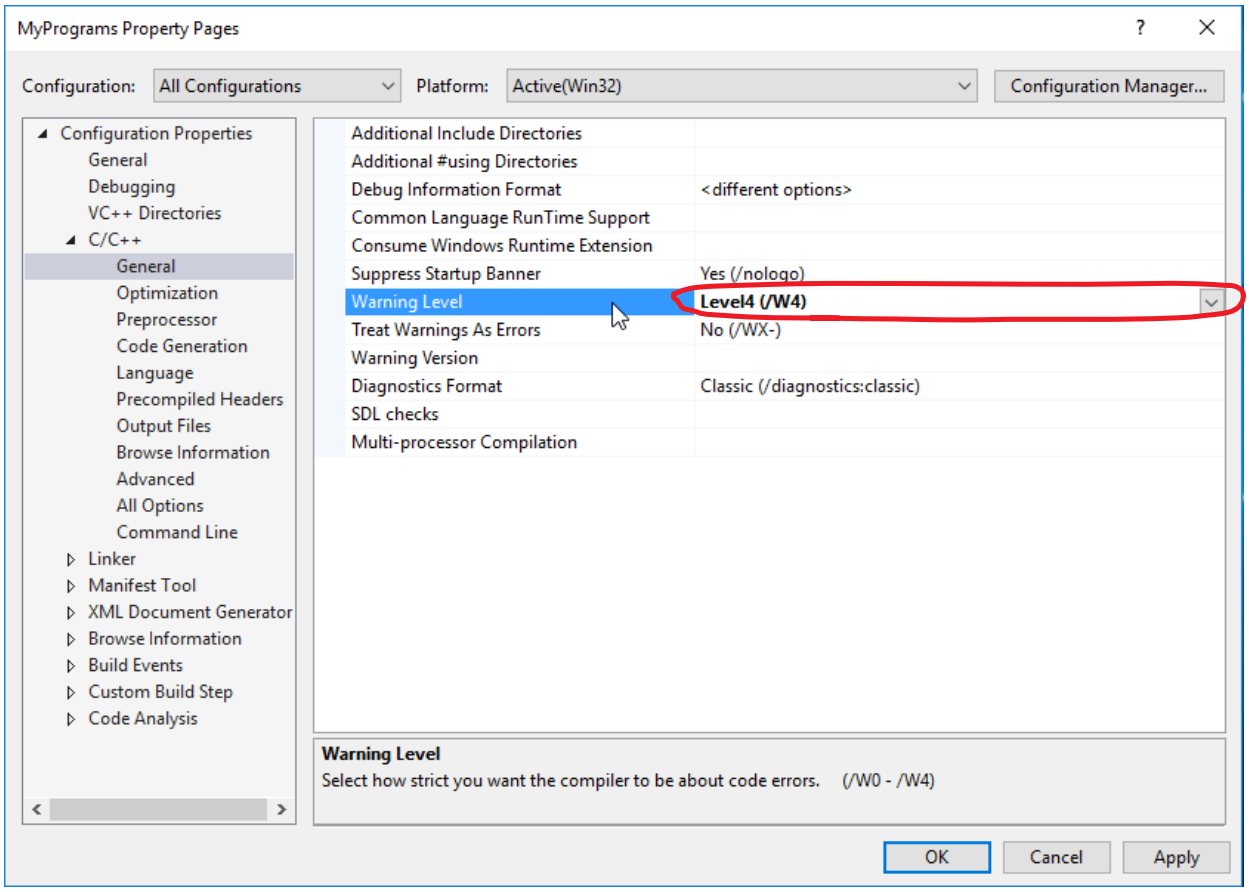

Figure 19

Disabling C4996 and Other Compiler Warnings

…affects an individual project, not the entire solution.

The Microsoft compilers generate a **C4996** warning when certain standard library functions it deems to be "unsafe" are used. Among these functions are **scanf**, **strcpy**, **fopen**, and others. While it is true that these functions can cause problems if used carelessly, Microsoft's solution is to use their own custom replacement functions instead. However, these replacements are not standard, are not supported by other compilers, and will result in compilation errors when the code is compiled with any compiler that doesn't support them. Thus, this warning should be disabled and the standard functions should be used:

1. Open the *"Property Pages"* window using the technique shown on page 16.
2. Select **Configuration Properties → C/C++ → Command Line** from the *"Property Pages"* window (Figure 20). **There must be at least one *.c* or *.cpp* file in the project for the C/C++ item to be visible.**
3. Type ***/wd4996*** in the ***Additional Options*** field, separating it from any other items that may already be there with a space.
4. You may add any additional warnings you wish to disable using the same technique.
5. Click ***OK*** to close the window and accept the change.

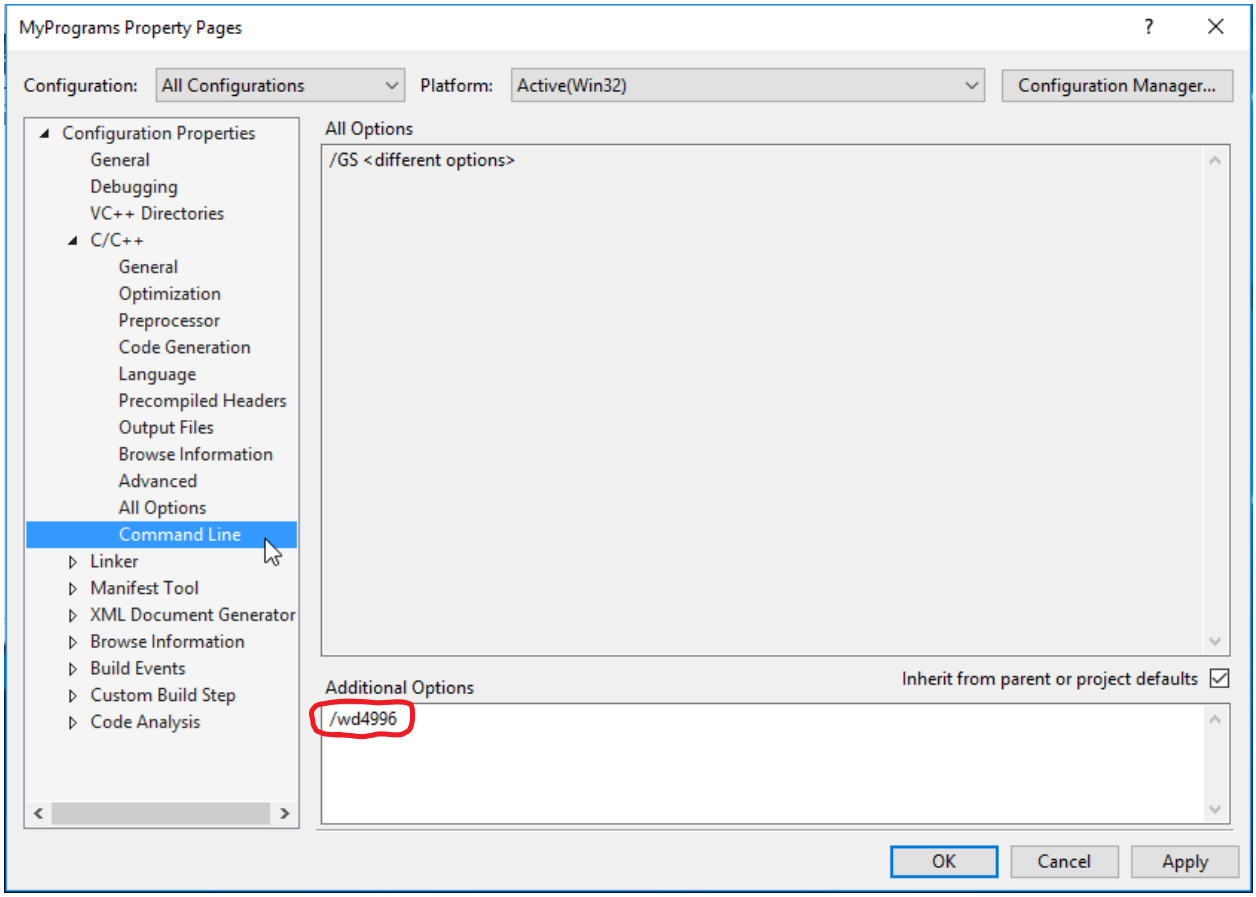

Figure 20

Setting the Project Subsystem

...affects an individual project, not the entire solution.

If the subsystem is not set correctly the program's Command Window will not stay open after the program is run using **Ctrl-F5**. To set the subsystem:

1. Open the *"Property Pages"* window using the technique shown on page 16.
2. Select **Configuration Properties → Linker → System** from the *"Property Pages"* window (Figure 21). **There must be at least one *.c* or *.cpp* file in the project for the C/C++ item to be visible.**
3. Select **Console (/SUBSYSTEM:CONSOLE)** from the drop-down menu to the right of the **SubSystem** item.
4. Click ***OK*** to close the window and accept the change.

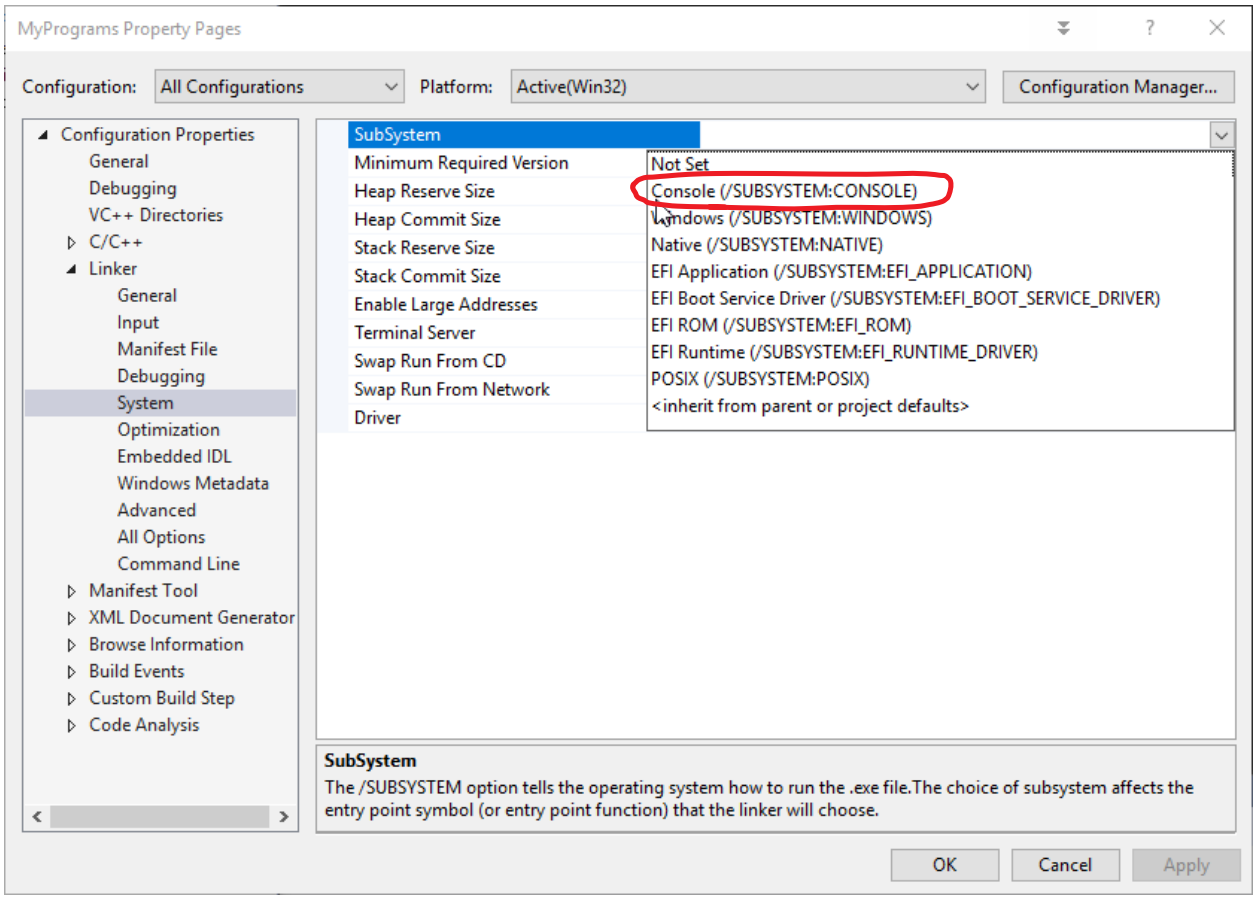

Figure 21

Removing Source Code Files from a Project

Use this procedure to exclude/remove any number of source code files from an existing project. There are two distinctly different ways to do this: If the ***Exclude From Project*** option is used the file(s) will merely be removed from the project's file list but will not be deleted from the computer itself. This will permit the file(s) to be added back into the project at a later time if desired. However, if the ***Remove*** option is chosen the file(s) will instead be removed from the project's file list and deleted from the computer itself.

1. In the *"Solution Explorer"* window, click the black arrowheads (if any) adjacent to the *"Source Files"* and *"Header Files"* folders to expand them and view the files that are already part of your project. Both of these will be empty in a new project.
2. Select the file(s) you wish to exclude or remove by clicking on them once. Multiple files may be selected by holding the *Ctrl* key depressed as you select them.
3. Right-click on the selection(s) to open the context menu (Figure 22).
4. Click either **Exclude From Project** or **Remove**, as desired, but be careful which one you choose.

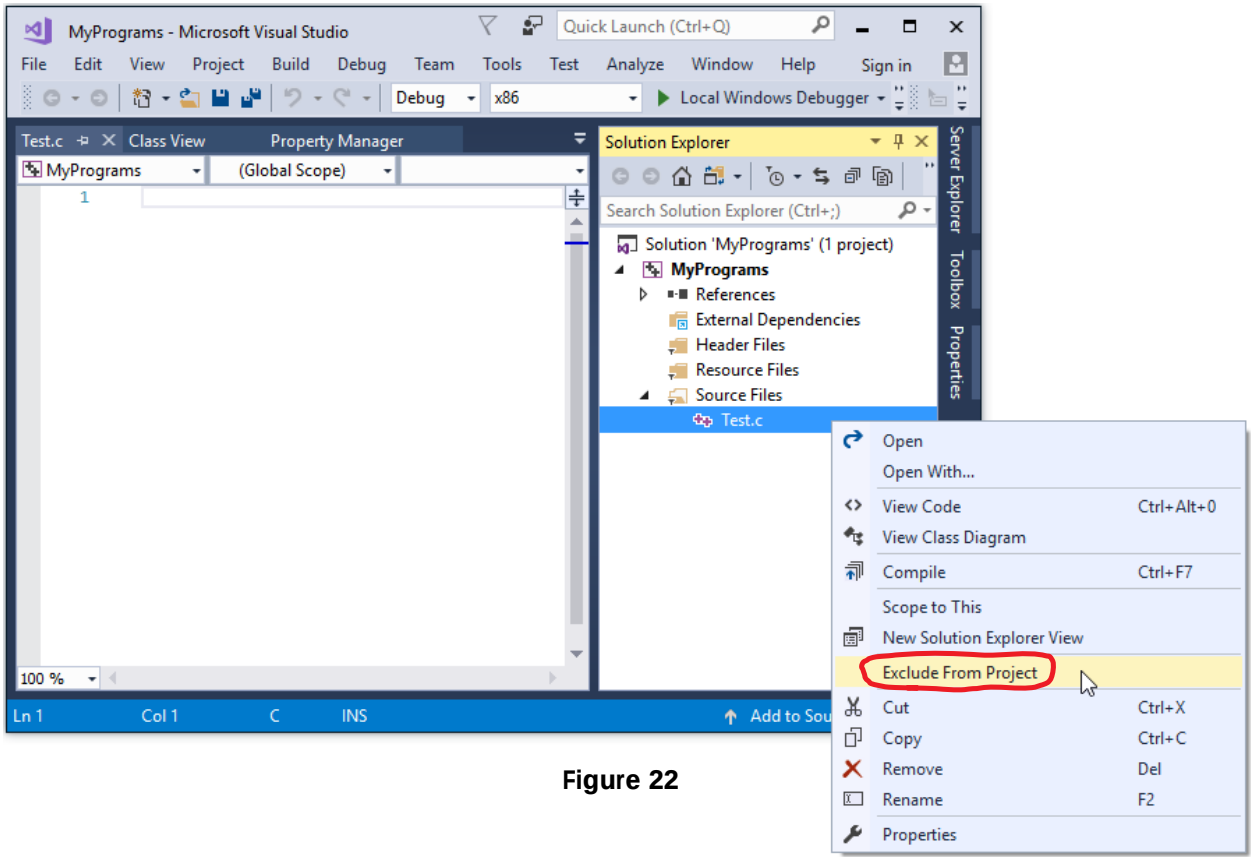

Figure 22

Reusing the Same Project for Every Exercise

Although it is possible to create a separate project for each programming exercise in this course, doing so is time consuming and unnecessary. Instead, I recommend that you use only one common project for all programming exercises. Then, when you are finished with a particular exercise, simply "Exclude" (but don't delete) its file(s) from the project and create new ones for the next exercise. By doing this the files for all exercises will remain in your project directory in case you need them again later. To exclude a file from a project but not delete it entirely merely right-click its name in the "Solution Explorer" window to bring up the context menu shown in Figure 23. Then click the **Exclude From Project** item. To add new files follow the procedure outlined on page14.

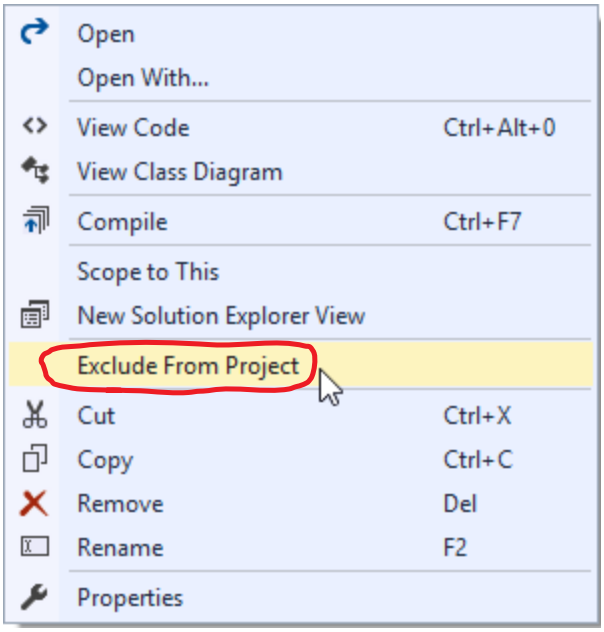

Figure 23

Compiling and Running a Program Using the IDE

Once you have created an IDE project, added your source code file(s), and written your code, you are then ready to compile everything into an executable program and run/test it.

Compiling (Building):

To compile your implementation files in into an executable program do **Build → Build Solution**

1. If the **Build** menu item is not available enable the **Build** toolbar by selecting **View → Toolbars → Build**. You may then use one of the build Icons from that toolbar.
2. If asked about building a "debug" file click the **Yes** button.
3. If errors or warnings occur, correct them and repeat the build.

Running:

Be sure you have made the change described on page 22 of this document before attempting to run your program from the IDE.

Running using method 1: Press Ctrl+F5 (same as **Debug → Start Without Debugging**)

The advantage of running your program using this method is that the command window that displays the results of your program run should stay open when the program terminates so you can capture it if desired. The disadvantage is that the IDE's integrated debugger cannot be.

Running using method 2: Press F5 (same as **Debug → Start Debugging**)

The advantage of running your program using this method is that it can be used in conjunction with the IDE's integrated debugger, which lets you step through your program one statement at a time or pause it at selected points of your choosing (breakpoints) so you can examine the values of variables. The disadvantage is that the command window will not stay open when the program terminates, but this issue can be easily handled in other ways.

Keeping the Command Window Open after a Program Runs

A frequent question concerns how to keep a program's command window open after that program has terminated so that a copy of it can be captured if desired. The answer is usually simple: Use method 1 above to run the program, which will usually result in the command window staying open until another key is pressed. Another approach is to place a "breakpoint" on the line containing the `return` statement in the `main` function, then run the program using method 2 above. Breakpoints are discussed in the next section but recapping this briefly, a "breakpoint" is a point in the program where execution will pause. One way to set a breakpoint is to place your cursor on the desired line and press **F9**.

If for some reason method 1 above doesn't work and you don't want to use method 2 with a breakpoint, you can instead use the worst method of all, which consists of placing one or more calls to the `getchar` function (in C) or the `cin.get` function (in C++) just before the `return` statement in the `main` function, but this should not be necessary and is a bad practice in general since it will cause the program to always pause, even when you run it later outside the IDE and don't want it to. Calling **system("Pause")** should never be done since it is not portable.

Using the IDE's Debugger

Overview: An IDE's built-in debugger is a powerful and easy to use tool that allows programmers to pause their programs at arbitrary points called "breakpoints", step through their code one statement at a time, and examine the values of variables and other expressions. These three things alone allow many program bugs to be found more quickly and easily than with other methods. Although the thought of using a debugger can be intimidating to beginning programmers, the fact is learning the basics is trivial, yet provides an invaluable debugging aid. This document just covers those basics and complete documentation can be found by searching the Web.

Controlling Debugging: Debugging can be controlled through your choice of menu selections, icon clicks, and/or keyboard shortcuts. Figures 24A and 24B show a portion of Visual Studio's Debug menu before and after the program starts running, respectively. This menu is useful when you're first learning and it also shows you some equivalent keyboard shortcuts. Figures 25A and 25B show icons for starting and continuing the program for debugging, respectively. And Figure 26 is a set of miscellaneous debugging icons, which are only visible while debugging. All of these icons are located just below the main window's title bar.

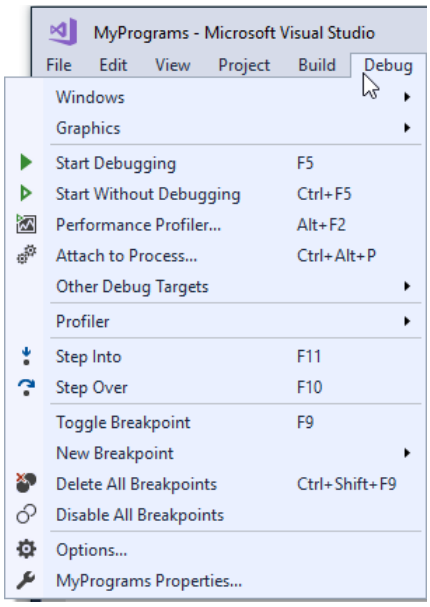

Figure 24A

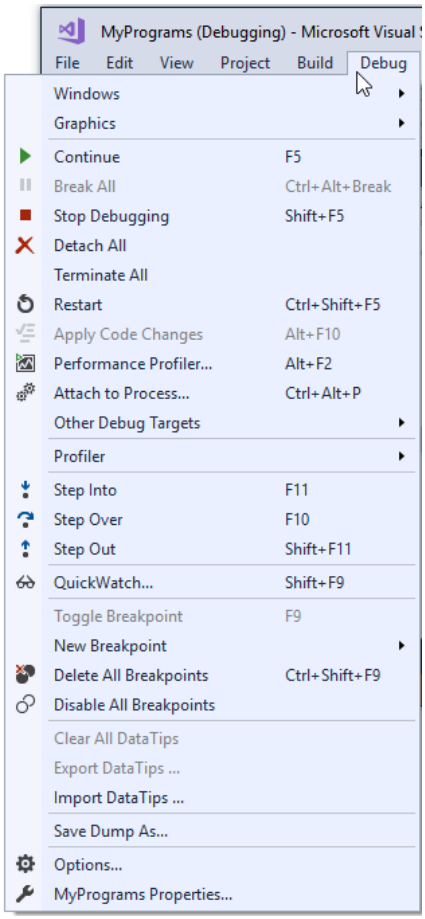

Figure 24B

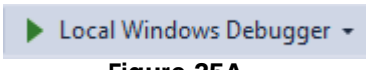

Figure 25A

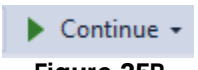

Figure 25B

Figure 26

Using the IDE's Debugger, continued

The Program: Figure 27 below shows a typical program in a Visual Studio window. Functionally it compares two variables and prints their value if they are equal, prompts the user to enter a value, reads and prints that value, executes a printing loop, computes a new value to be printed, then prints that value. We will set some breakpoints to pause the program, then explain single-stepping through the code while examining the values of the variables.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     int inValue = 9, outValue = 14, count;
6
7     if (inValue == outValue)
8        printf("outValue = %d\n", outValue);
9
10    printf("Enter a value: ");
11    scanf("%d", &inValue);
12    printf("inValue = %d\n", inValue);
13
14    for (count = 0; count < inValue; ++count)
15       printf("count = %d\n", count);
16
17    outValue = inValue * 2;
18    printf("outValue = %d\n", outValue);
19
20    return 0;
21 }
22
```

Figure 27

Setting Breakpoints: A breakpoint may be placed at any line containing a code statement and the program will pause when it is reached. The simplest way to set a breakpoint is to click in the gray margin to the left of the line number, or you can place the cursor on that line and either press **F9** or select from the **Debug** menu. Five breakpoints have been set in Figure 28, as indicated by the red dots.

IMPORTANT: When a pause occurs the statement on that line WILL NOT yet have been executed. To view the effects of that statement either single-step (page 30) to the next statement or set a breakpoint (page 27) on that next statement and continue (page 30).

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     int inValue = 9, outValue = 14, count;
6
7     if (inValue == outValue)
8        printf("outValue = %d\n", outValue);
9
10    printf("Enter a value: ");
11    scanf("%d", &inValue);
12    printf("inValue = %d\n", inValue);
13
14    for (count = 0; count < inValue; ++count)
15       printf("count = %d\n", count);
16
17    outValue = inValue * 2;
18    printf("outValue = %d\n", outValue);
19
20    return 0;
21 }
22
```

Figure 28

Starting Debugging: An easy way to start the program is to press the **F5** key, but you may instead click the **Local Windows Debugger** icon (Figure 25A) or select **Start Debugging** from the **Debug** menu (Figure 24A). A yellow arrowhead in the gray margin next to a line indicates a debugging pause, at which point the values of variables and other expressions can be examined.

Figure 29 below shows that the program does not pause at the first breakpoint on line 8, but instead pauses at line 10. This is because variables **inValue** and **outValue** are not equal, so the statement on line 8 is never reached. Breakpoints may be added, deleted, or disabled at any time. The easiest way to delete them individually is by clicking the red dots in the left margin or placing the cursor on that line and pressing **F9**. You can easily delete them all by pressing **Ctrl+Shift+F9**.

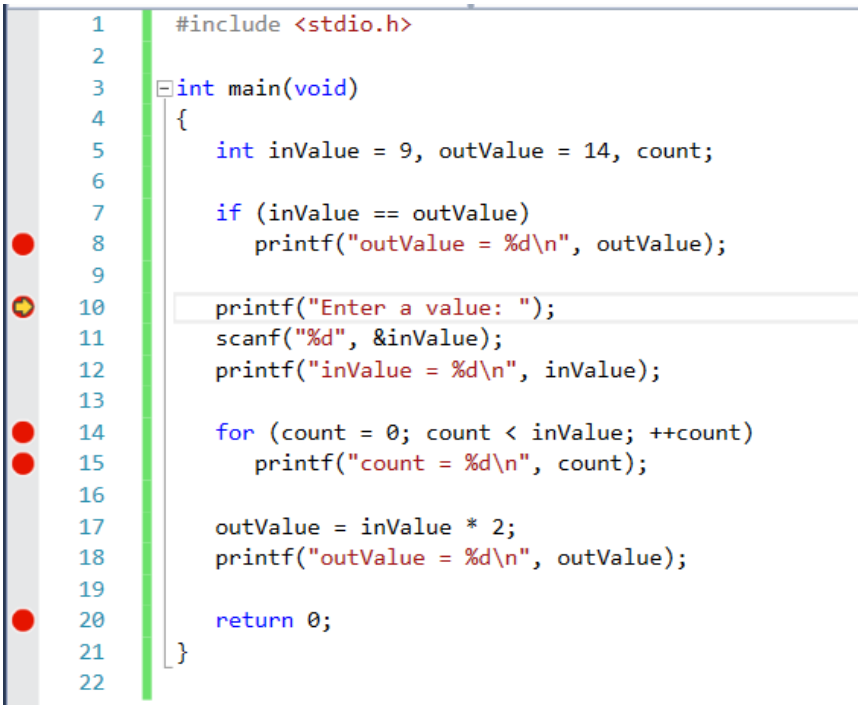

Figure 29

Examining Variables and Other Expressions: Whenever a program is paused at a breakpoint or after a single-step you may view the value of any variable or other expression currently in scope.

IMPORTANT: When a pause occurs the statement on that line WILL NOT yet have been executed. To view the effects of that statement either single-step (page 30) to the next statement or set a breakpoint (page 27) on that next statement and continue (page 30).

To view a variable merely hover the mouse pointer over it. To view an entire expression you must first select it, then hover the mouse pointer over that selection. To view more than one variable or expression simultaneously without having to hover the mouse pointer, right-click on the variable or selected expression whose value you want to view and select **Add Watch** from the resulting context menu. This will open a "watch" window containing that variable or expression along with any other variables or expressions you previously watched, as shown in Figure 30.

Once the watch window is open it will automatically open every time you debug the program unless you explicitly close it. To add additional variables or expressions you may either type them into the watch window manually, drag them in from your code, or right-click on them in your code and select **Add Watch** from the resulting context menu. Watch window entries may be removed by selecting them and pressing **Delete**.

Note that in this example variables **inValue** and **outValue** contain the values to which they were initialized, whereas variable **count** has an arbitrary "garbage" value because it wasn't initialized. As you progress through the code any changes to watched values will be displayed during each pause, thereby allowing you to see how your code is affecting them. The integer numeric values in this example are displayed in decimal, but by right-clicking anywhere on the watch window you may change it to hexadecimal instead. Do the same to change it back to decimal.

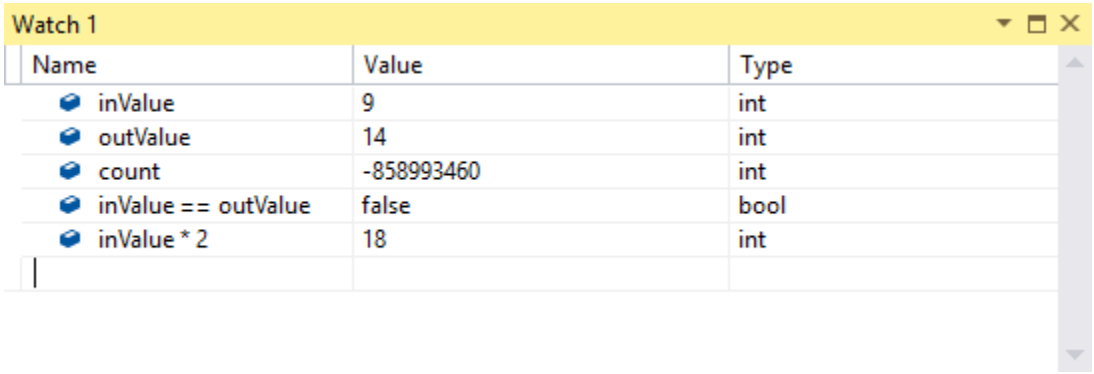

Figure 30

Terminating, Continuing, and Single-Stepping:
Whenever a program is paused during debugging you have three main choices. You may either

1. terminate the program;
2. resume program execution to whichever comes first of the next breakpoint, user input, or program end (known as "continuing");
3. execute only the next statement then pause again (known as "single-stepping").

Terminating:
Press **Shift+F5** or click the **Stop Debugging** icon (1st icon in Figure 26) or select the **Stop Debugging** item in the **Debug** menu.

Continuing:
Press **F5** or click the **Continue** icon (Figure 25B) or select the **Continue** item in the **Debug** menu. It is important to note that if there is any code that requires user input, such as the **scanf** on line 11 in this example, the program will stop and wait for that input just as it does when not debugging. This does not represent a debugging pause so no variables/expressions can be examined and the program cannot be continued or single-stepped until the user provides that input.

Single-Stepping:
Single-stepping is an extremely useful tool for progressing through your code one statement at a time to see its effect on your variables and input/output operations. There are three basic stepping operations:

1. **Step Over** – **F10** or the **Step Over** icon (6th icon in Figure 26) or the **Step Over** item in the **Debug** menu.
   This is the most common stepping operation and simply executes the code on the current line and pauses on the next. It's important to note that if there is any code that requires user input, such as the **scanf** on line 11 in this example, the program will stop and wait for that input just as it does when not debugging. This does not represent a debugging pause so no variables/expressions can be examined and the program cannot be continued or single-stepped until the user provides that input.

2. **Step Into** – **F11** or the **Step Into** icon (5th icon in Figure 26) or the **Step Into** item in the **Debug** menu.
   For statements that contain one or more function calls this will allow you to into the functions' code so you can debug it or merely look at it. You should normally not step into library functions since their source code is usually not available, but you may want to step into the code for functions you write if you are trying to debug them.

3. **Step Out** – **Shift+F11** or the **Step Out** icon (7th icon in Figure 26) or the **Step Out** item in the **Debug** menu.
   If you have stepped into a function by mistake or simply want to complete the function you are currently in for any reason, do a step out. If there are any breakpoints or user inputs in that function, however, the program will still pause/stop at them.

Exercise Command Line Argument Requirements

Some exercises may require the use of command line I/O redirection (note 4.2), command line arguments (note 8.3), or both. This will always be explicitly stated or unambiguously implied in the individual requirements for those exercises.

What is a Command Line?

A command line consists of the command(s) necessary to run a program. It consists of one or more space-separated strings, where the first string specifies the name of the program file to be executed and any additional strings provide information needed by the program itself, the operating system, or both. Each string not pertaining to I/O redirection, including the name of the program file itself, is known as a command line "argument" and any C or C++ program can easily determine the number of and values of these arguments by inspecting the `argc` and `argv` parameters of function `main`, respectively. Good practice dictates that when `argc` is present it always be used for either command line argument count validation, command line argument processing, or both. Information pertaining to I/O redirection is used by the operating system and is never part of `argc` or `argv`.

Command Line Examples

If a program is to be executed from within the IDE always omit the name of the executable file from the command line argument list since the IDE supplies it automatically. In the following examples the name of the executable file is assumed to be `MyPgm.exe`, which means that `argv[0]` will always represent the string `MyPgm.exe`, typically with the entire directory path prepended to it:

If the non-IDE command line is `MyPgm.exe box set price`
or if the IDE command line is `box set price`
the result will be:
`argc` = 4; `argv[1]` = box; `argv[2]` = `set`; `argv[3]` = `price`; and there is no I/O redirection

If the non-IDE command line is `MyPgm.exe box > set < price`
or if the IDE command line is `box > set < price`
the result will be:
`argc` = 2; `argv[1]` = box; stdout will be written to file `set`; stdin will be read from file `price`

Command Line Arguments Containing Spaces

Sometimes command line arguments containing spaces are needed, such as in certain file/directory names, grammatical phrases, etc. Let's assume we wish to represent the following three phrases as three individual command line arguments:

The old
gray mare
is not

If simply placed together on the command line they would be erroneously interpreted as six separate arguments rather than three:

The old gray mare is not

Although operating system dependent, the solution is simple and can even be used if desired when no spaces are present. The first of the following techniques is the most common but if it doesn't work it's worth trying the next two:

"The old" "gray mare" "is not"	double-quotes around each argument
'The old' 'gray mare' 'is not'	single-quotes around each argument
The\ old gray\ mare is\ not	escape the desired whitespace(s)

Specifying Command Line Arguments from within the IDE

...affects an individual project, not the entire solution. Assume that in addition to the name of the executable program itself, which is always required and which this example will assume is ***Test.exe***, two additional command line arguments of ***File1.txt*** and ***Hello world!*** are needed. If running the program from outside the IDE, such as from a command window, an icon, or a batch file, the required command line would typically be

Test.exe File1.txt "Hello world!"

But from within the IDE it would only be

File1.txt Hello world!"

since the IDE automatically supplies the executable file name as the first argument. In either case:

`argv[0]` would represent string ***Test.exe*** (typically with a prepended directory path);
`argv[1]` would represent string ***File1.txt***;
`argv[2]` would represent space-containing string ***Hello world!***;

To place the required arguments on the command line from within the IDE do the following:

1. Open the ***Property Pages*** window (Figure 31);
2. Select ***All Configurations*** from the ***Configuration:*** dropdown list in the upper-left of the window;
3. Expand the ***Configuration Properties*** category in the left frame;
4. Highlight the ***Debugging*** item, which will expose the selections shown in the right frame;
5. Select ***Command Arguments***, then enter: ***File1.txt "Hello world!"***
   a. Note: To have a string containing spaces to serve as a single argument, put double quotes around it. ie., ***"Mary Smith"*** instead of ***Mary Smith***
6. Click ***OK*** to close the window and accept the change, then run the program as usual.

If it is also desired to incorporate I/O redirection (note 4.2) into the command line, simply appending ***< InputFile.txt*** to the command arguments shown below, for example, would cause all reads done by `scanf`, `getchar`, `cin >>`, `cin.get`, etc. to come from file ***InputFile.txt*** rather than the keyboard.

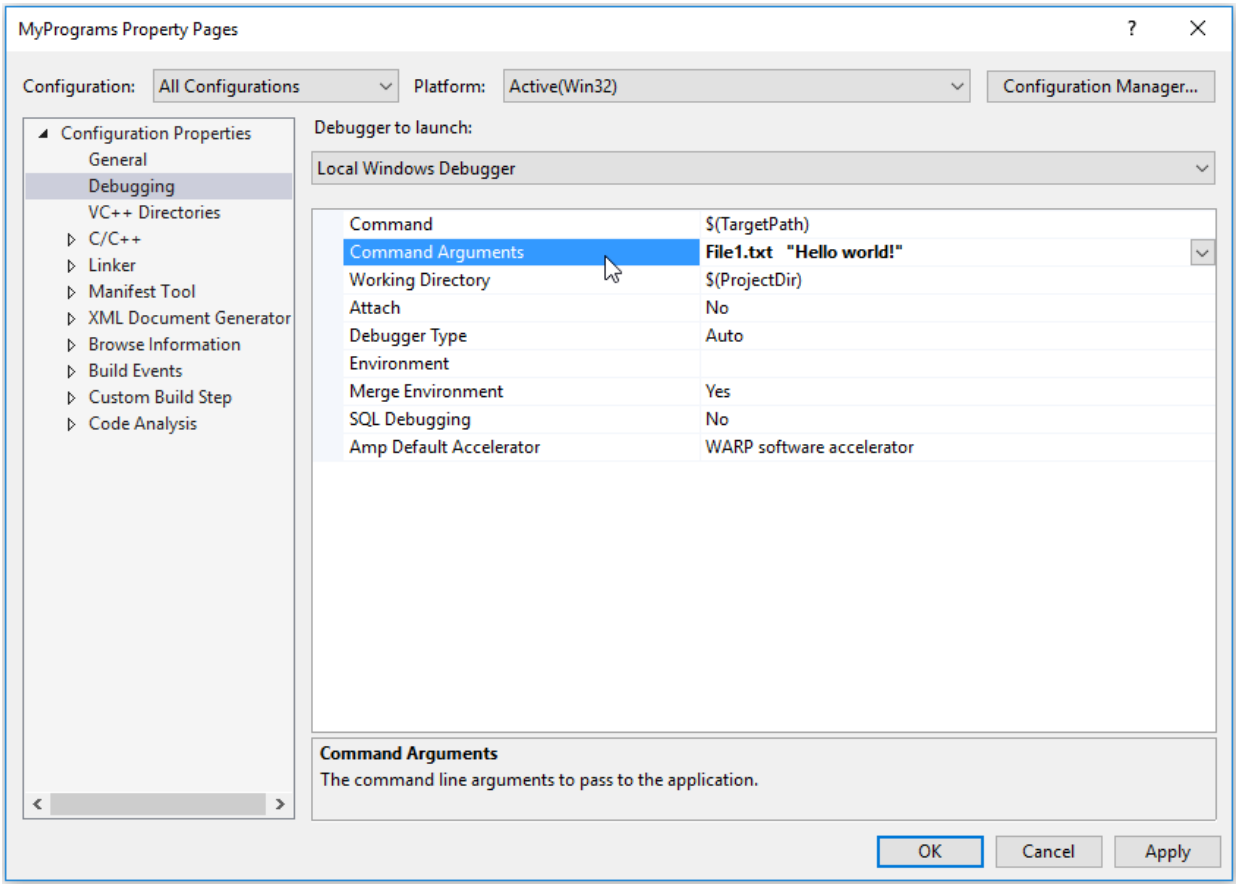

Figure 31