

Table of Contents

Introduction – How Programs Are Created and Run.....	2
The IDE (Integrated Development Environment)	3
Required Change to the IDE's Global Text Editor Tab Settings.....	4
Removing Existing "Hard" Tabs.....	4
Suggested Change to the IDE's Global Text Editor Line Numbering.....	5
Suggested Change to the IDE's Global Compiler Settings	6
Creating a New Project.....	7
Adding One or More Source Code Files to a Project.....	10
Changes to the Project Settings	13
Determining/Changing a Project's "Working Directory".....	15
What is a "Working Directory"	15
Determining the "Working Directory"	15
Changing the "Working Directory"	16
Removing Source Code Files from a Project.....	17
Reusing the Same Project for Every Exercise	18
Compiling and Running a Program Using the IDE.....	18
Keeping the Command Window Open after a Program Runs	19
Using the IDE's Debugger	20
Exercise Command Line Argument Requirements	25
Specifying Command Line Arguments from within the IDE.....	26

Introduction – How Programs Are Created and Run

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

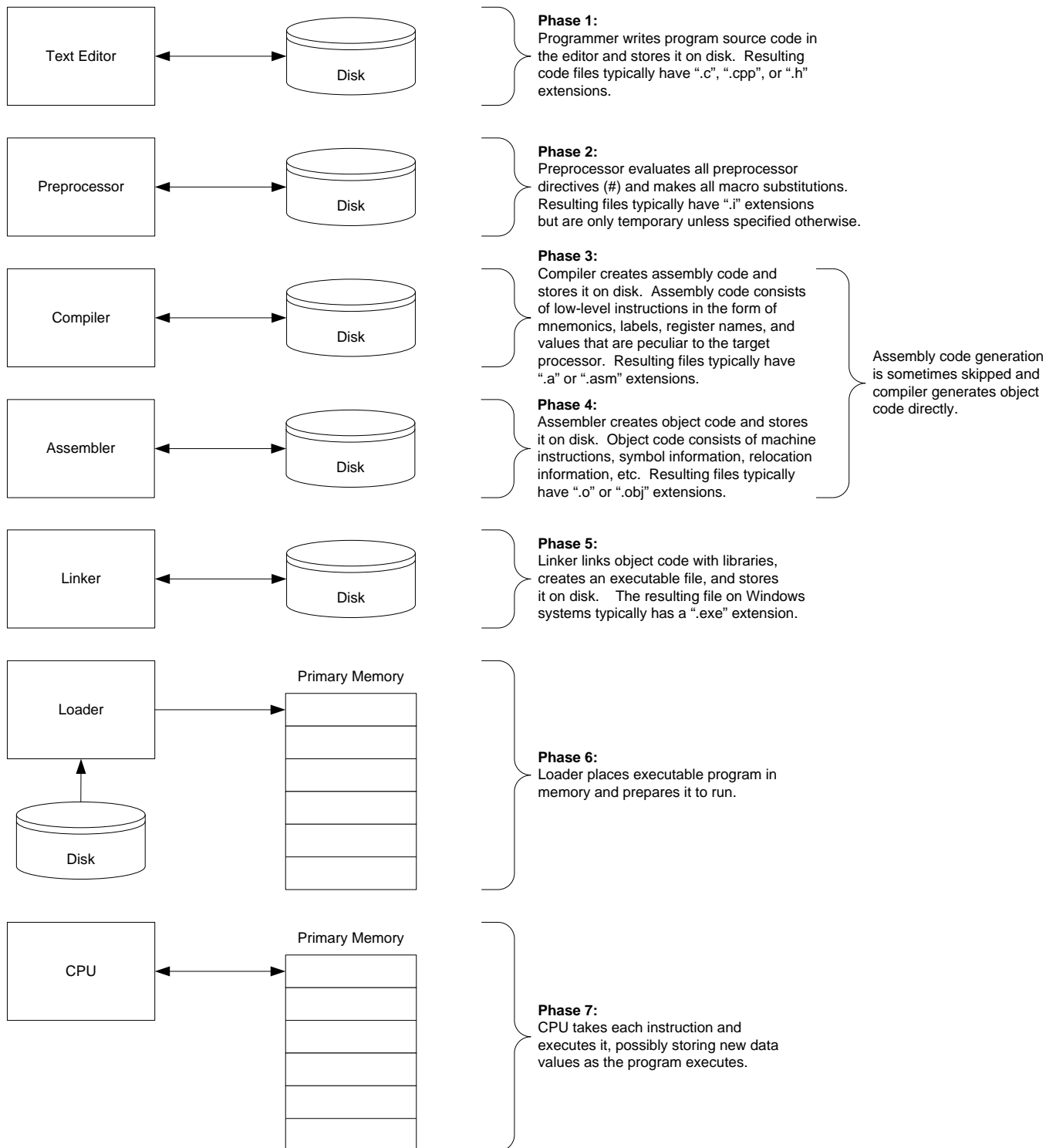


Figure 1

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.

Code::Blocks

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 "Code::Blocks v16.01" IDE with the MinGW GCC compiler is used on a Windows system 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 Code::Blocks, 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 Code::Blocks

The Code::Blocks IDE and accompanying MinGW GNU compiler are both available free from <http://www.codeblocks.org/>. Go to the "Downloads" page and get the "binary release" of the version containing the GCC compiler (not the TDM-GCC compiler).

Installing Code::Blocks and Opening the IDE

Install Code::Blocks as you would any other application, following the screen prompts and accepting the installation defaults, then open the IDE.

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.

Suggested Change to the IDE's Global 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 possibly 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 **Settings → Editor...** to open the "Configure editor" dialog box (Figure 2);
2. Select **General settings** in the left frame and the **Editor settings** tabbed page in the right frame;
3. In the right frame make sure the "Use TAB character" check box is not checked and the "TAB size in spaces:" field contains the desired value (I recommend 3 or 4);
4. Click **OK** at the bottom of the dialog box to close it and accept changes.

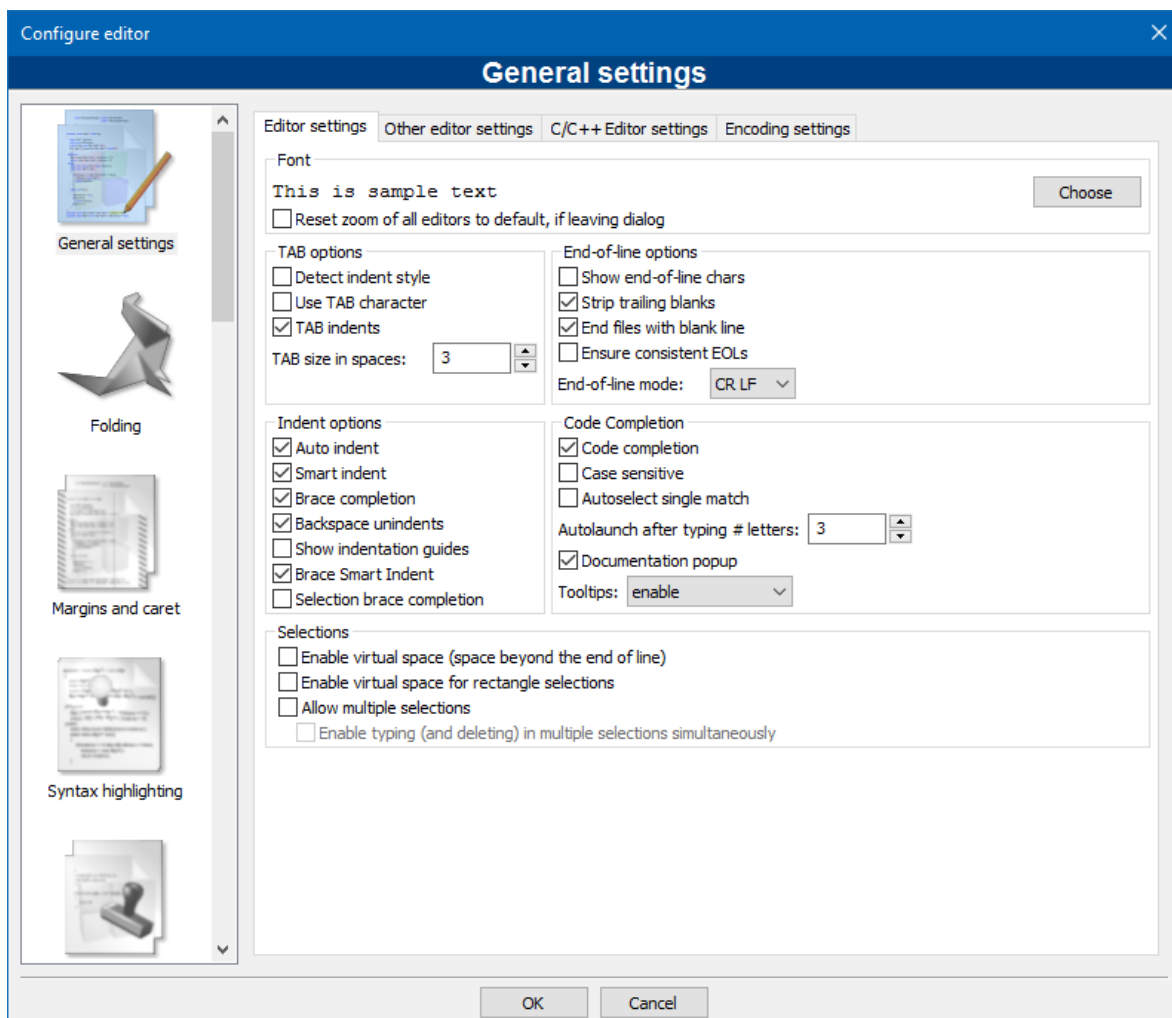


Figure 2

Removing Existing "Hard" Tabs

The previous procedure will not affect any "hard" tabs that are already in the file. These must either be replaced one-at-a-time manually, or by using the instructor-supplied "Hard Tab Removal Tool", or by some other means.

Suggested Change to the IDE's Global Text Editor Line Numbering

...affects the entire IDE. It is recommended that the text editor built into the IDE be used for the editing of all source files in the project. By default the editor displays line numbers in such files, and these are very useful when trying to match compiler error/warning messages to the code causing them and when referring to code in general. In case line numbers are not displayed use the following procedure to enable them. 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 **Settings → Editor...** to open the "Configure editor" dialog box (Figure 3);
2. Select **General settings** in the left frame and the **Other editor settings** tabbed page in the right frame;
3. In the right frame make sure the "Show line numbers" check box is checked;
4. Click **OK** at the bottom of the dialog box to close it and accept changes.

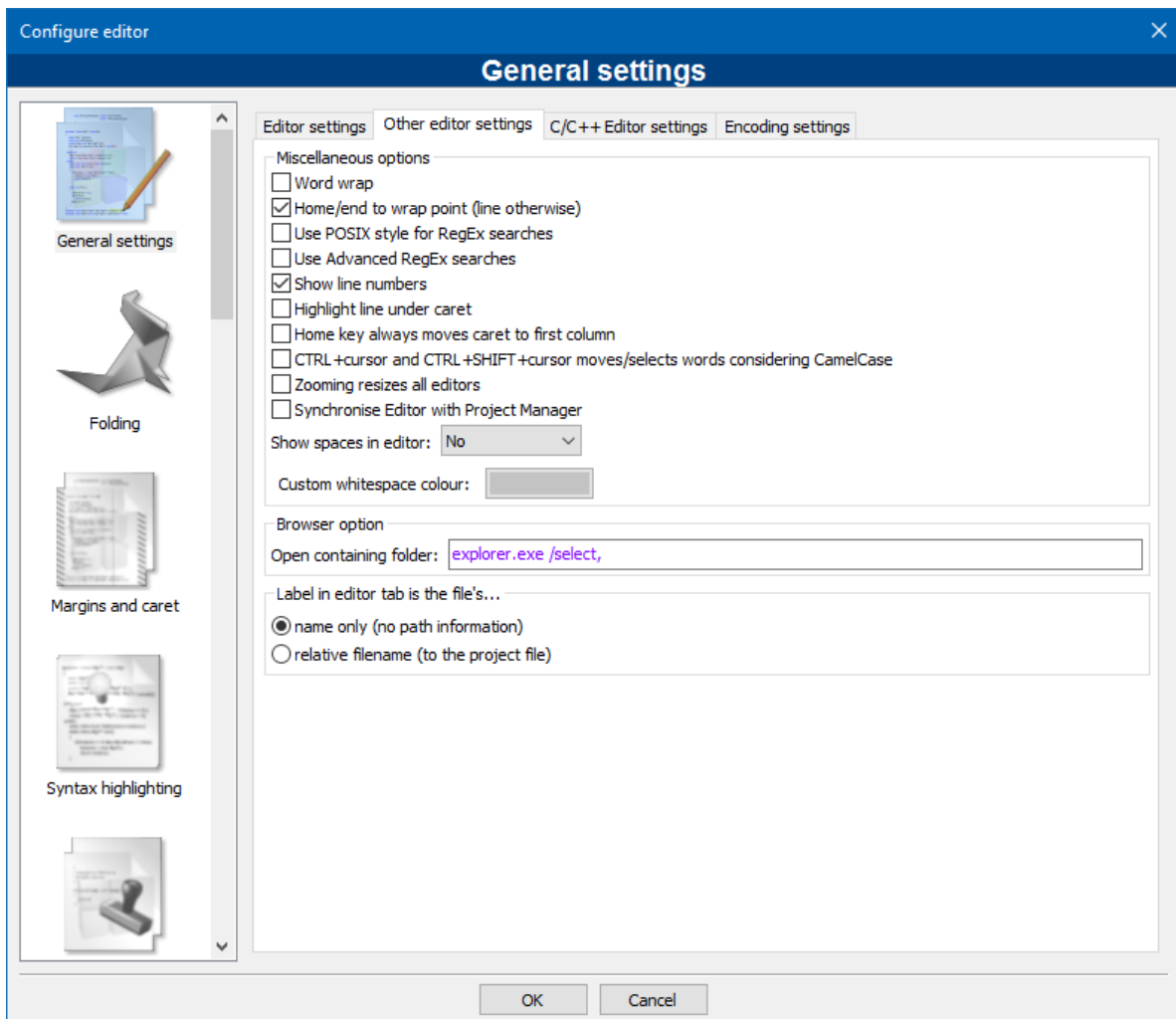


Figure 3

Suggested Change to the IDE's Global Compiler Settings

...affects the entire IDE. Programs are most portable when they adhere to the C++98 language standard for C++ and the C89/C90 language standards for C. However, these are very old standards and many useful features have been added to the newer versions. Since many of these features are in common use today some of them, as appropriate, are allowed in this course and should be enabled as follows:

1. From the IDE's main window menu select **Settings → Compiler...** to open the "Configure editor" dialog box (Figure 4);
2. Select **Global compiler settings** in the left frame and the **Compiler Flags** tabbed page in the right frame;
3. Make sure that only the 5 check boxes indicated are checked;
4. Click the **OK** at the bottom of the dialog box to close it and accept changes.

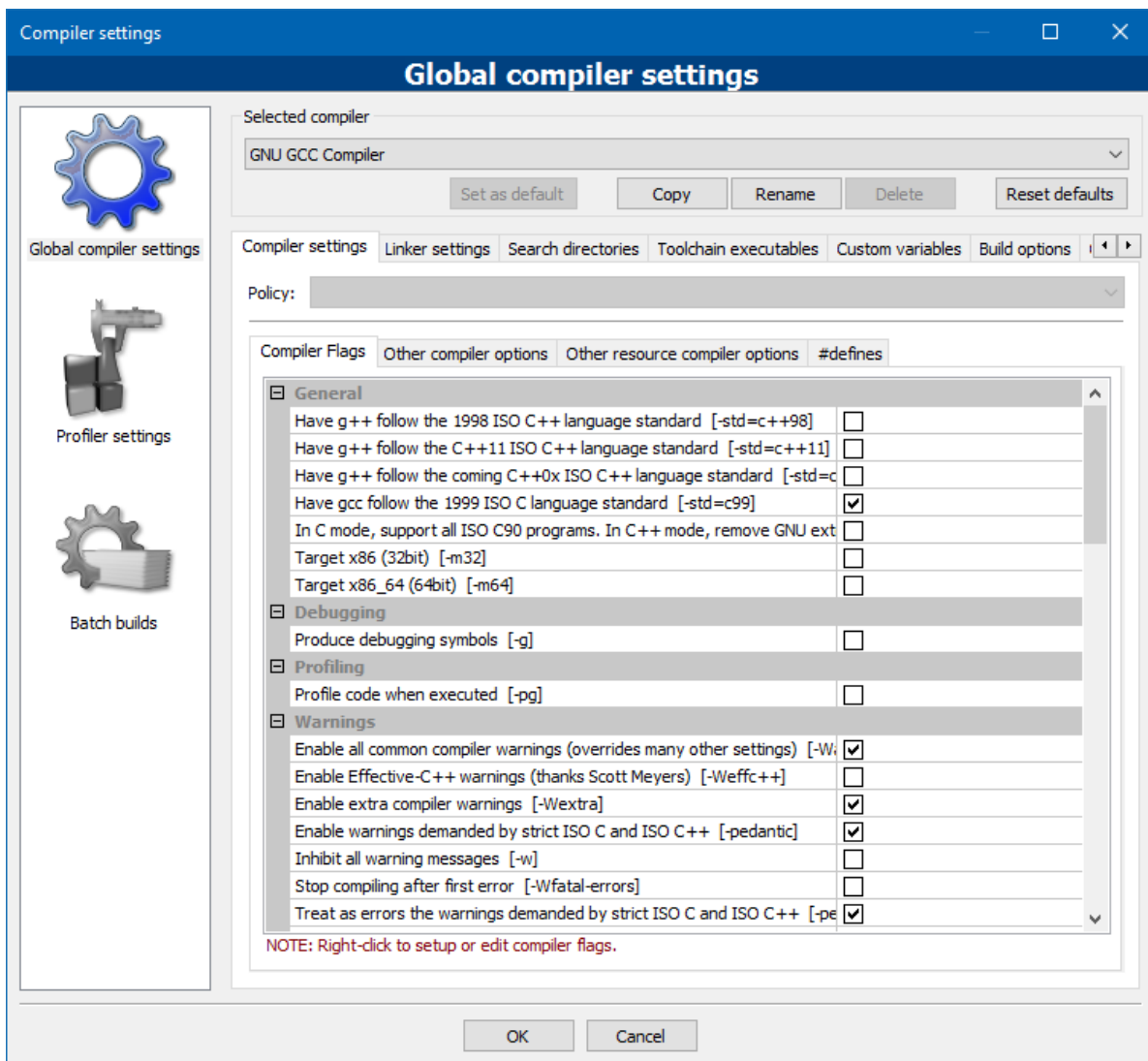


Figure 4

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. 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 of this document titled “Reusing the Same IDE Project for Every Exercise”.)

1. From the IDE’s main window menu select **File → New → Project...** This opens the “New from template” dialog box (Figure 5);
2. In the left frame select **Projects** and in the right frame select **Empty project**;
3. Click the **Go** button; This closes the “New from template” dialog box and opens the “Empty project” dialog box (Figure 6 on the next page);

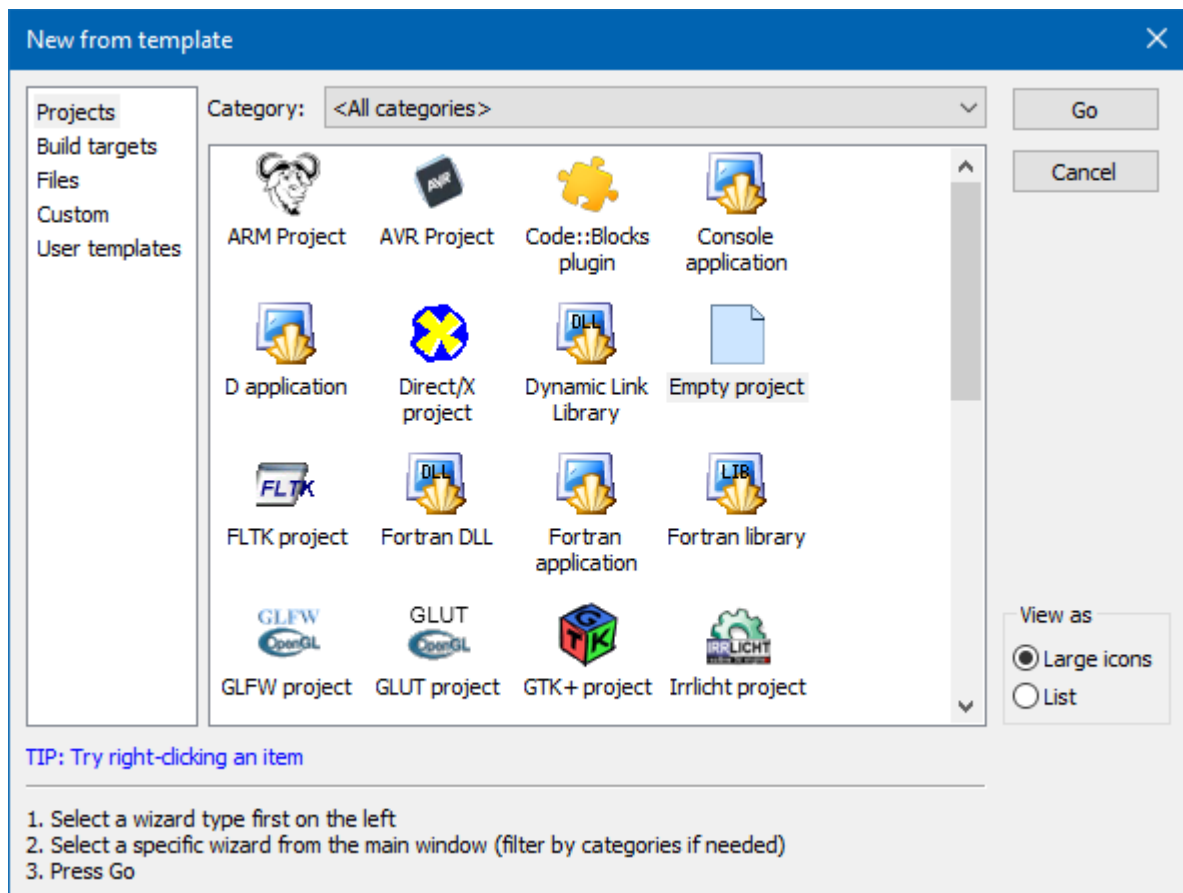


Figure 5

Continued on the next page...

Creating a New Project, continued

4. If the *"Empty project – Welcome to the new empty project wizard!"* dialog box is displayed instead of the dialog box in Figure 4, click the **N**ext button to display Figure 6.
5. In the **Project title:** field, enter **MyPrograms**
6. In the **Folder to create project in:** field, enter **C:\Projects**
7. Click the **N**ext button; this displays the dialog box in Figure 7 on the next page.

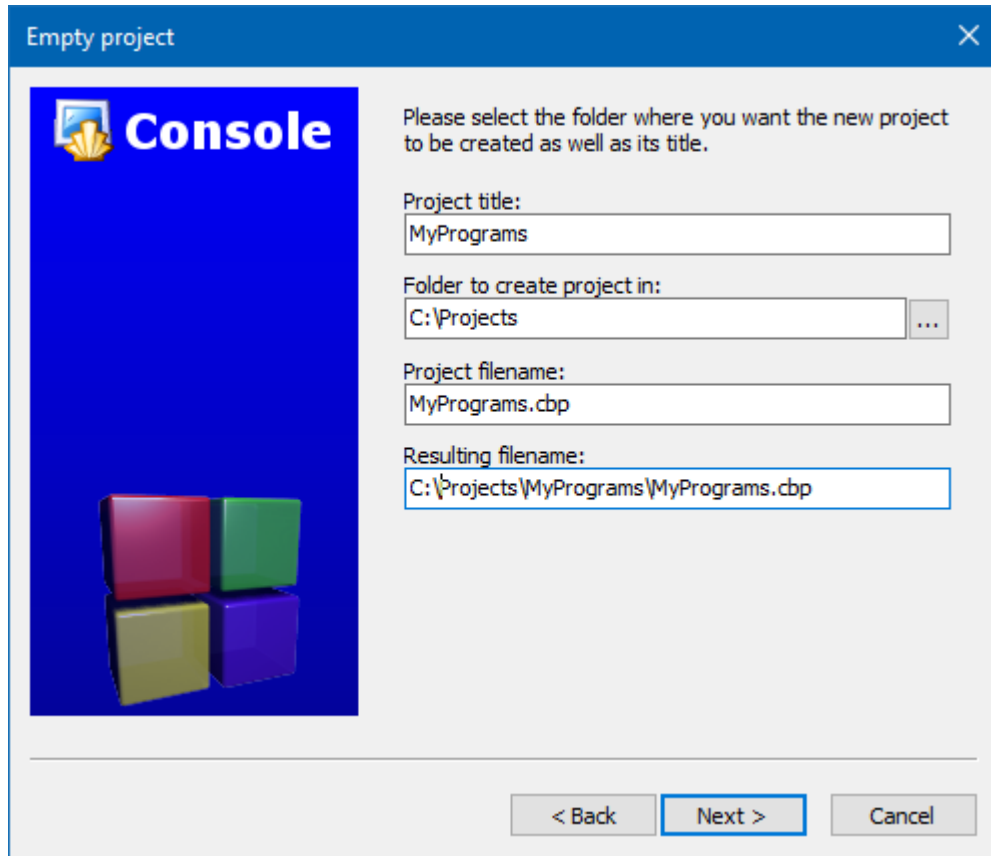


Figure 6

Continued on the next page...

Creating a New Project, continued

8. Select **GNU GCC Compiler** from the **Compiler:** dropdown menu;
9. Check the "Create "Debug" Configuration" check box;
10. Uncheck the "Create "Release" Configuration" check box;
11. Leave the other settings unaltered;
12. Click the **Finish** button. This closes the "Empty Project" dialog box.

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.

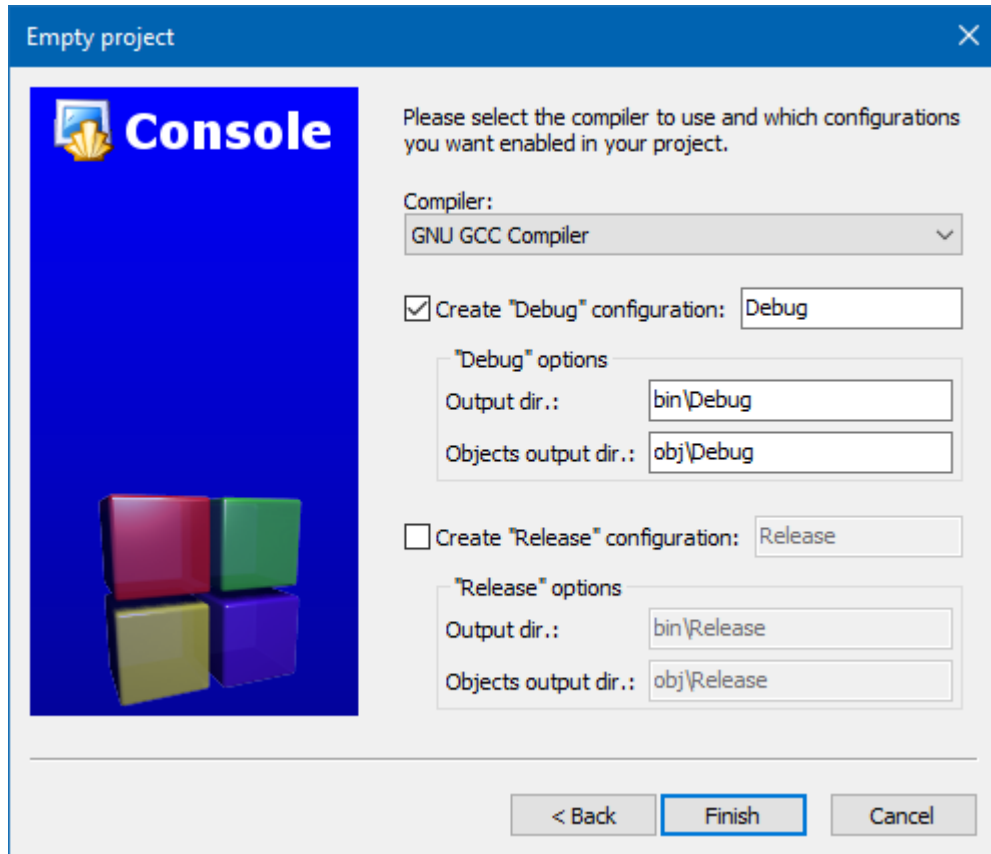


Figure 7

Adding One or More Source Code Files to a Project

Use this procedure to create and add any number of new source code files to an active project, or to add any number of existing source code files to that project. The IDE will then be able to automatically compile and link these files together as appropriate to produce an executable program file.

Figure 8 illustrates the minimum appearance of the IDE's main window if the **MyPrograms** project is active. Note the name of the project in square brackets in the title bar at the top of that window. There may also be other things on the window that are not shown in the figure.

1. If the "Management" frame on the left side of the main window is not visible open it by selecting **View → Manager** from the main menu.
2. If the desired project is not listed in the "Management" frame try to open it by selecting it from the recent projects list in **File → Recent projects**. If it is not listed navigate to it by selecting **File → Open...**
3. If the desired project is listed in the "Management" frame but not active (not named in the title bar at the top of the window), activate it by right-clicking on its name and selecting **Activate project**.

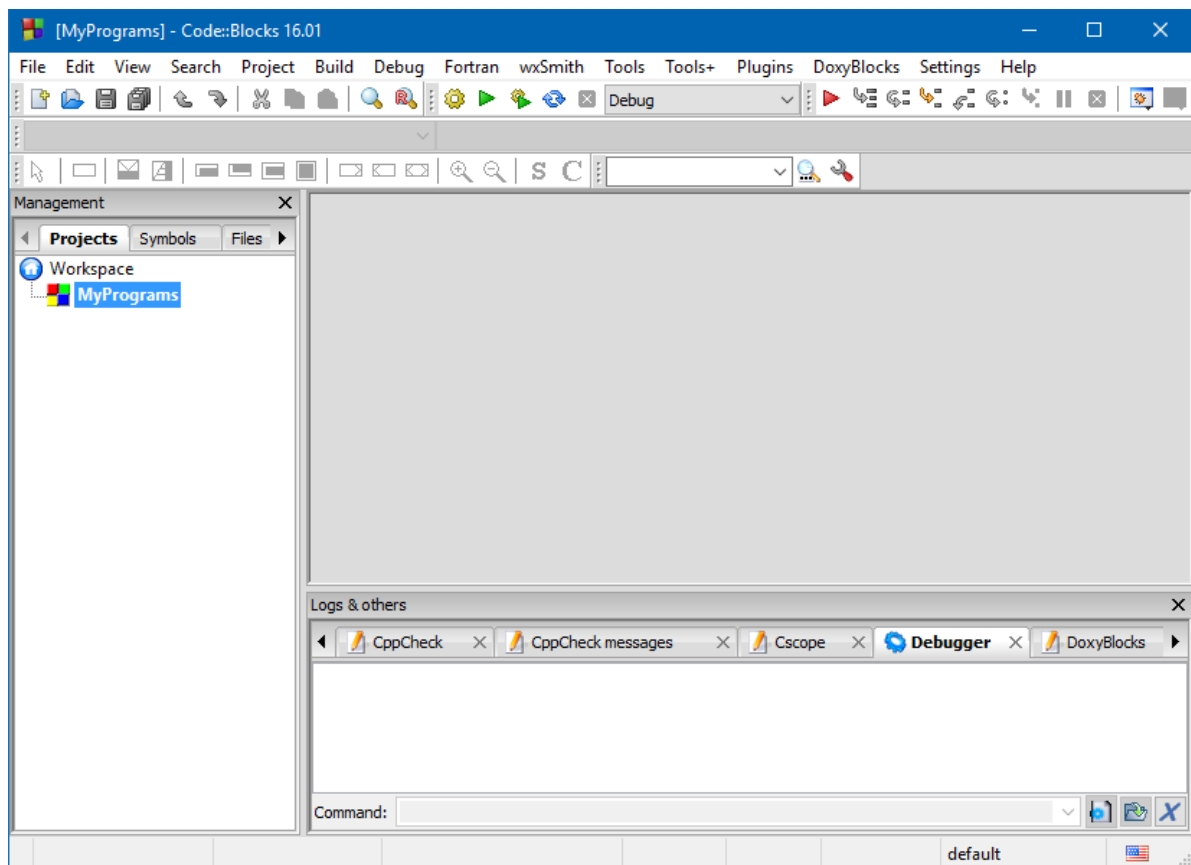


Figure 8

Continued on the next page...

To create a **new source code file** and add it to the project:

1. Select **File → New → Empty file**. This opens a message box asking if you want to add this file - click the **Yes** button; This then opens the "Save file" dialog box (Figure 9);
2. By default it will attempt to create new files in the project directory itself, which I recommend for this course. However, if you wish to create them in other directories merely navigate there;
3. In the **File name:** field near the bottom of the page delete the default file name and extension and type the desired file name and extension (.c, .cpp, or .h). The name "Test.c" has been chosen for this example.
4. Click the **Save** button.
5. Repeat this procedure for each new source code file you wish to create and add.
6. To verify that files have been created and added simply expand the project tree in the "Management" frame by clicking the + sign in the box next to the project name and its various components.

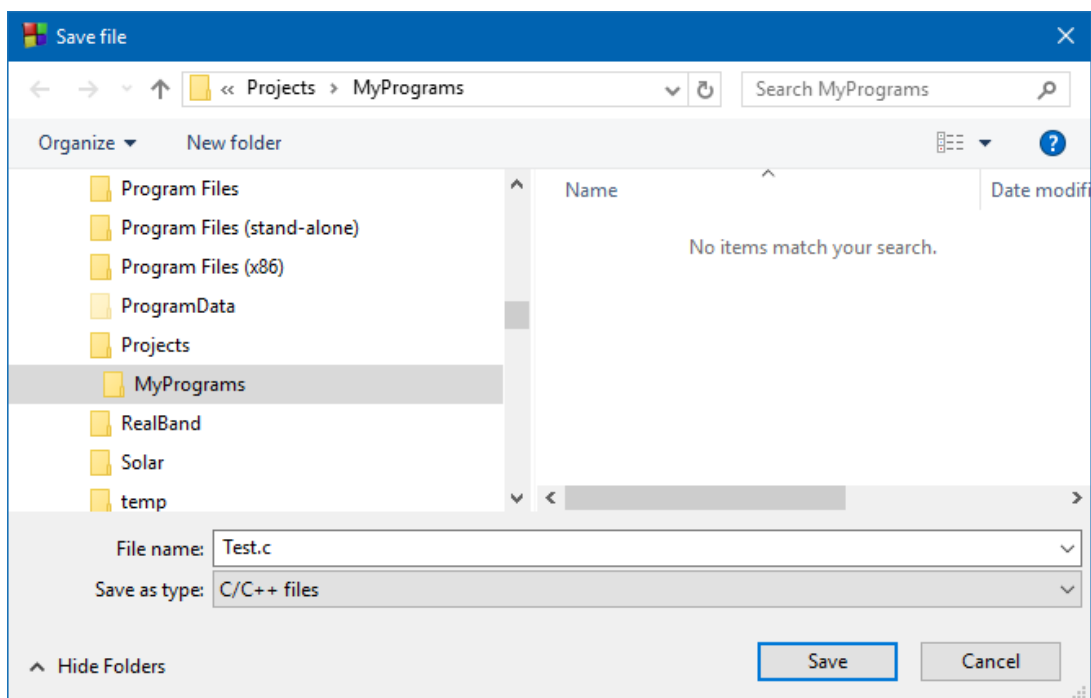


Figure 9

Continued on the next page...

To add an existing source code file to the project:

1. Right-click on the project name in the "Management" frame and select **Add files...** This opens the "Add files to project..." dialog box (Figure 10);
2. By default it will attempt to add existing files from the project directory itself, which I recommend for this course. However, if you have placed them somewhere else merely navigate there;
3. Select one or more source code files you wish to add to the project. An existing file named "AnotherFile.h" has been selected in this example.
4. Click the **Open** button to add them.
5. Repeat this procedure if you wish to add additional existing source code files.
6. To verify that files have been added simply expand the project tree in the "Management" frame by clicking the + sign in the box next to the project name and its various components.

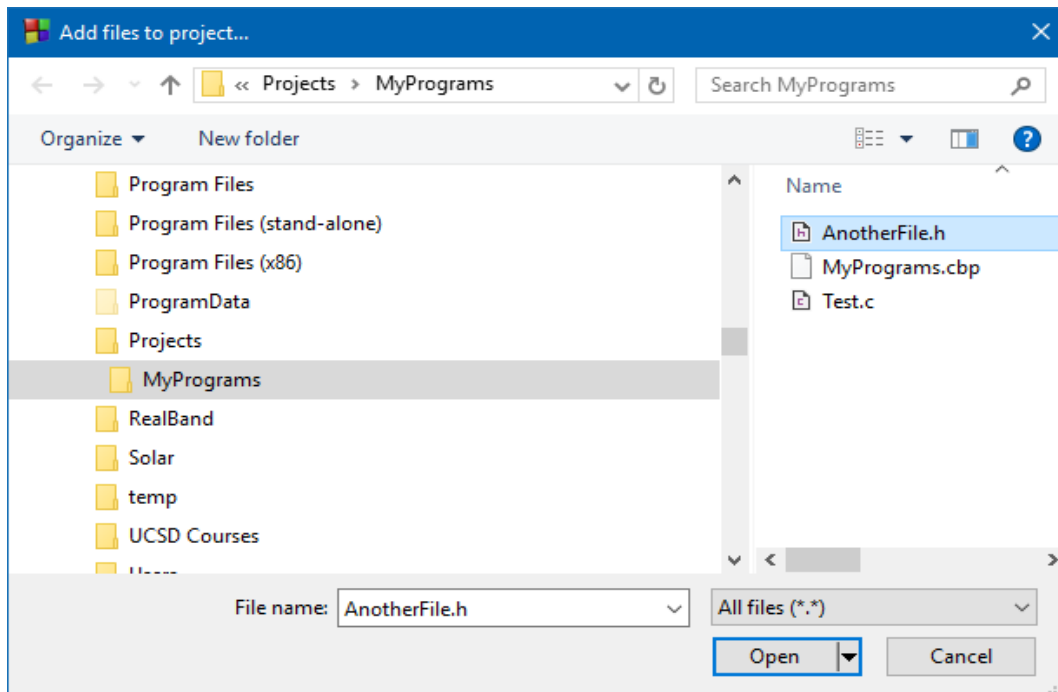


Figure 10

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. Highlight the project (not the workspace) you are using ;
2. Select **Project → Properties** (Figure 11) to open the “Project/targets options” window as shown in Figure 12 on the next page.

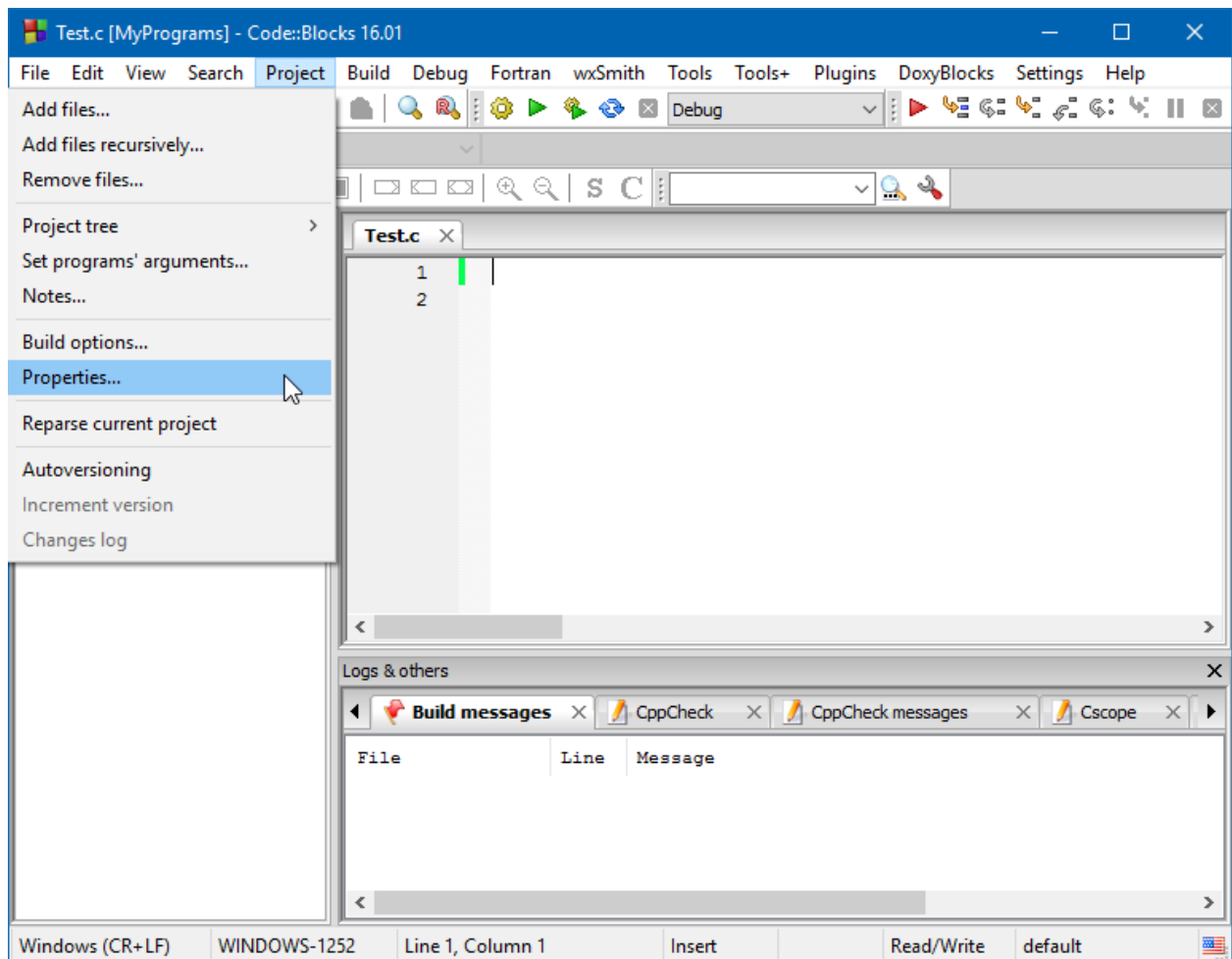


Figure 11

Continued on the next page...

Changes to the Project Settings, continued

The tabs near the top of the “Project/targets options” window (Figure 12) list the various categories for which project settings changes can be made. I recommend you look through them just to get a feel for what’s there, even if you don’t understand most of it.

3. No changes will be made at this point.

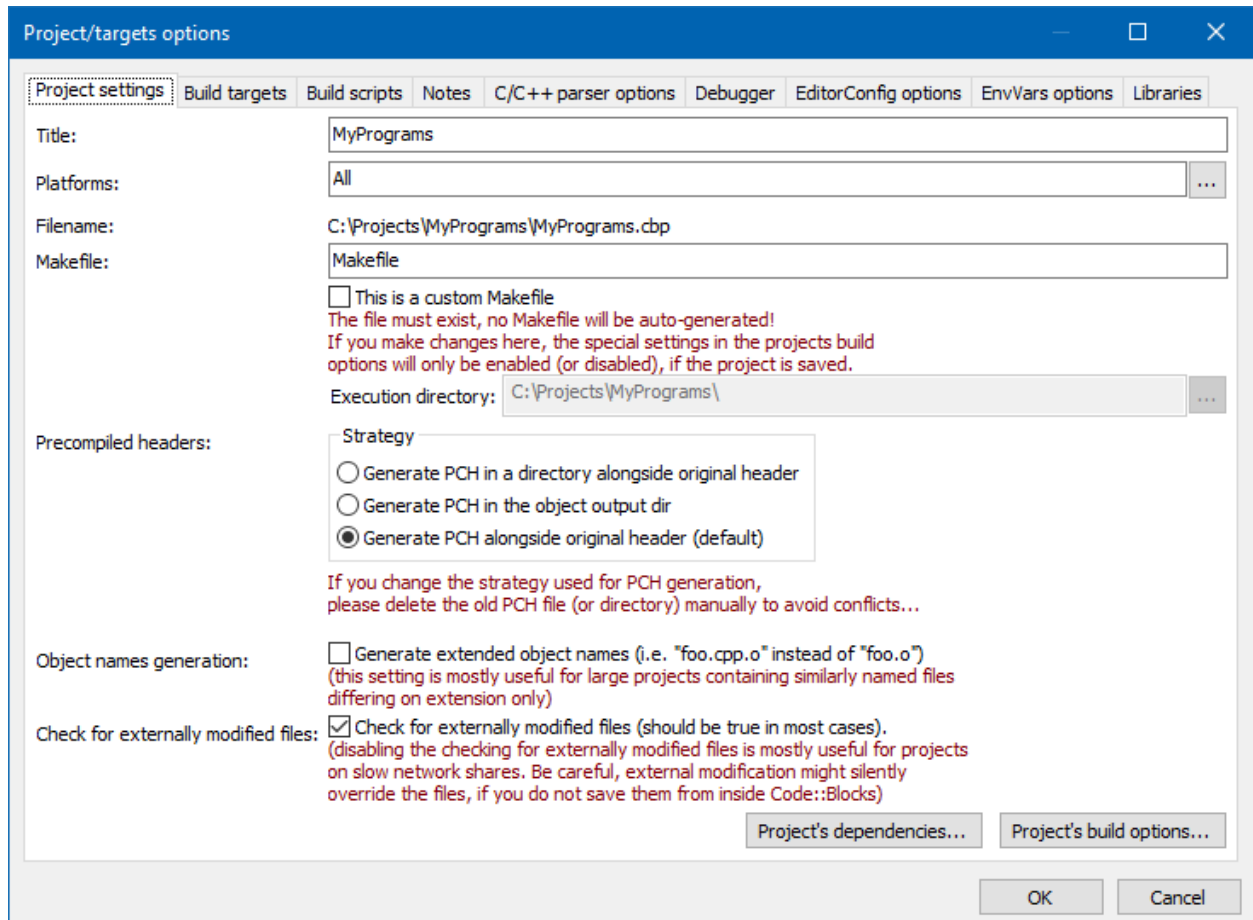


Figure 12

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.cbp** 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 <stdio.h> // Remove if/when no longer needed
```

```
#include <unistd.h> // Remove before assignment checker submission
```

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 16 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 "Project/targets options" window using the technique shown on page 13.
2. Select the **Build targets** tab near the top of the window (Figure 13) and find the **Execution working dir:** field. By default the working directory is a single period, which 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 ellipsis to the right of that field and browsing for the desired directory.
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.

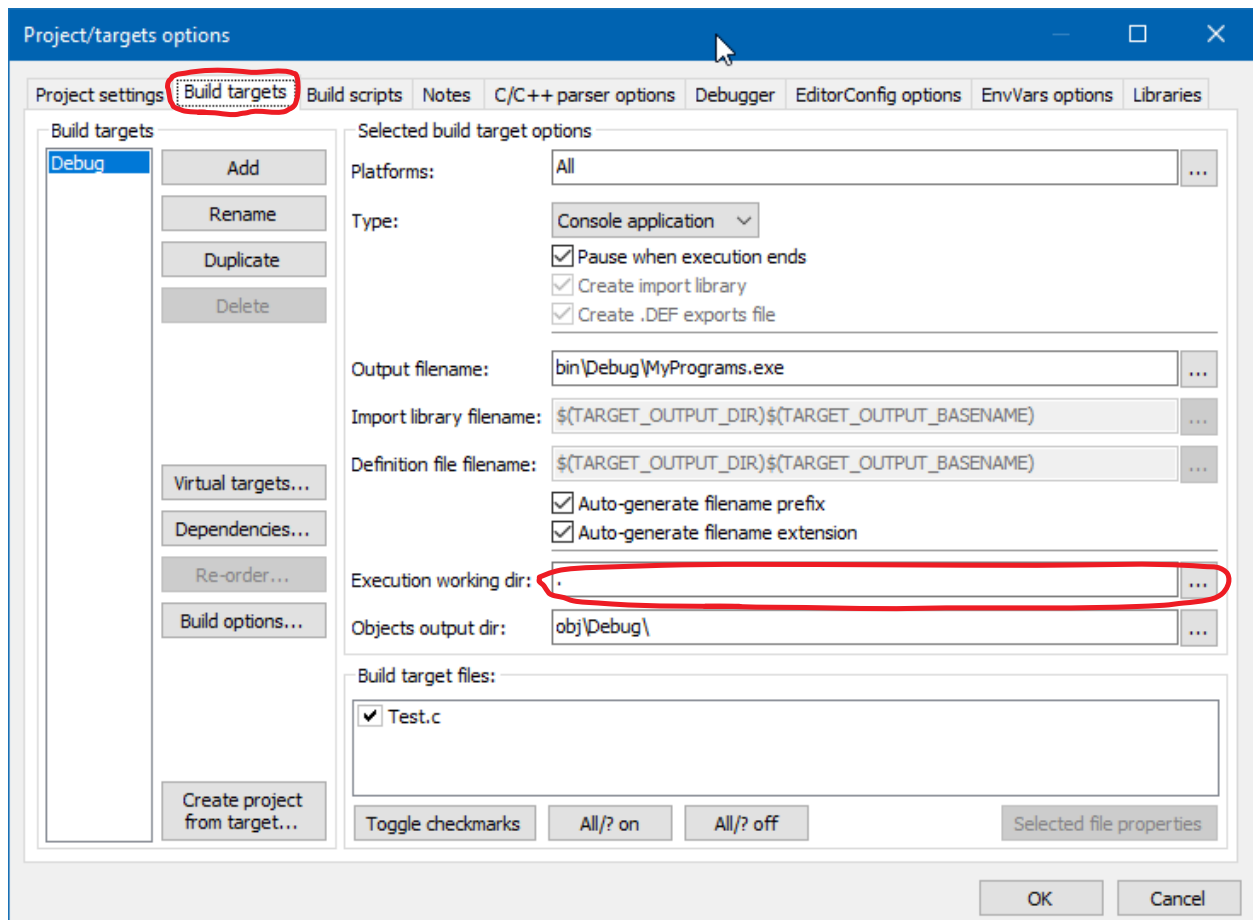


Figure 13

Removing Source Code Files from a Project

Use this procedure to remove files from an existing project. Although they will be removed from the project's file list they won't be deleted from the computer itself. This will permit them to be added back into the project at a later time if desired.

1. Make the files visible by expanding the project tree in the "Management" frame by clicking the + sign in the box next to the project name and its various components (Figure 14);
2. Select the file you wish to remove by clicking on it once;
3. Right-click on the selection to open the context menu (Figure 15);
4. Click **Remove file from project**;
5. Repeat for each file you wish to remove.

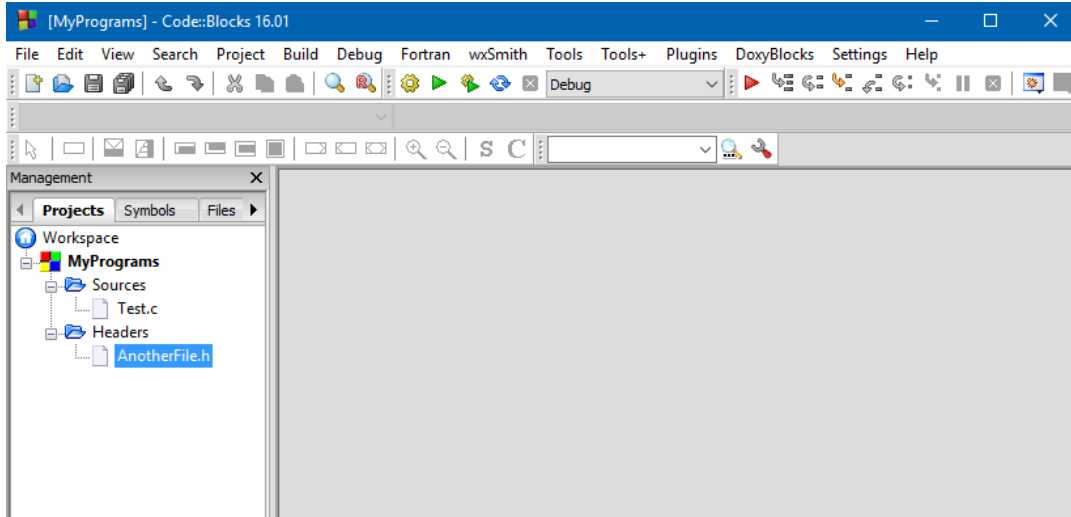


Figure 14

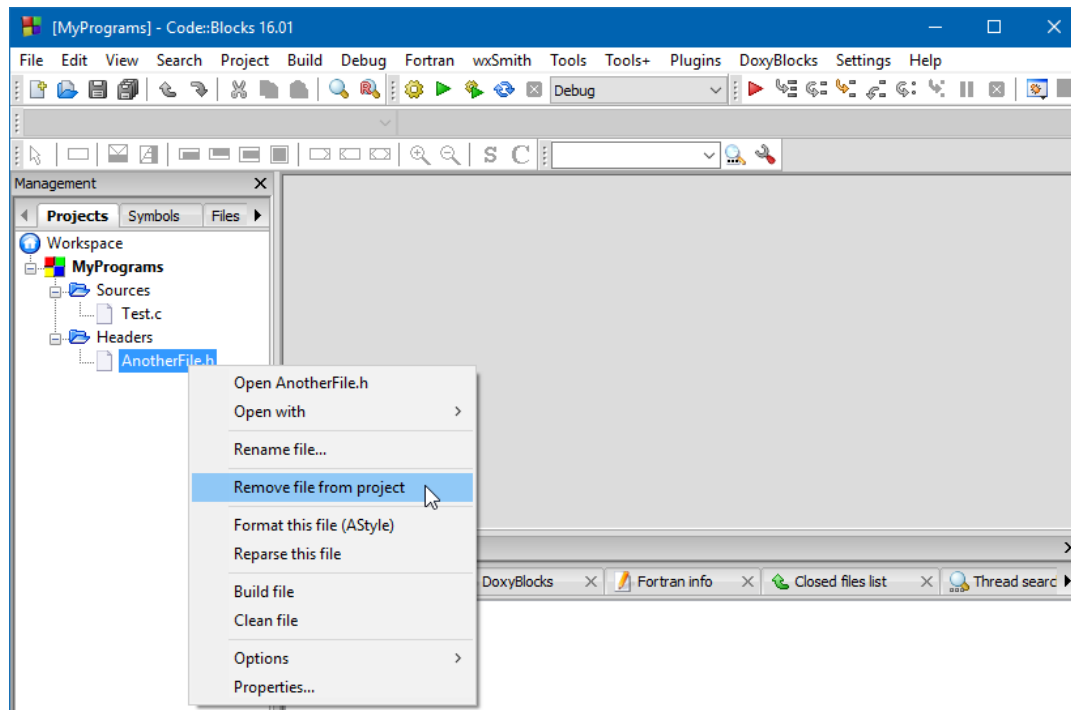


Figure 15

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 remove its file(s) from the project and create and add new ones using the procedures previously outlined.

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. Refer to Figure 16 for these operations.

To Compile the Code:

Make sure the “Logs & others” frame is visible at the bottom of the IDE main window and the “Build messages” tabbed page is selected. This is where all errors/warnings that occur during the build will be reported. If it is not visible do

View → Logs

and select the “Build messages” tabbed page. To build the executable program do

Build → Build

If any errors/warnings are reported (as they are in this example) correct them build again.

To Run the Program:

Once your code compiles without errors (and preferably also without warnings), run the resulting program by doing

Build → Run

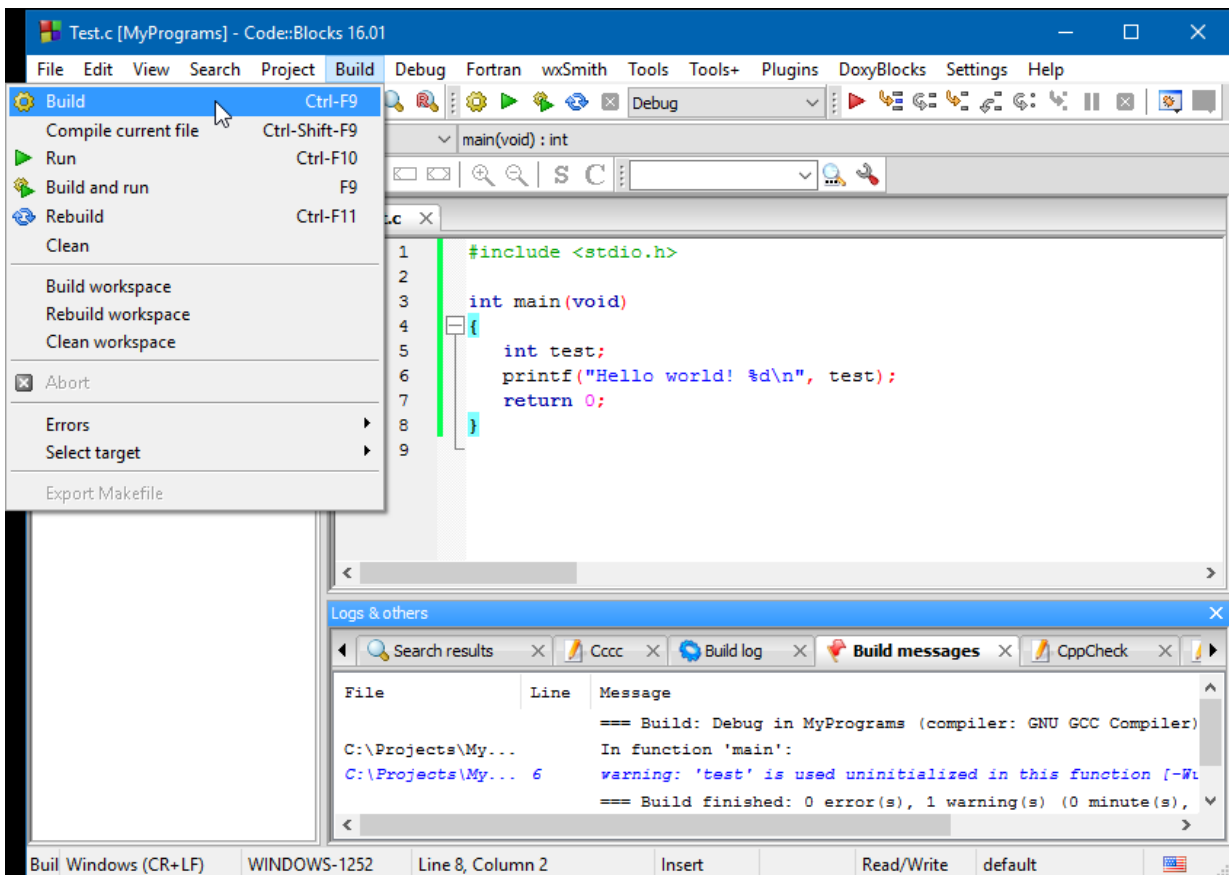


Figure 16

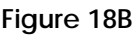
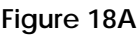
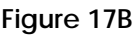
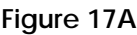
Keeping the Command Window Open after a Program Runs

If you are running the program as previously described its command window should stay open after the program has terminated so it can be examined or captured if desired. However, if you are running the program in the “debug” mode (**Debug → Start/Continue**) as you should be doing if you are attempting to debug it, the command window will typically close upon program termination. The best solution to this is to place a breakpoint on the line containing the **return** statement in the **main** function. 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 **F5**.

However, if for some reason the breakpoint method doesn’t work 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.

- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

Controlling Debugging: Debugging can be controlled through your choice of menu selections, icon clicks, and/or keyboard shortcuts. Figures 17A and 17B show a portion of Code::Blocks' 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 18A and 18B show the debugging icons before and after the program starts running, respectively. These are located just below the main window's title bar.



Using the IDE's Debugger, continued

The Program: Figure 19 below shows a typical program in a Code::Blocks 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 19

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 empty area to the right of the line number, or you can place the cursor on that line and either press **F5** or select from the **Debug** menu. Five breakpoints have been set in Figure 20, 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 24) to the next statement or set a breakpoint (page 21) on that next statement and continue (page 24).

```
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 20

Starting Debugging: An easy way to start the program is to press the **F8** key, but you may instead click the **Debug/Continue** icon (1st icon in Figure 18A) or select **Start/Continue** from the **Debug** menu (Figure 17A). A yellow arrowhead in the empty space next to a line indicates a debugging pause, at which point the values of variables and other expressions can be examined.

Figure 21 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 or deleted at any time. The easiest way to delete them individually is by clicking the red dots or placing the cursor on that line and pressing **F5**. You can delete them all at once from the **Debug** menu.

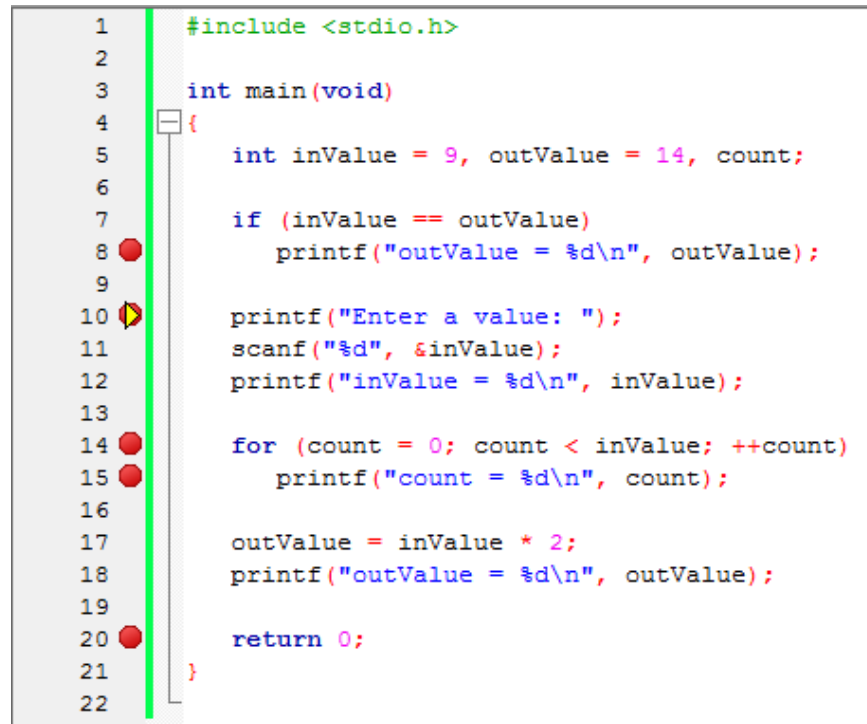


Figure 21

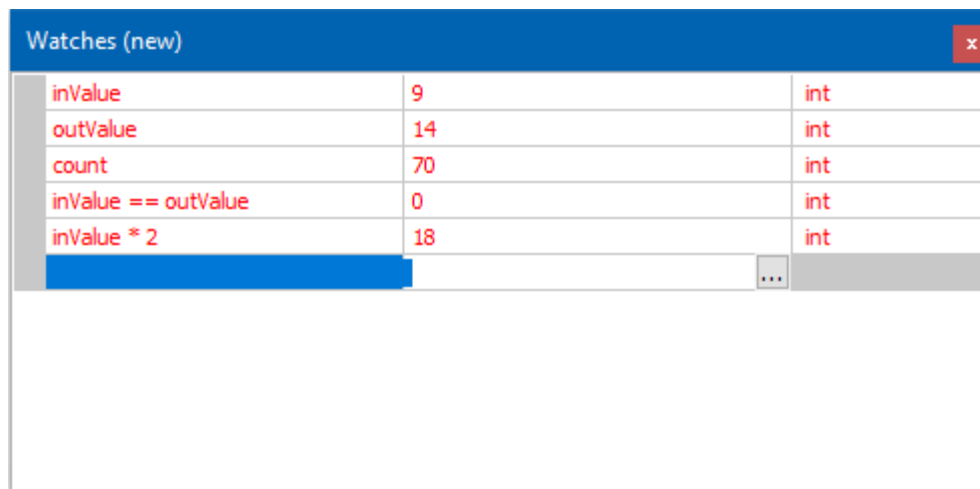
Examining Variables and Other Expressions: Whenever a program is paused at a breakpoint or after a single-step you may view the value of any variable or other expression currently in scope.

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 24) to the next statement or set a breakpoint (page 21) on that next statement and continue (page 24).

The “watch” window may be used to simultaneously view the values of any number of variables or other expressions that are in scope. It will open automatically when something is added to it or it may be opened manually by selecting **Debugging Windows → Watches** from the **Debug** menu. Items may be added by either selecting them in the code and dragging them onto the watch window, right-clicking them in the code and selecting **Watch ‘...’** from the resulting context menu (... should indicate what you are about to watch), or simply manually typing any variable or expression you want to watch directly into the watch window. You may do this for as many items as desired, as shown in Figure 22.

The watch window will close when you stop debugging, although everything you were watching will be preserved in case you want to debug again. Watch window entries may be removed by selecting them and pressing **Delete**, or may be modified by right-clicking them, selecting **Rename** from the resulting context menu, and making the desired changes.

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 on an entry and selecting **Properties** from the resulting context menu, you can change this as appropriate in the resulting **Edit watch** window.



Watches (new)		
inValue	9	int
outValue	14	int
count	70	int
inValue == outValue	0	int
inValue * 2	18	int
...		

Figure 22

2
3 **Terminating, Continuing, and Single-Stepping:**

4 Whenever a program is paused during debugging you have three main choices. You may either

- 5 1. terminate the program;
 - 6 2. resume program execution to whichever comes first of the next breakpoint, user input, or
 - 7 program end (known as "continuing");
 - 8 3. execute only the next statement then pause again (known as "single-stepping").
- 9

10 **Terminating:**

11 Press **Shift+F8** or click the **Stop debugger** icon (last icon in Figure 18B) or select the **Stop debugger** item in

12 the **Debug** menu.

13

14 **Continuing:**

15 Press **F8** or click the **Debug/Continue** icon (1st icon Figure 18B) or select the **Start/Continue** item in the

16 **Debug** menu. It is important to note that if there is any code that requires user input, such as the **scanf**

17 on line 11 in this example, the program will stop and wait for that input just as it does when not

18 debugging. This does not represent a debugging pause so no variables/expressions can be examined

19 and the program cannot be continued or single-stepped until the user provides that input.

20

21 **Single-Stepping:**

22 Single-stepping is an extremely useful tool for progressing through your code one statement at a time to

23 see its effect on your variables and input/output operations. There are three basic stepping operations:

24

- 25 1. **Step Over – F7** or the **Next Line** icon (3rd icon in Figure 18B) or the **Next line** item in the **Debug**
- 26 menu.
- 27 This is the most common stepping operation and simply executes the code on the current line
- 28 and pauses on the next. It's important to note that if there is any code that requires user input,
- 29 such as the **scanf** on line 11 in this example, the program will stop and wait for that input just as it
- 30 does when not debugging. This does not represent a debugging pause so no
- 31 variables/expressions can be examined and the program cannot be continued or single-
- 32 stepped until the user provides that input.
- 33
- 34 2. **Step Into – F11** or the **Step Into** icon (4th icon in Figure 18B) or the **Step Into** item in the **Debug**
- 35 menu.
- 36 For statements that contain one or more function calls this will allow you to into the functions'
- 37 code so you can debug it or merely look at it. You should normally not step into library functions
- 38 since their source code is usually not available, but you may want to step into the code for
- 39 functions you write if you are trying to debug them.
- 40
- 41 3. **Step Out – Shift+F11** or the **Step Out** icon (5th icon in Figure 18B) or the **Step Out** item in the
- 42 **Debug** menu.
- 43 If you have stepped into a function by mistake or simply want to complete the function you are
- 44 currently in for any reason, do a step out. If there are any breakpoints or user inputs in that
- 45 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. From the IDE's main window menu select **Project → Set programs' arguments...** to open the "Select Target" dialog box (Figure 23);
2. In the **Program arguments:** frame 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**
3. 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.

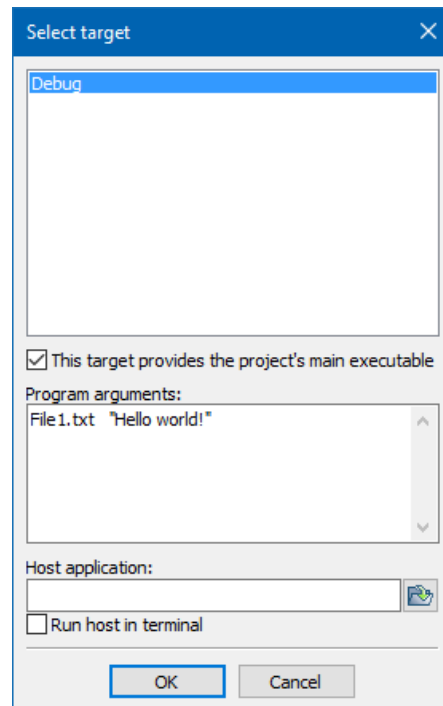


Figure 23