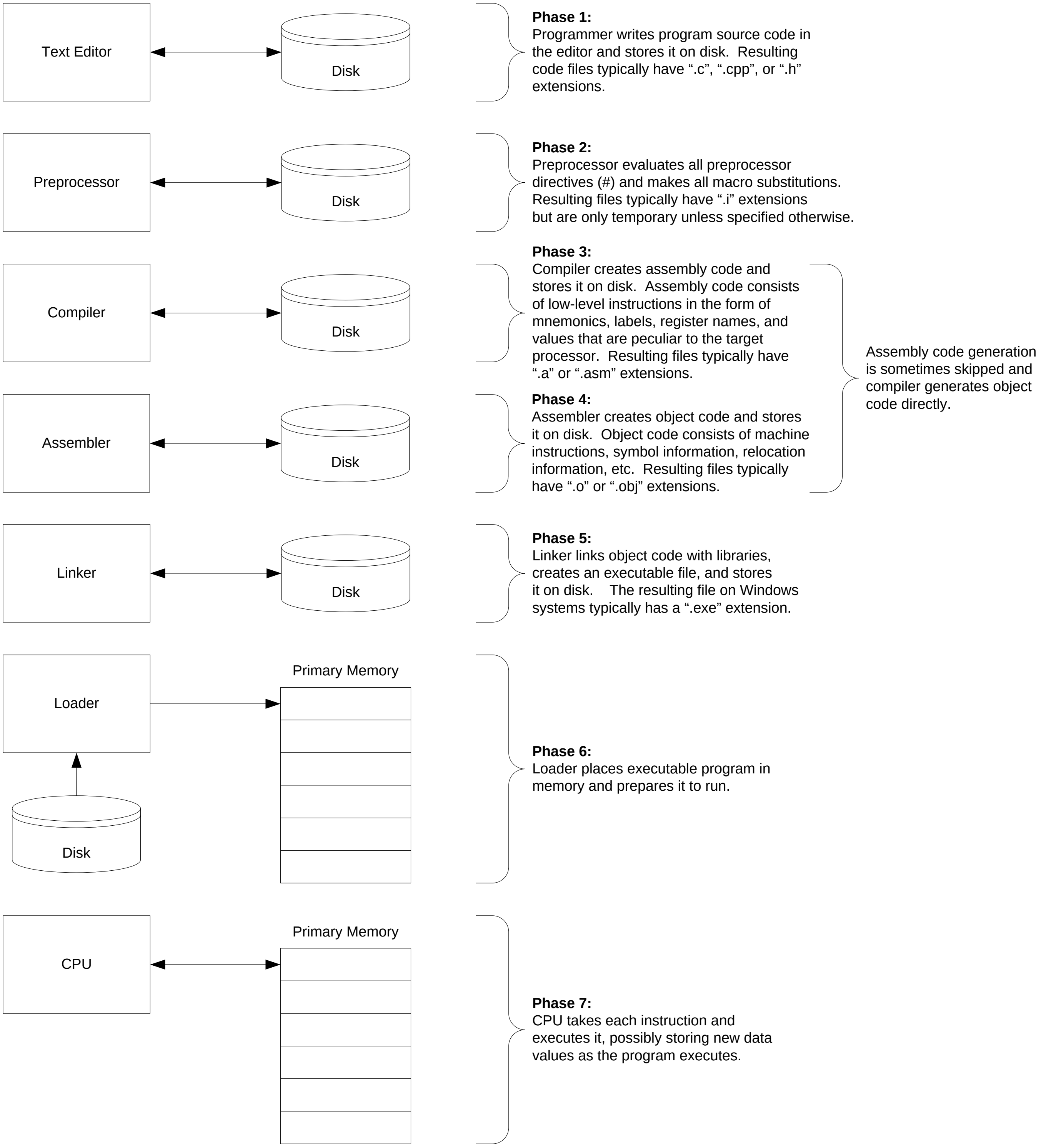
Using The Compiler's IDE – Xcode – v10

C/C++ Programming I & II

Table of Contents

Introduction – How Programs Are Created and Run

Figure 0 below illustrates the typical process of creating and running a program:



Phase 1:
Programmer writes program source code in the editor and stores it on disk. Resulting code files typically have ".c", ".cpp", or ".h" extensions.

Phase 2:
Preprocessor evaluates all preprocessor directives (#) and makes all macro substitutions. Resulting files typically have ".i" extensions but are only temporary unless specified otherwise.

Phase 3:
Compiler creates assembly code and stores it on disk. Assembly code consists of low-level instructions in the form of mnemonics, labels, register names, and values that are peculiar to the target processor. Resulting files typically have ".a" or ".asm" extensions.

Phase 4:
Assembler creates object code and stores it on disk. Object code consists of machine instructions, symbol information, relocation information, etc. Resulting files typically have ".o" or ".obj" extensions.

Assembly code generation is sometimes skipped and compiler generates object code directly.

Phase 5:
Linker links object code with libraries, creates an executable file, and stores it on disk. The resulting file on Windows systems typically has a ".exe" extension.

Phase 6:
Loader places executable program in memory and prepares it to run.

Phase 7:
CPU takes each instruction and executes it, possibly storing new data values as the program executes.

Figure 0

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 doing so in this course simply because of the unnecessary complexity (and pain) involved.

Getting Xcode

The Xcode IDE and compiler are available as a free package from the Mac App Store developer download page at https://developer.apple.com/downloads/. Download the newest version that is compatible with your operating system. If that is not the version covered by this document (v10.x.x) contact the instructor for the appropriate document. If you are in doubt about compatibility you can either search for this information online or simply start with the newest version, try to install it, and keep moving back to earlier versions until the installation succeeds.

Installing Xcode

The Xcode installation procedure is illustrated in detail beginning on the next page.

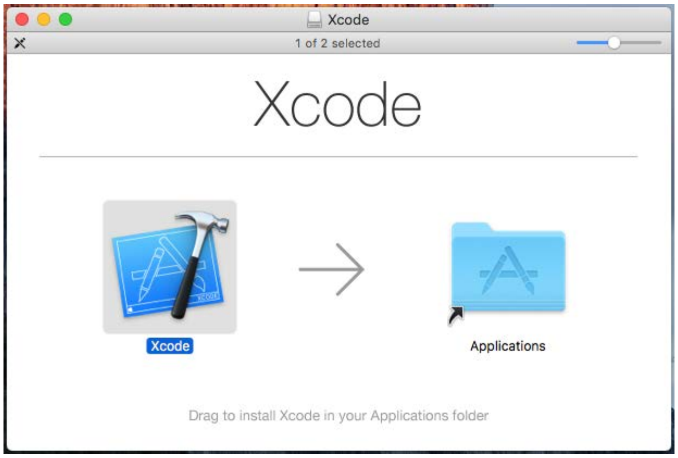
Changing the IDE Defaults

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 Xcode

Start the installation of Xcode as you would any other installation. Depending upon the current state of your machine, some of the windows below may not appear because what they represent has already been accomplished in some way. Merely respond to those that do appear.

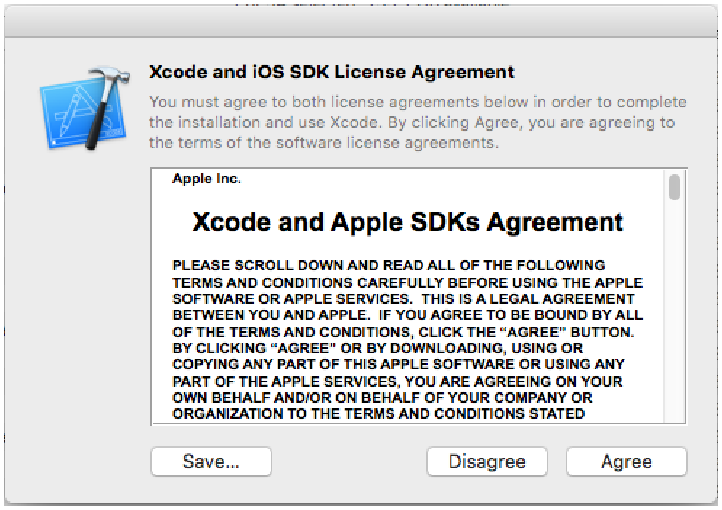
1. If a window like that in Figure 1 does not open, navigate to the ***Applications*** folder and double-click on the ***Xcode*** item. If such a window then opens, drag the ***Xcode*** icon onto the ***Applications*** icon, which will start the installation of ***Xcode*** into the ***Applications*** folder.



Drag to install Xcode in your Applications folder

Figure 1

2. If a licensing agreement window (Figure 2) opens, click ***Agree*** if you agree to the terms.



Xcode and iOS SDK License Agreement

You must agree to both license agreements below in order to complete the installation and use Xcode. By clicking Agree, you are agreeing to the terms of the software license agreements.

Apple Inc.

Xcode and Apple SDKs Agreement

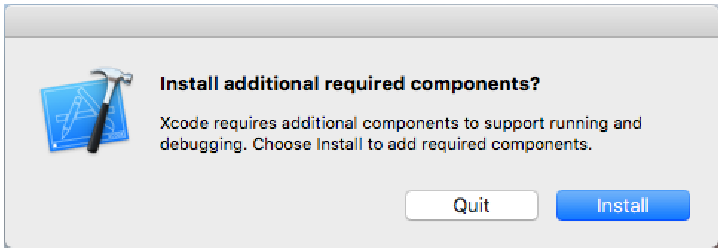
PLEASE SCROLL DOWN AND READ ALL OF THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE USING THE APPLE SOFTWARE OR APPLE SERVICES. THIS IS A LEGAL AGREEMENT BETWEEN YOU AND APPLE. IF YOU AGREE TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS, CLICK THE "AGREE" BUTTON. BY CLICKING "AGREE" OR BY DOWNLOADING, USING OR COPYING ANY PART OF THIS APPLE SOFTWARE OR USING ANY PART OF THE APPLE SERVICES, YOU ARE AGREEING ON YOUR OWN BEHALF AND/OR ON BEHALF OF YOUR COMPANY OR ORGANIZATION TO THE TERMS AND CONDITIONS STATED

Figure 2

Continued on the next page...

Installing Xcode, continued

3. If an “additional requirements” window (Figure 3) opens, click ***Install***.

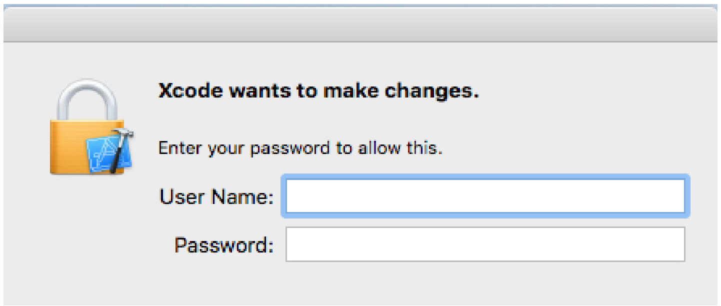


Install additional required components?

Xcode requires additional components to support running and debugging. Choose Install to add required components.

Figure 3

4. If a “User Name and Password” window (Figure 4) opens, enter the required information then click ***OK***.



Xcode wants to make changes.

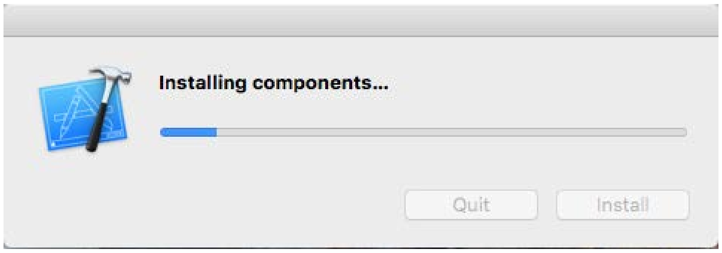
Enter your password to allow this.

User Name:

Password:

Figure 4

5. If an “Installing Components” window (Figure 5) opens, wait for the component installation to complete, at which time the window in Figure 6 on the next page will open.

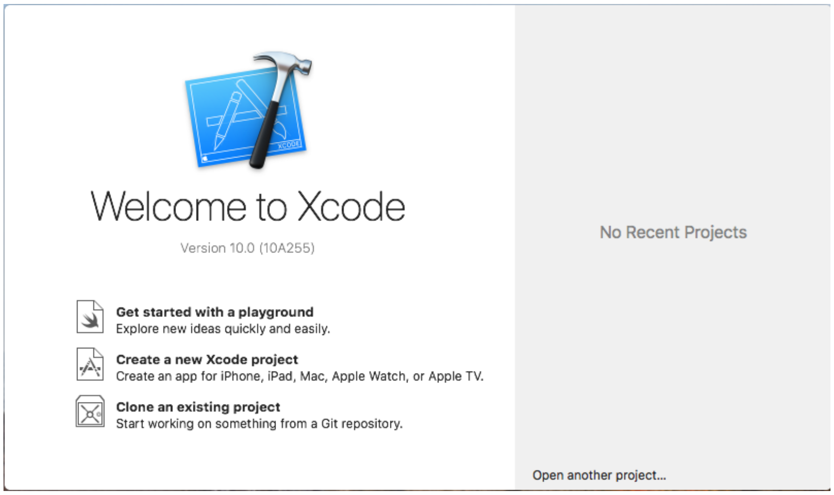


Installing components...

Figure 5

Opening the Xcode IDE

1. Start the Xcode application the way you would start any other, and the “Welcome to Xcode” window in Figure 6 will open.



Welcome to Xcode

Version 10.0 (10A255)

Get started with a playground
Explore new ideas quickly and easily.

Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.

Clone an existing project
Start working on something from a Git repository.

No Recent Projects

Open another project...

Figure 6

2. If Xcode is running the desktop menu bar should display the Xcode menu, which should look similar to Figure 7. Otherwise, restart Xcode. This menu bar will be referred to frequently in the remainder of this document.

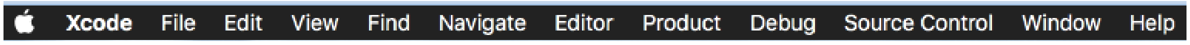


Figure 7

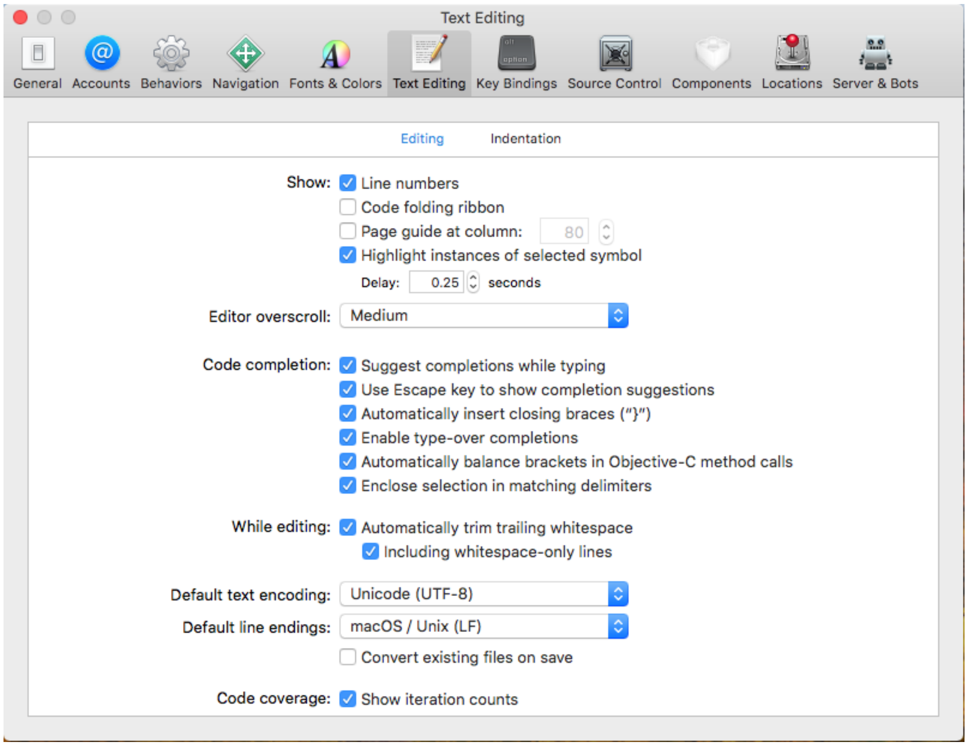
Enabling Text Editor Line Numbering

…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 may not display line numbers in such files, but these are very useful when trying to match compiler error/warning messages to the code causing them and when referring to code in general. To ensure that line numbers are 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 Xcode menu (Figure 7) select the **Xcode → Preferences…** to open the "Xcode Preferences" window in Figure 8. If the display is not similar to that shown, select the "Text Editing" icon.

2. If the "Show: Line numbers" check box is not checked, check it.

3. Close the window.



Show: ☑ Line numbers
☐ Code folding ribbon
☐ Page guide at column: 80
☑ Highlight instances of selected symbol
Delay: 0.25 seconds

Editor overscroll: Medium

Code completion: ☑ Suggest completions while typing
☑ Use Escape key to show completion suggestions
☑ Automatically insert closing braces ("}")
☑ Enable type-over completions
☑ Automatically balance brackets in Objective-C method calls
☑ Enclose selection in matching delimiters

While editing: ☑ Automatically trim trailing whitespace
☑ Including whitespace-only lines

Default text encoding: Unicode (UTF-8)
Default line endings: macOS / Unix (LF)
☐ Convert existing files on save

Code coverage: ☑ Show iteration counts

Figure 8

Using “Projects”

Most IDEs use the concept of a “project”, which consists of all source code files, settings, and other resources necessary to create a single working program. 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 using only one common project for everything and simply removing (but not delete) the previous exercise’s source code file(s) and creating and adding new file(s) for the next exercise as you progress through them.

Re-opening an Existing Project

To re-open an existing project any of the following techniques may be used. The first requires that Xcode already be open:

1. Select **File → Open...** from the Xcode menu (Figure 7) and use the resulting window to locate and open the desired project file. Xcode project files have a **.xcodeproj** extension.

2. Double-click on the project’s desktop icon (if you created one).

3. Use the “Finder” to locate and open the desired project file.

Creating a New Project

This example assumes that you wish to create a new empty project named **MyPrograms** in a folder named **Projects**, but you may use any legal name and location desired.

1. From the Xcode menu (Figure 7) select **File → New → Project...** to open the “New Project” window in Figure 9.

2. Select **macOS** at the top of the window and **Command Line Tool** from in the **Application** frame.

3. Click ***Next***, which will open the window in Figure 10 on the next page.

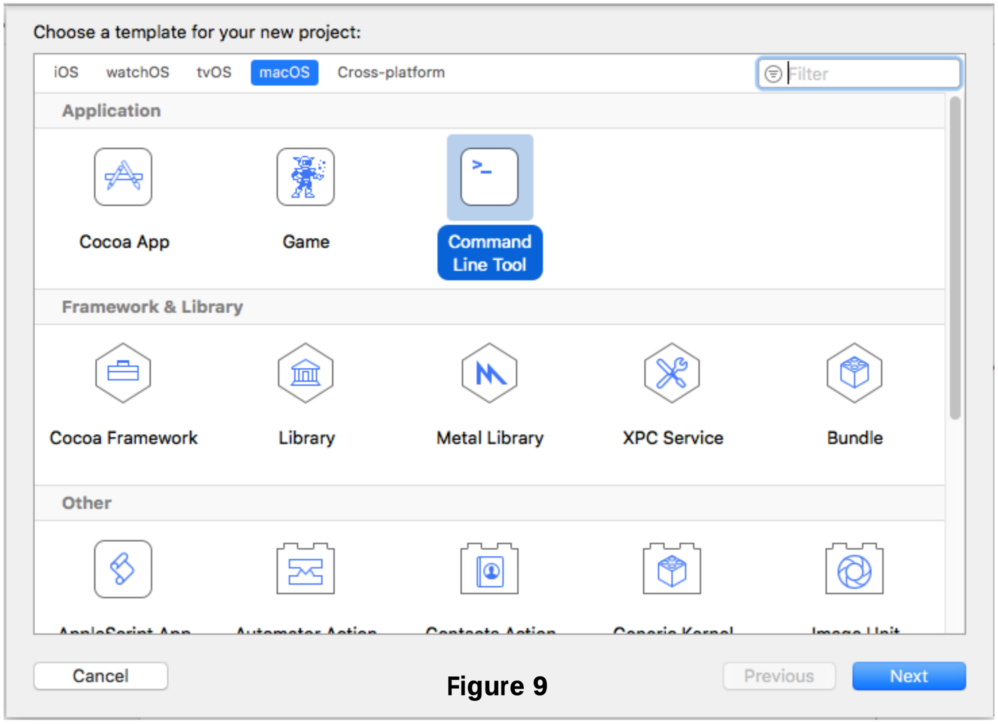

Choose a template for your new project:

Figure 9

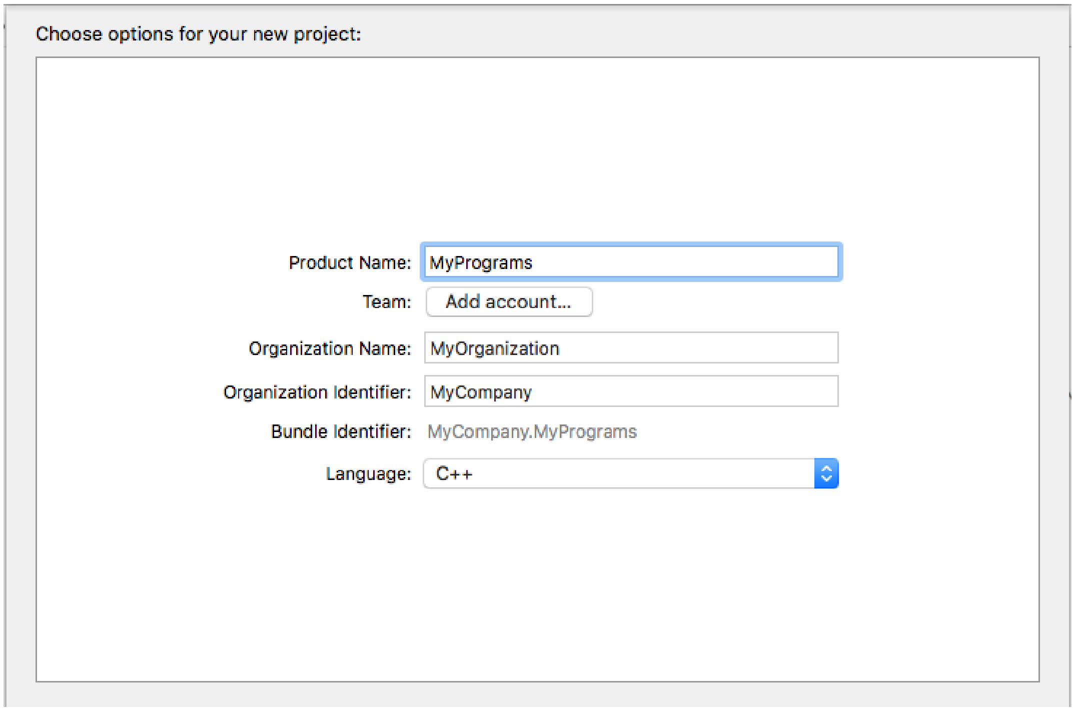
Continued on the next page...

Creating a New Project, continued

4. Enter the desired name for your project in the “Product Name” field (“MyPrograms” in this example) and whatever you wish in the “Organization Name” and “Company Identifier” fields. You may leave the “Team” field unaltered.

5. In the “Type” field selection “C++”. This configuration can be used for both C and C++ projects.

6. Click ***Next***, which opens the window in figure 11 on the next page.



Choose options for your new project:

Product Name: MyPrograms

Team: Add account...

Organization Name: MyOrganization

Organization Identifier: MyCompany

Bundle Identifier: MyCompany.MyPrograms

Language: C++

Figure 10

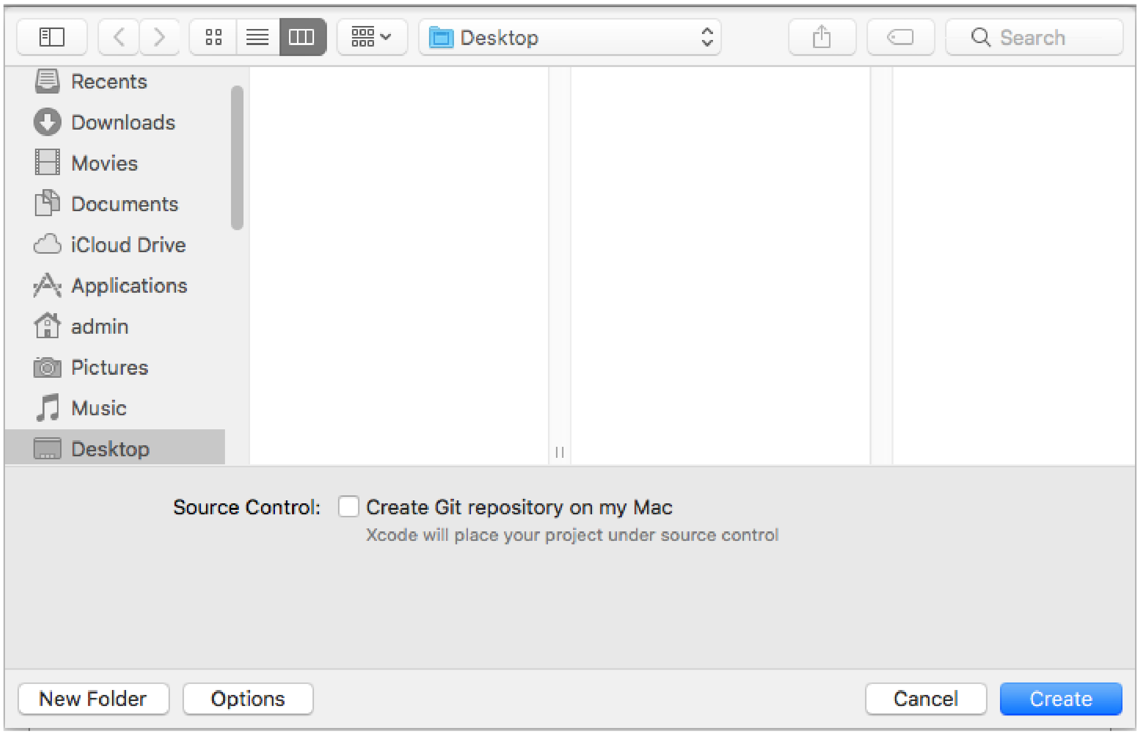
Continued on the next page...

Creating a New Project, continued

7. Make sure the “Source Control:” checkbox is <u>not</u> checked on the window in Figure 11 below.

8. To create the folder for your projects select the desired parent folder in the selection box at the top of the window (“Desktop” in this example).

9. Click ***New Folder***, which will open the window in Figure 12.

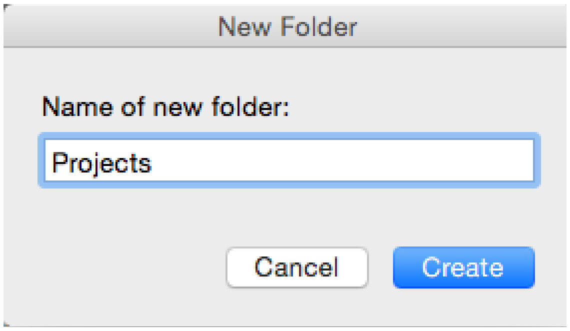


Source Control: ☐ Create Git repository on my Mac
Xcode will place your project under source control

Figure 11

10. Enter the name of the desired folder for your project in the “Name of new folder:” field (**Projects** in this case).

11. Click ***Create***, which will create the project folder and add it as shown in Figure 13 on the next page.



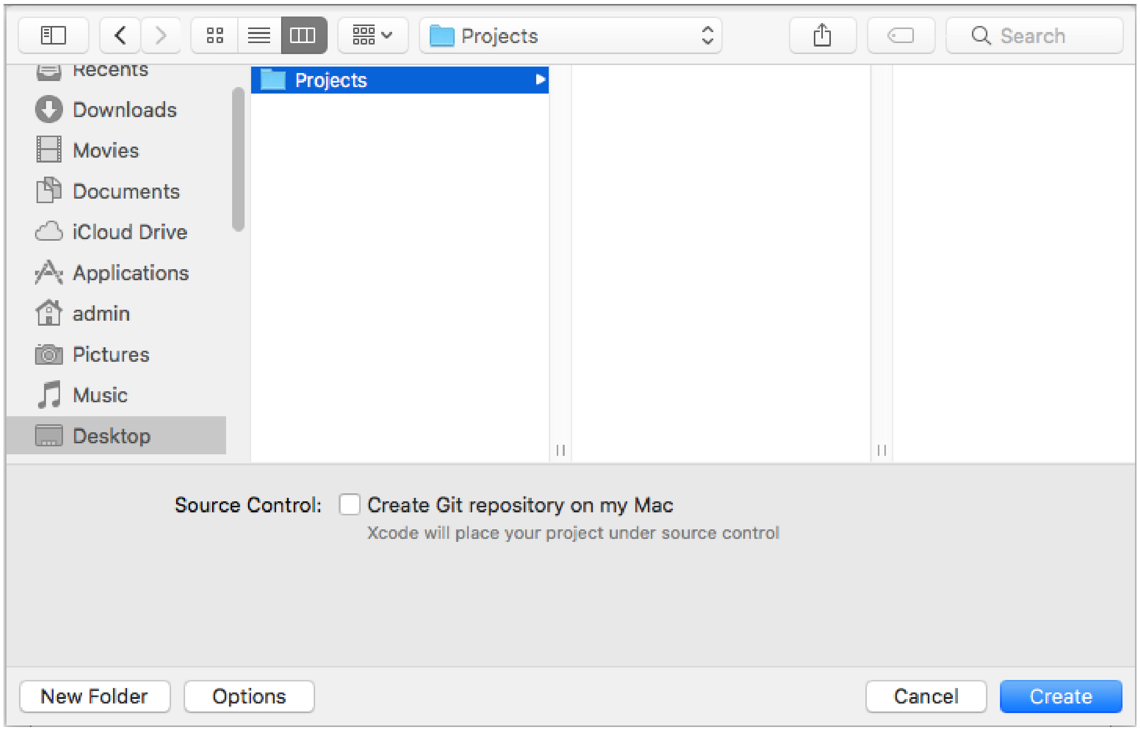
Name of new folder: Projects

Figure 12

Continued on the next page...

Creating a New Project, continued

12. Double-click the new folder, **Projects**, which will create the project in that folder and open the project window in Figure 14 on the next page.



Source Control: □ Create Git repository on my Mac
Xcode will place your project under source control

Figure 13

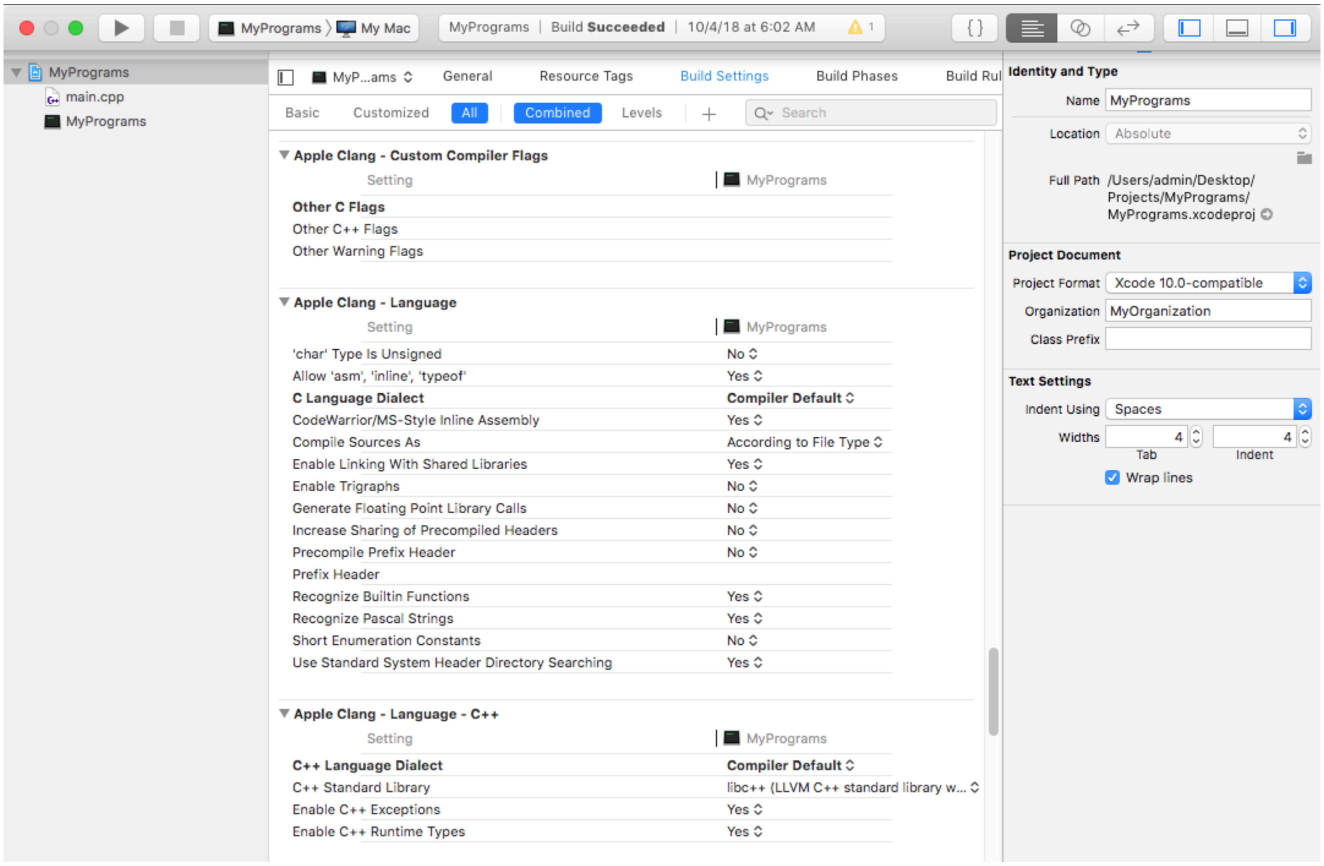
Continued on the next page...

Creating a New Project, continued

13. At this point I recommend creating a shortcut (alias) on your desktop for the project file that was just created in the folder you chose in Figure 12. That file will have the same name as the project itself with a **.xcodeproj** extension, that is, **MyPrograms.xcodeproj** in this case. This shortcut will allow you to easily open both Xcode and the project itself by merely double-clicking the icon.

14. Do the following to ensure the project window looks as shown below in Figure 14:
    a. Select "MyPrograms" in the "Project Navigator" frame;
    b. Select "Build Settings" near the top frame;
    c. Select "All" and "Combined" below "Build Settings";
    d. Scroll the middle frame up so the **Apple Clang – Custom Compiler Flags** group is first.

15. In the **Apple Clang – Custom Compiler Flags** group, double-click under "MyPrograms" opposite the setting named "Other C Flags", which opens the entry box in Figure 15 on the next page.



Apple Clang - Custom Compiler Flags

Setting	MyPrograms
Other C Flags	
Other C++ Flags	
Other Warning Flags	

Apple Clang - Language

Setting	MyPrograms
'char' Type Is Unsigned	No
Allow 'asm', 'inline', 'typeof'	Yes
C Language Dialect	**Compiler Default**
CodeWarrior/MS-Style Inline Assembly	Yes
Compile Sources As	According to File Type
Enable Linking With Shared Libraries	Yes
Enable Trigraphs	No
Generate Floating Point Library Calls	No
Increase Sharing of Precompiled Headers	No
Precompile Prefix Header	No
Prefix Header	
Recognize Builtin Functions	Yes
Recognize Pascal Strings	Yes
Short Enumeration Constants	No
Use Standard System Header Directory Searching	Yes

Apple Clang - Language - C++

Setting	MyPrograms
C++ Language Dialect	**Compiler Default**
C++ Standard Library	libc++ (LLVM C++ standard library w...
Enable C++ Exceptions	Yes
Enable C++ Runtime Types	Yes

Identity and Type

Name: MyPrograms
Location: Absolute
Full Path: /Users/admin/Desktop/Projects/MyPrograms/MyPrograms.xcodeproj

Project Document

Project Format: Xcode 10.0-compatible
Organization: MyOrganization
Class Prefix:

Text Settings

Indent Using: Spaces
Widths: 4 Tab, 4 Indent
☑ Wrap lines

Figure 14

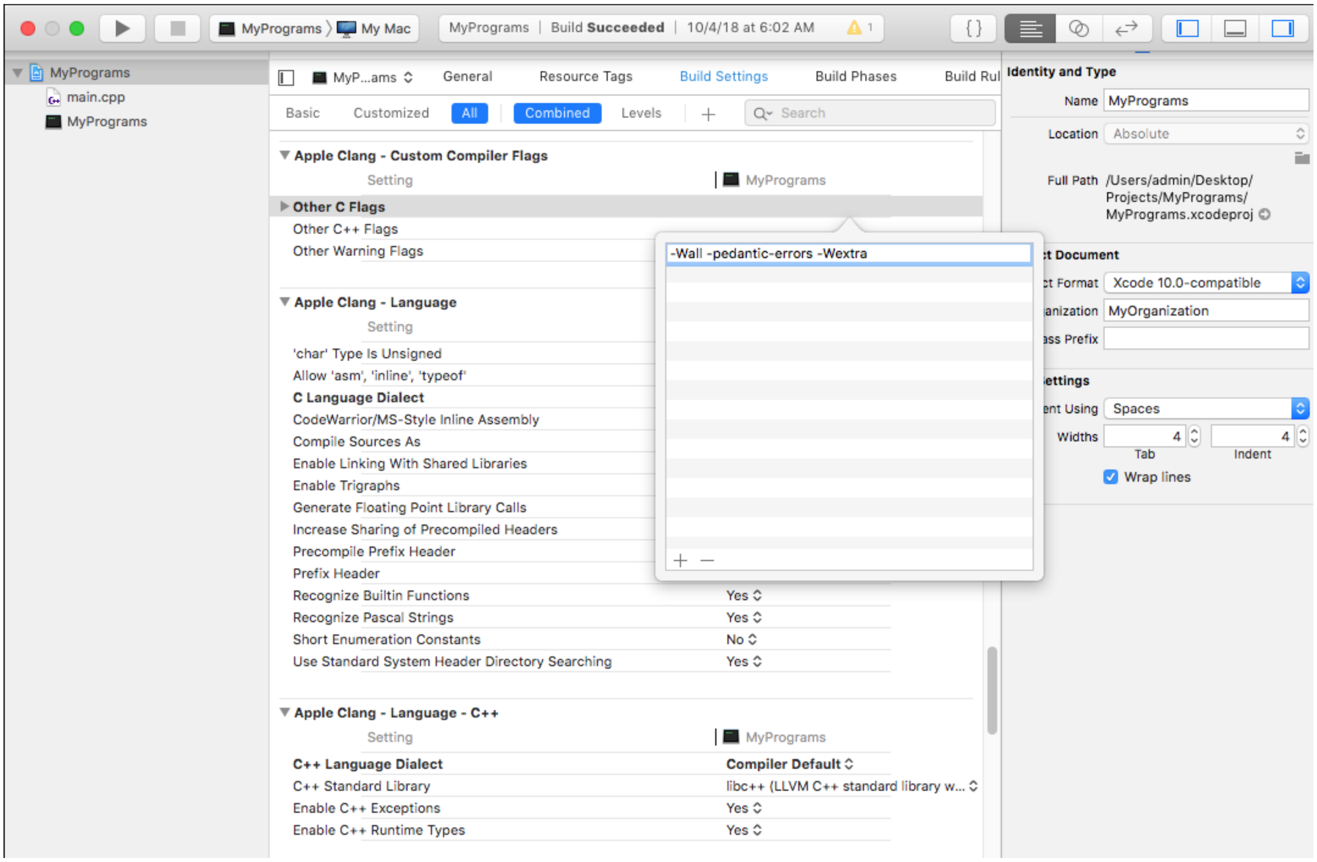
Continued on the next page...

Creating a New Project, continued

16. Click the plus sign in the lower left corner of the entry box and enter the following text on the first line as shown in Figure 15: **-Wall -pedantic-errors -Wextra**

17. Click anywhere outside that box to close it.

18. The "Other C Flags" and "Other C++ Flags" items should then contain the entered information, as shown in Figure 16 on the next page.



MyPrograms 〉 My Mac | MyPrograms | Build Succeeded | 10/4/18 at 6:02 AM ⚠ 1

MyPrograms
- main.cpp
- MyPrograms

MyP...ams ◇ | General | Resource Tags | **Build Settings** | Build Phases | Build Rul

Basic | Customized | **All** | **Combined** | Levels | + | Search

▼ Apple Clang - Custom Compiler Flags

Setting	MyPrograms
▶ Other C Flags	
Other C++ Flags	
Other Warning Flags	

-Wall -pedantic-errors -Wextra

▼ Apple Clang - Language

Setting	
'char' Type Is Unsigned	
Allow 'asm', 'inline', 'typeof'	
C Language Dialect	
CodeWarrior/MS-Style Inline Assembly	
Compile Sources As	
Enable Linking With Shared Libraries	
Enable Trigraphs	
Generate Floating Point Library Calls	
Increase Sharing of Precompiled Headers	
Precompile Prefix Header	
Prefix Header	
Recognize Builtin Functions	Yes ◇
Recognize Pascal Strings	Yes ◇
Short Enumeration Constants	No ◇
Use Standard System Header Directory Searching	Yes ◇

▼ Apple Clang - Language - C++

Setting	MyPrograms
C++ Language Dialect	**Compiler Default** ◇
C++ Standard Library	libc++ (LLVM C++ standard library w... ◇
Enable C++ Exceptions	Yes ◇
Enable C++ Runtime Types	Yes ◇

Identity and Type

Name: MyPrograms

Location: Absolute

Full Path: /Users/admin/Desktop/Projects/MyPrograms/MyPrograms.xcodeproj

t Document

t Format: Xcode 10.0-compatible

anization: MyOrganization

ass Prefix:

ettings

ent Using: Spaces

Widths: 4 Tab, 4 Indent

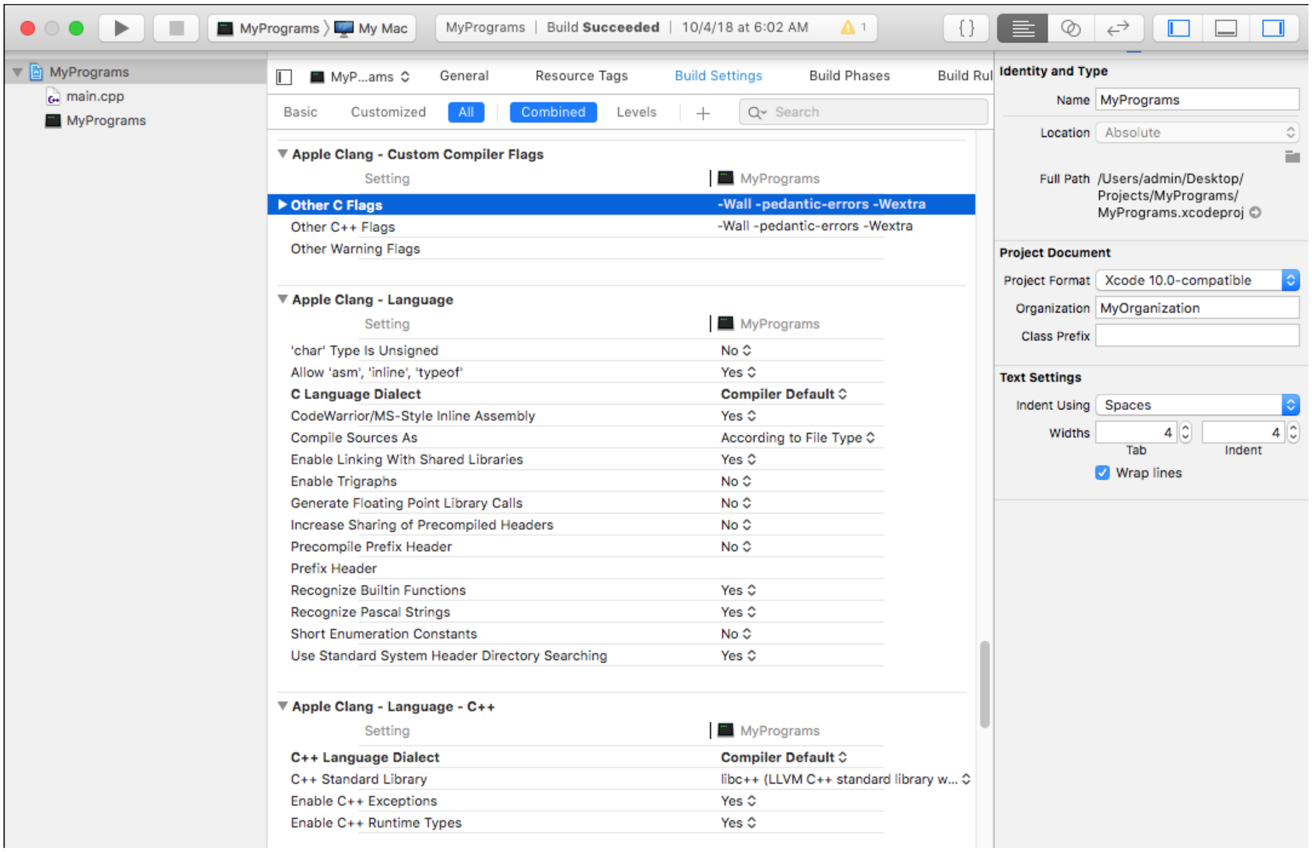
☑ Wrap lines

Figure 15

Continued on the next page...

Creating a New Project, continued

19. Figure 16 below shows the result of the previous step.



▼ Apple Clang - Custom Compiler Flags

Setting	MyPrograms
▶ Other C Flags	-Wall -pedantic-errors -Wextra
Other C++ Flags	-Wall -pedantic-errors -Wextra
Other Warning Flags	

▼ Apple Clang - Language

Setting	MyPrograms
'char' Type Is Unsigned	No
Allow 'asm', 'inline', 'typeof'	Yes
C Language Dialect	**Compiler Default**
CodeWarrior/MS-Style Inline Assembly	Yes
Compile Sources As	According to File Type
Enable Linking With Shared Libraries	Yes
Enable Trigraphs	No
Generate Floating Point Library Calls	No
Increase Sharing of Precompiled Headers	No
Precompile Prefix Header	No
Prefix Header	
Recognize Builtin Functions	Yes
Recognize Pascal Strings	Yes
Short Enumeration Constants	No
Use Standard System Header Directory Searching	Yes

▼ Apple Clang - Language - C++

Setting	MyPrograms
C++ Language Dialect	**Compiler Default**
C++ Standard Library	libc++ (LLVM C++ standard library w...
Enable C++ Exceptions	Yes
Enable C++ Runtime Types	Yes

Identity and Type

Name: MyPrograms
Location: Absolute
Full Path: /Users/admin/Desktop/Projects/MyPrograms/MyPrograms.xcodeproj

Project Document

Project Format: Xcode 10.0-compatible
Organization: MyOrganization
Class Prefix:

Text Settings

Indent Using: Spaces
Widths: 4 (Tab) 4 (Indent)
☑ Wrap lines

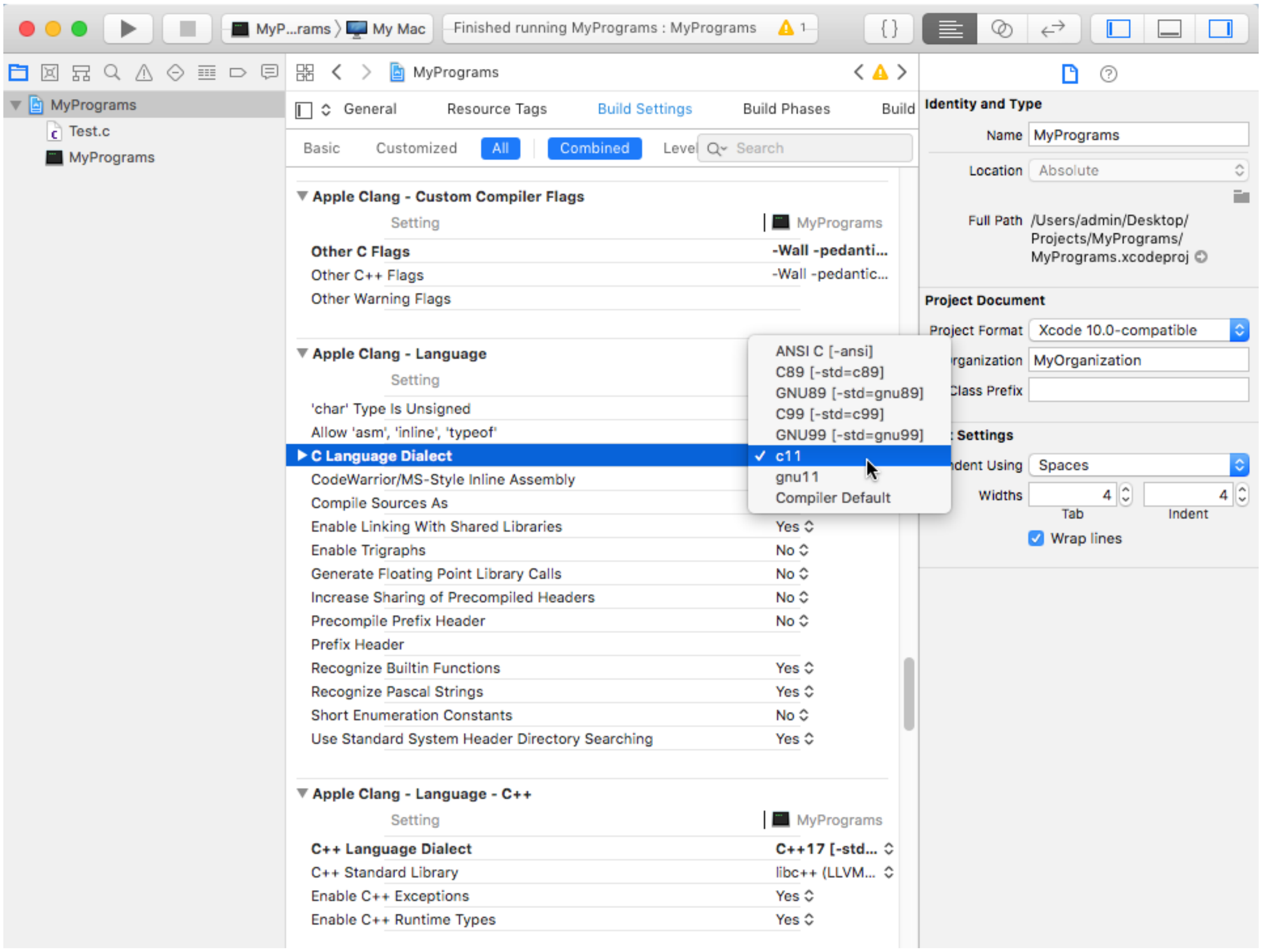
Figure 16

Continued on the next page...

Creating a New Project, continued

20. In this step you can select the version of the C language standard and extensions you want the compiler to enforce. Selecting an older standard with no extensions will provide the best compatibility with other compilers while selecting a newer standard will provide more features but potentially less compatibility. I recommend C11 (not GNU11) for this course.

21. To make your selection single-click the right column of the "C Language Dialect" item in the **Apple Clang – Language** group and make your choice from the resulting selection box as shown in Figure 17 below.



▼ Apple Clang - Custom Compiler Flags

Setting	MyPrograms
Other C Flags	**-Wall -pedanti...**
Other C++ Flags	-Wall -pedantic...
Other Warning Flags	

▼ Apple Clang - Language

Setting	
'char' Type Is Unsigned	
Allow 'asm', 'inline', 'typeof'	
C Language Dialect	ANSI C [-ansi] C89 [-std=c89] GNU89 [-std=gnu89] C99 [-std=c99] GNU99 [-std=gnu99] ✓ c11 gnu11 Compiler Default
CodeWarrior/MS-Style Inline Assembly	
Compile Sources As	
Enable Linking With Shared Libraries	Yes
Enable Trigraphs	No
Generate Floating Point Library Calls	No
Increase Sharing of Precompiled Headers	No
Precompile Prefix Header	No
Prefix Header	
Recognize Builtin Functions	Yes
Recognize Pascal Strings	Yes
Short Enumeration Constants	No
Use Standard System Header Directory Searching	Yes

▼ Apple Clang - Language - C++

Setting	MyPrograms
C++ Language Dialect	**C++17 [-std...**
C++ Standard Library	libc++ (LLVM...
Enable C++ Exceptions	Yes
Enable C++ Runtime Types	Yes

Identity and Type

Name: MyPrograms
Location: Absolute
Full Path: /Users/admin/Desktop/Projects/MyPrograms/MyPrograms.xcodeproj

Project Document

Project Format: Xcode 10.0-compatible
rganization: MyOrganization
Class Prefix:

Settings

dent Using: Spaces
Widths: 4 Tab, 4 Indent
☑ Wrap lines

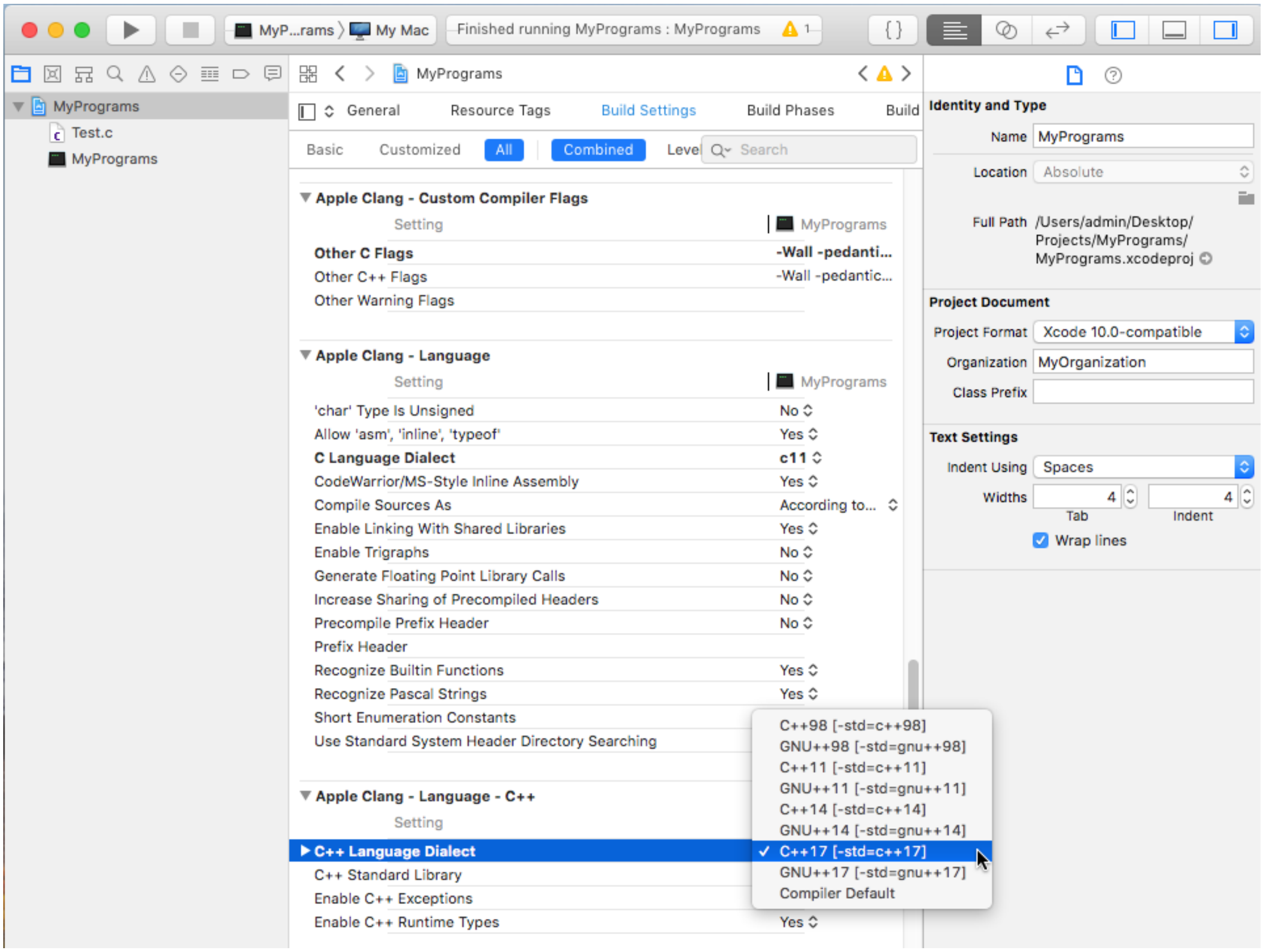
Figure 17

Continued on the next page...

Creating a New Project, continued

22. In this step you can select the version of the C++ language standard and extensions you want the compiler to enforce. Selecting an older standard with no extensions will provide the best compatibility with other compilers while selecting a newer standard will provide more features but potentially less compatibility. I recommend C++17 (not GNU++17) for this course.

23. To make your selection single-click the right column of the "C++ Language Dialect" item in the **Apple Clang – Language – C++** group and make your choice from the resulting selection box as shown in Figure 18 below.



Apple Clang - Custom Compiler Flags

Setting	MyPrograms
Other C Flags	**-Wall -pedanti...**
Other C++ Flags	-Wall -pedantic...
Other Warning Flags	

Apple Clang - Language

Setting	MyPrograms
'char' Type Is Unsigned	No ◇
Allow 'asm', 'inline', 'typeof'	Yes ◇
C Language Dialect	**c11 ◇**
CodeWarrior/MS-Style Inline Assembly	Yes ◇
Compile Sources As	According to... ◇
Enable Linking With Shared Libraries	Yes ◇
Enable Trigraphs	No ◇
Generate Floating Point Library Calls	No ◇
Increase Sharing of Precompiled Headers	No ◇
Precompile Prefix Header	No ◇
Prefix Header	
Recognize Builtin Functions	Yes ◇
Recognize Pascal Strings	Yes ◇
Short Enumeration Constants	
Use Standard System Header Directory Searching	

Apple Clang - Language - C++

Setting	
▶ C++ Language Dialect	
C++ Standard Library	
Enable C++ Exceptions	
Enable C++ Runtime Types	Yes ◇

C++98 [-std=c++98]
GNU++98 [-std=gnu++98]
C++11 [-std=c++11]
GNU++11 [-std=gnu++11]
C++14 [-std=c++14]
GNU++14 [-std=gnu++14]
✓ C++17 [-std=c++17]
GNU++17 [-std=gnu++17]
Compiler Default

Identity and Type

Name: MyPrograms
Location: Absolute
Full Path: /Users/admin/Desktop/Projects/MyPrograms/MyPrograms.xcodeproj

Project Document

Project Format: Xcode 10.0-compatible
Organization: MyOrganization
Class Prefix:

Text Settings

Indent Using: Spaces
Widths: 4 Tab, 4 Indent
☑ Wrap lines

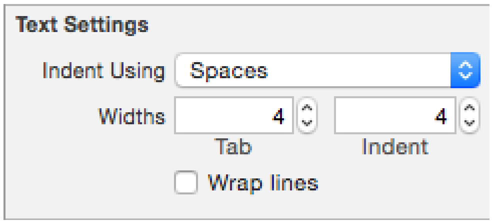
Figure 18

Continued on the next page...

Creating a New Project, continued

Changing the Text Editor Tab Settings

24. Note the “Text Settings” item in the right frame of previous Figure 18, shown enlarged in Figure 19 below. 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 indent widths and to insert spaces each time the keyboard Tab key is pressed. Although 4-column tab stops and indent widths are fine and are the most common, you may choose a different common value if you wish. However, the use of “Spaces” for indenting is mandatory in this course. The other choice is “Tabs”, which are not allowed in this course and are discouraged in general because the actual size of a 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. If necessary, make appropriate changes to the “Text Settings”, making sure the “Wrap lines” checkbox is not checked.



Text Settings

Indent Using: Spaces

Widths: Tab 4, Indent 4

□ Wrap lines

Figure 19

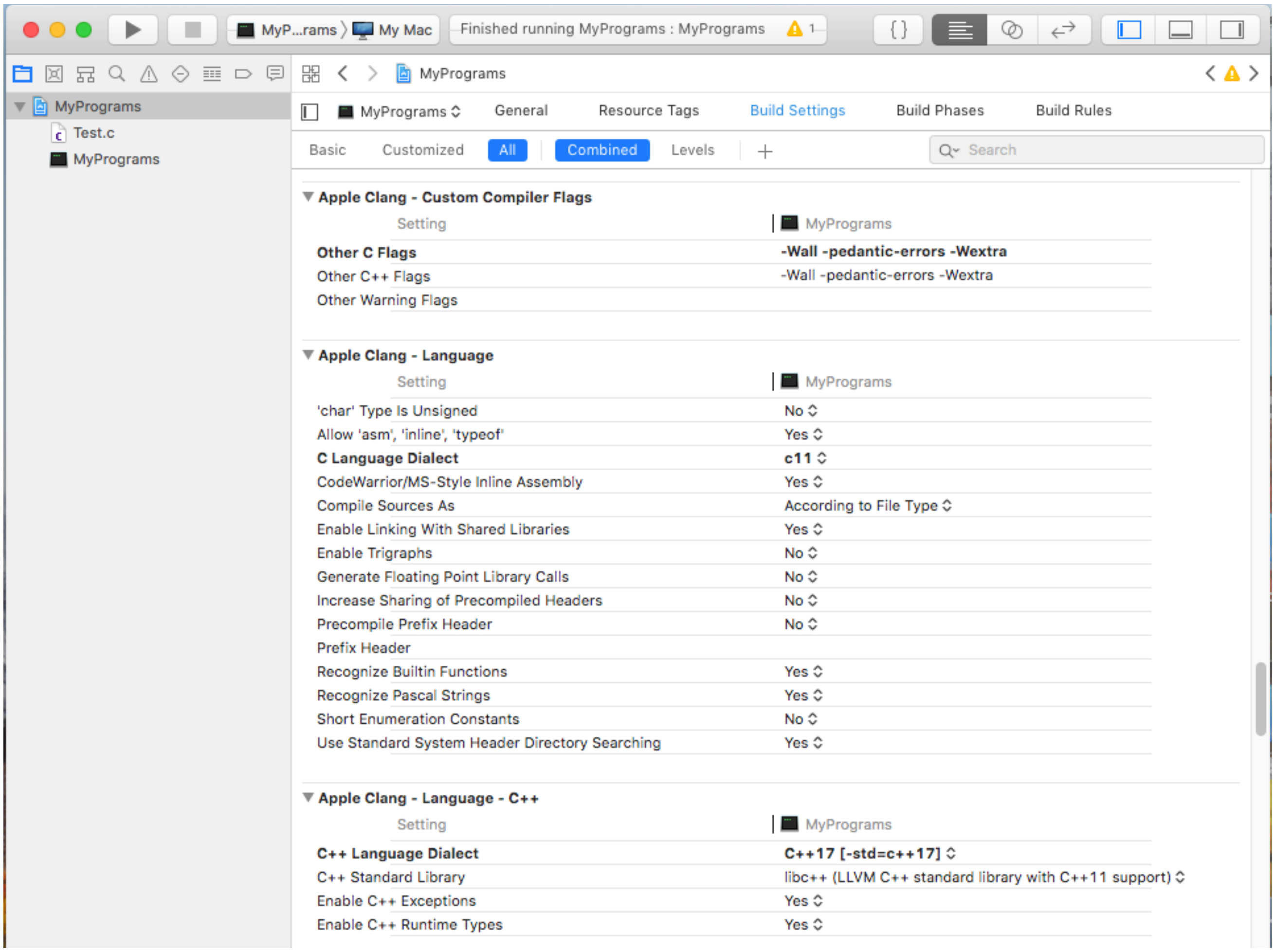
Removing Existing “Hard” Tabs

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

Continued on the next page...

Creating a New Project, continued

25. To avoid unnecessary clutter, select **View → Inspectors → Hide Inspectors** from the Xcode menu (Figure 7). This will hide the inspectors frame and allow more space for other things, as shown in Figure 20.



Apple Clang - Custom Compiler Flags

Setting	MyPrograms
Other C Flags	**-Wall -pedantic-errors -Wextra**
Other C++ Flags	-Wall -pedantic-errors -Wextra
Other Warning Flags	

Apple Clang - Language

Setting	MyPrograms
'char' Type Is Unsigned	No ◇
Allow 'asm', 'inline', 'typeof'	Yes ◇
C Language Dialect	**c11 ◇**
CodeWarrior/MS-Style Inline Assembly	Yes ◇
Compile Sources As	According to File Type ◇
Enable Linking With Shared Libraries	Yes ◇
Enable Trigraphs	No ◇
Generate Floating Point Library Calls	No ◇
Increase Sharing of Precompiled Headers	No ◇
Precompile Prefix Header	No ◇
Prefix Header	
Recognize Builtin Functions	Yes ◇
Recognize Pascal Strings	Yes ◇
Short Enumeration Constants	No ◇
Use Standard System Header Directory Searching	Yes ◇

Apple Clang - Language - C++

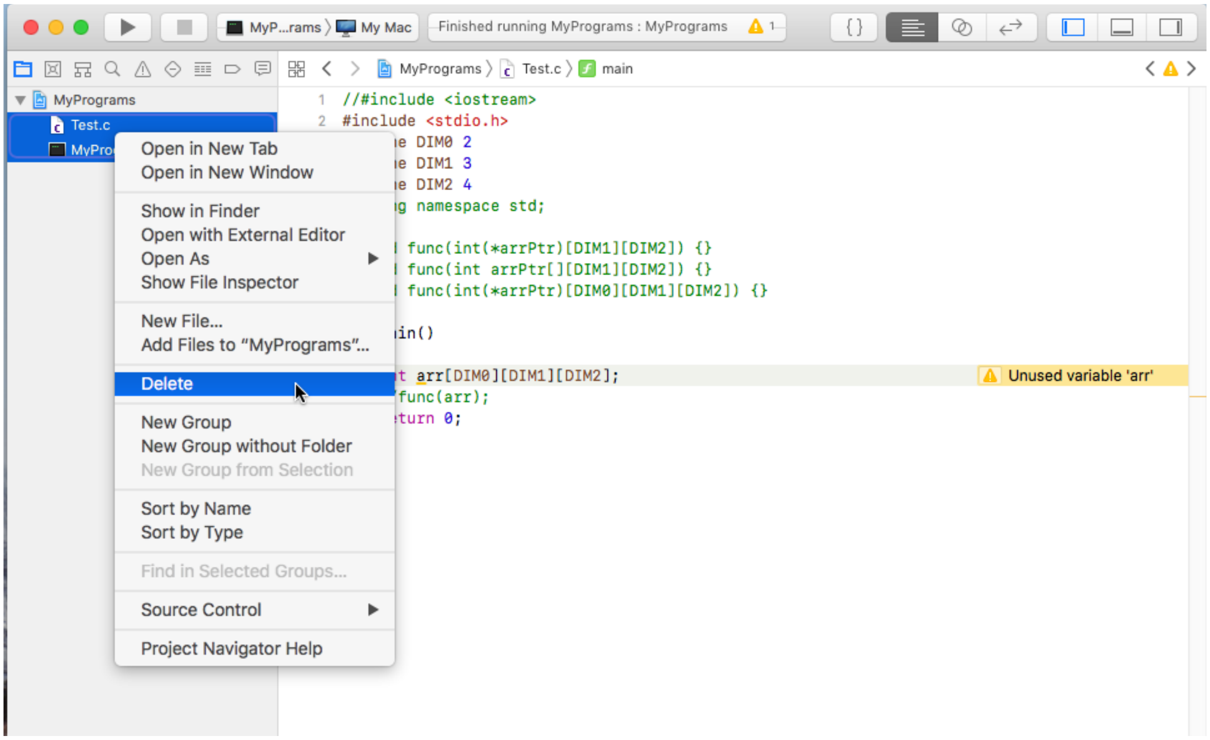
Setting	MyPrograms
C++ Language Dialect	**C++17 [-std=c++17] ◇**
C++ Standard Library	libc++ (LLVM C++ standard library with C++11 support) ◇
Enable C++ Exceptions	Yes ◇
Enable C++ Runtime Types	Yes ◇

Figure 20

Continued on the next page...

Creating a New Project, continued

26. To clean the unnecessary information from the project you created, select everything below the **MyPrograms** item with the blue icon in the “Project Navigator” frame in Figure 21, right-click on the selection to expose the context menu, then select “Delete” from that menu. This will open the window in Figure 22. (The items to be deleted on your machine may differ from those shown below.)

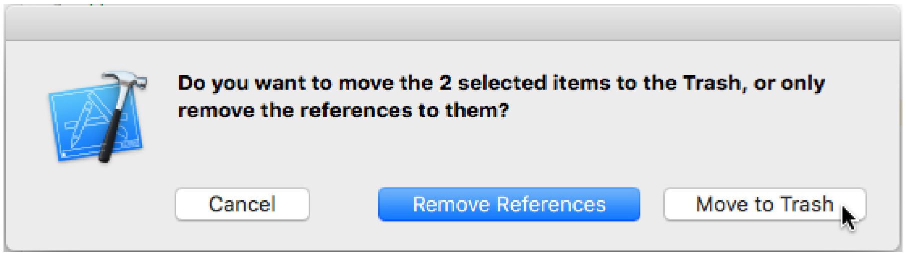


```
//#include <iostream>
#include <stdio.h>
e DIM0 2
e DIM1 3
e DIM2 4
g namespace std;

func(int(*arrPtr)[DIM1][DIM2]) {}
func(int arrPtr[][DIM1][DIM2]) {}
func(int(*arrPtr)[DIM0][DIM1][DIM2]) {}

in()

t arr[DIM0][DIM1][DIM2];
func(arr);
turn 0;
```

Unused variable 'arr'

Figure 21

27. Click “Move to Trash” to remove the items from the project and the “Project Navigator” frame of the project window.



Do you want to move the 2 selected items to the Trash, or only remove the references to them?

Figure 22

28. The project is now ready for you to specify its “Working Directory”.

Creating a New Project, continued

Determining/Changing a Project's "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.xcodeproj** will have been automatically created in a directory named **MyPrograms**. That directory is known as the "project directory" and by default earlier versions of Xcode used it as the working directory for any program run by that project. However, newer versions of Xcode use a different working directory and a simple way to empirically determine it or 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 21 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. Select **Product → Scheme → Edit Scheme...** from the Xcode menu (Figure 7) as shown in Figure 23. This will open the scheme editing window shown in Figure 24.

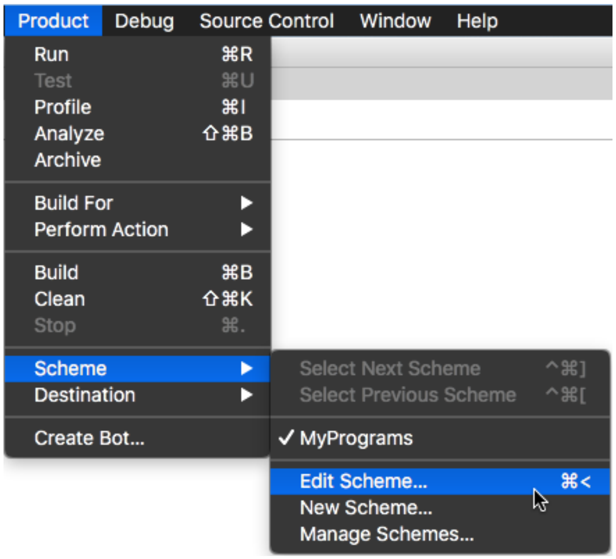
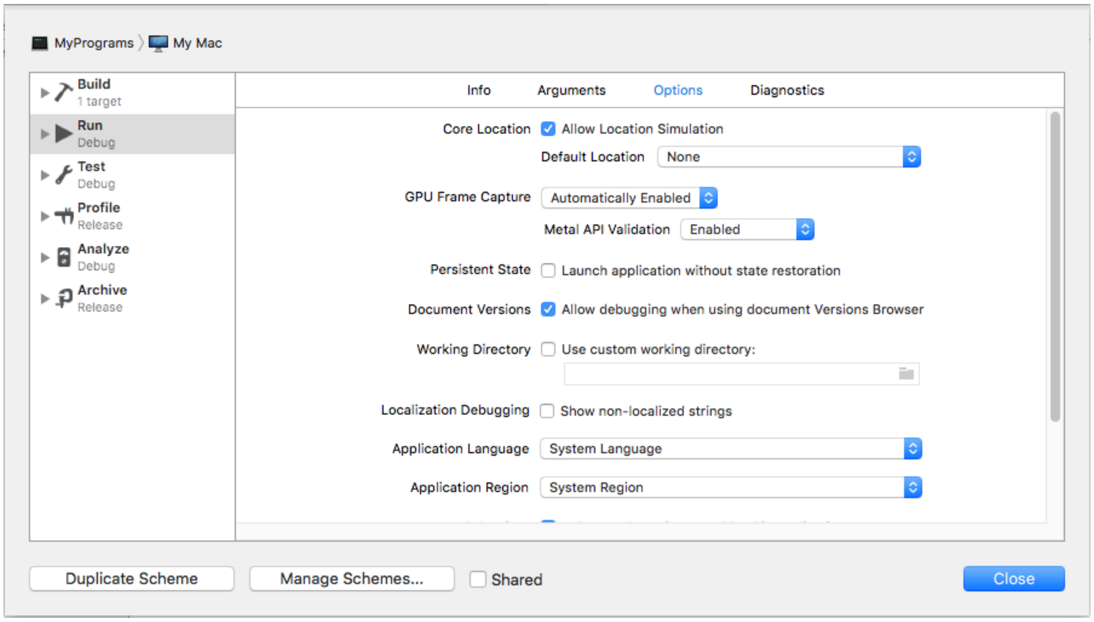


Figure 23

2. Select the ***Run Debug*** item in the left frame and the ***Options*** item at the top of the right frame.



Core Location ☑ Allow Location Simulation

Default Location None

GPU Frame Capture Automatically Enabled

Metal API Validation Enabled

Persistent State □ Launch application without state restoration

Document Versions ☑ Allow debugging when using document Versions Browser

Working Directory □ Use custom working directory:

Localization Debugging □ Show non-localized strings

Application Language System Language

Application Region System Region

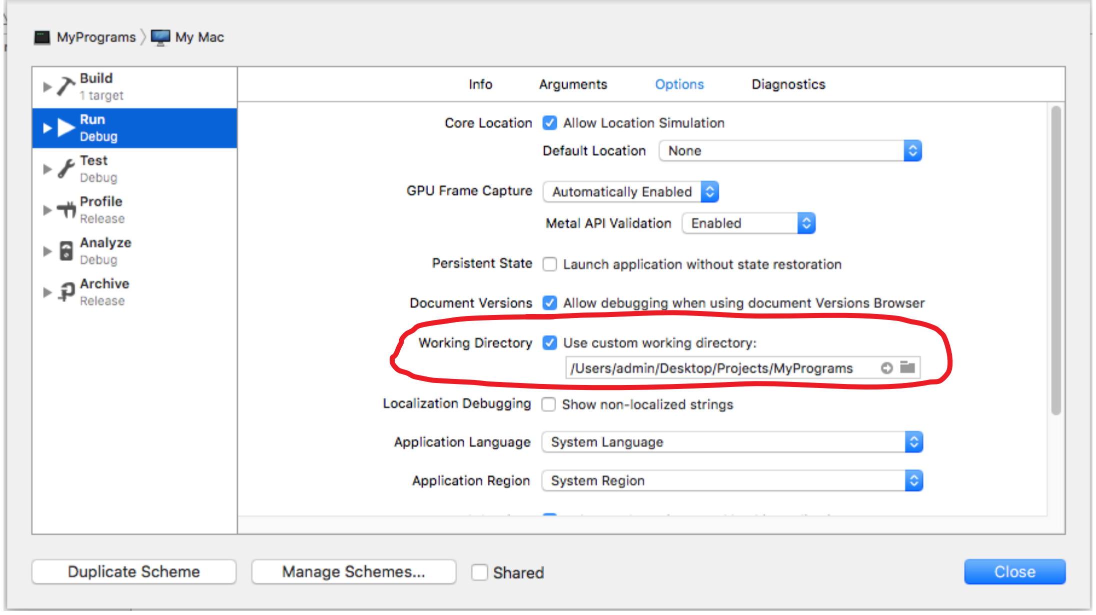
□ Shared

Figure 24

Continued on the next page...

Determining/Changing a Project's "Working Directory", continued

3. As shown in Figure 25, place a check in the ***Working Directory*** checkbox, which will enable the ***Use custom working directory:*** entry field and its corresponding navigation icon.
4. Either navigate to the desired directory on your machine and select it, or enter its path manually. In this case I've selected the project directory for the **MyPrograms** project.
5. Click ***Close*** to close the window and accept the change.
6. Your program will now use this directory for any "pathless" files it opens or creates.



Core Location ☑ Allow Location Simulation

Default Location: None

GPU Frame Capture: Automatically Enabled

Metal API Validation: Enabled

Persistent State ☐ Launch application without state restoration

Document Versions ☑ Allow debugging when using document Versions Browser

Working Directory ☑ Use custom working directory:

/Users/admin/Desktop/Projects/MyPrograms

Localization Debugging ☐ Show non-localized strings

Application Language: System Language

Application Region: System Region

☐ Shared

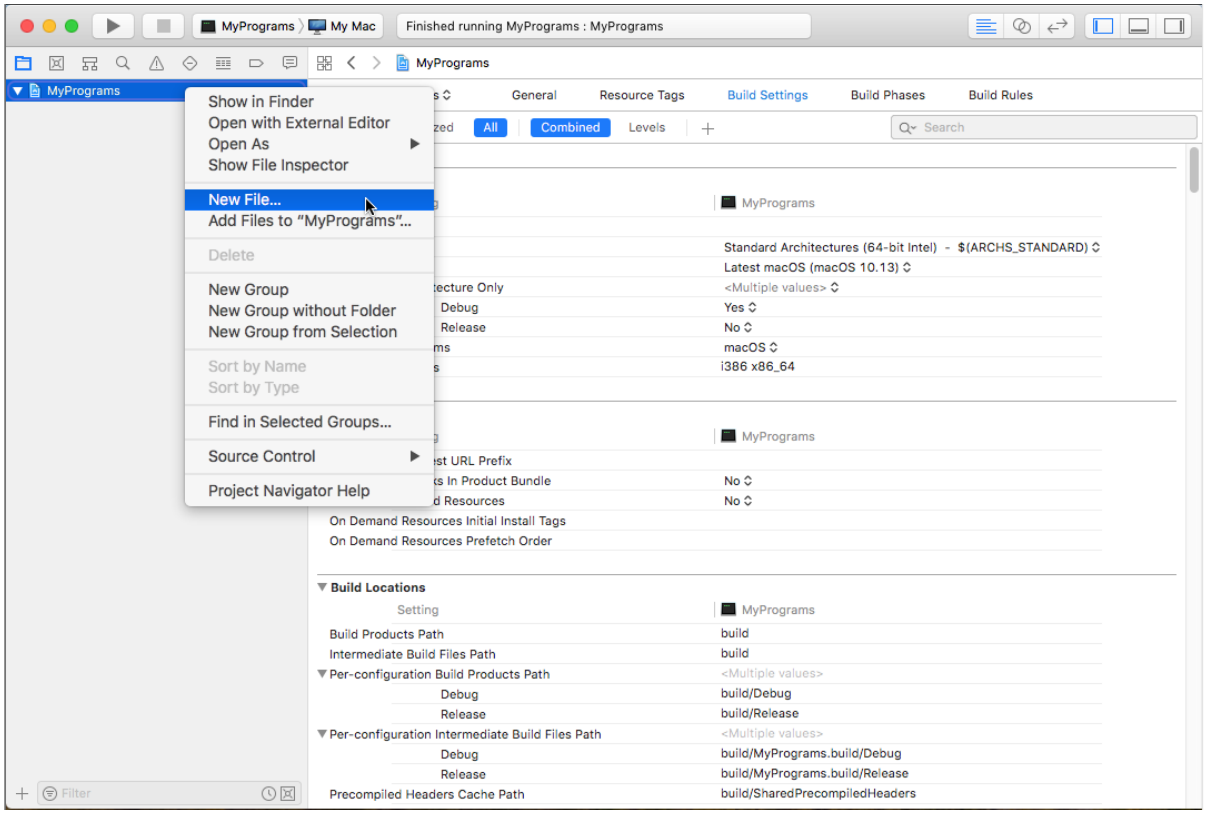
Figure 25

Adding Source Code Files to a Project

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

To create a <u>new source code file</u> and add it to the project:

1. As shown in Figure 26 below, right-click the project name in the "Project Navigator" frame and select **New File…** from the resulting context menu. This opens the "new file" window in Figure 27 on the next page.



Setting	MyPrograms
Build Products Path	build
Intermediate Build Files Path	build
Per-configuration Build Products Path	<Multiple values>
Debug	build/Debug
Release	build/Release
Per-configuration Intermediate Build Files Path	<Multiple values>
Debug	build/MyPrograms.build/Debug
Release	build/MyPrograms.build/Release
Precompiled Headers Cache Path	build/SharedPrecompiledHeaders

Figure 26

Continued on the next page…

Adding Code Files to a Project, continued

2. Select **macOS** at the top of the window and the type of file you want to create in the **Source** frame. In this example a **C** implementation file is selected.

3. Click ***Next***, which opens the file naming window in Figure 28.

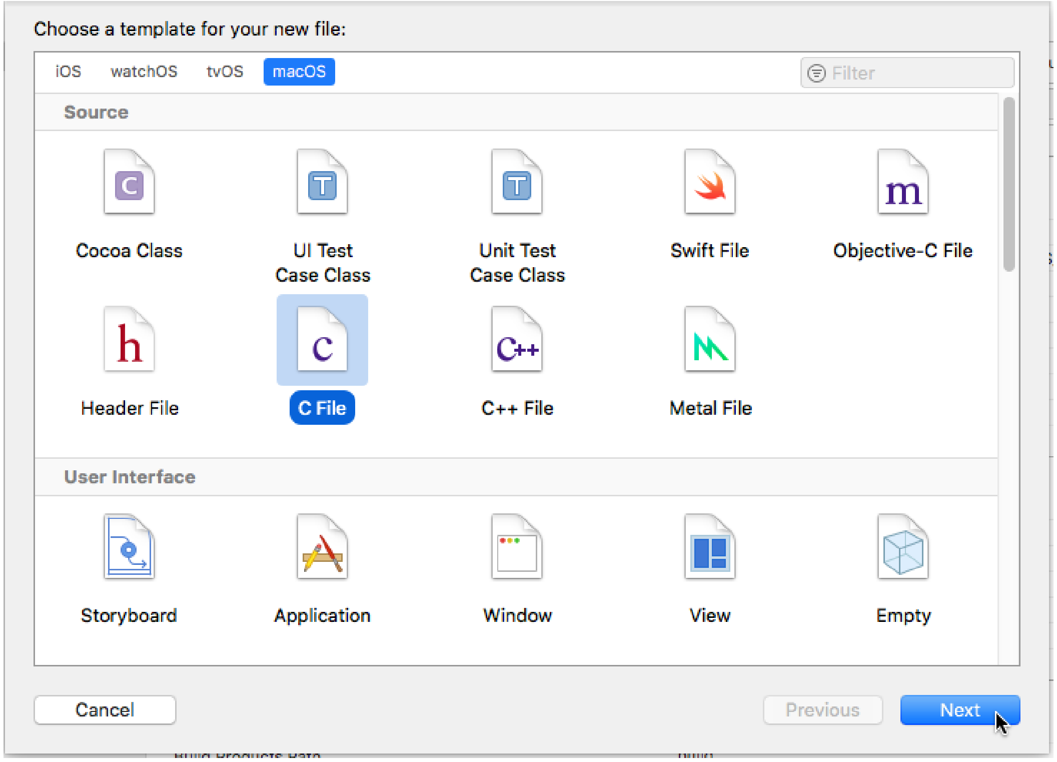
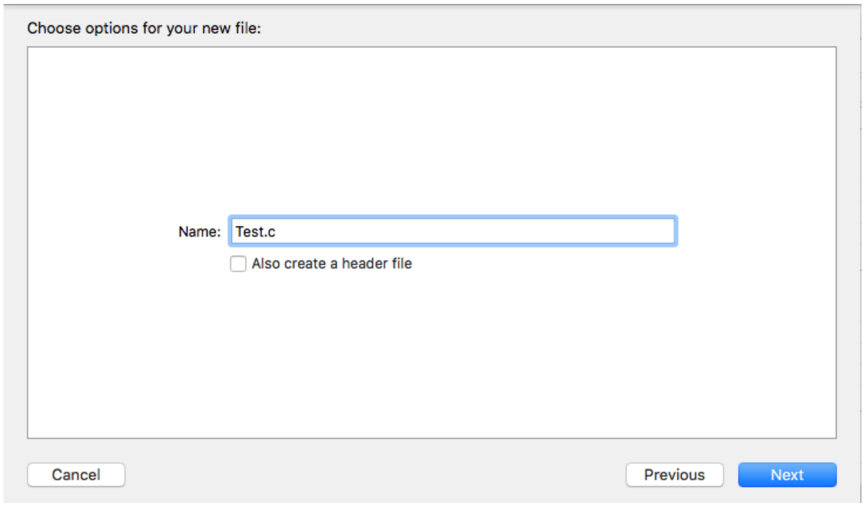


Figure 27

4. Type the name of the file you want to create into the "Name:" field and <u>uncheck</u> the "Also create a header file" checkbox. In this example the name "Test.c" was used.

5. Click ***Next***, which opens the window in Figure 29 on the next page.


Choose options for your new file:

Name: Test.c

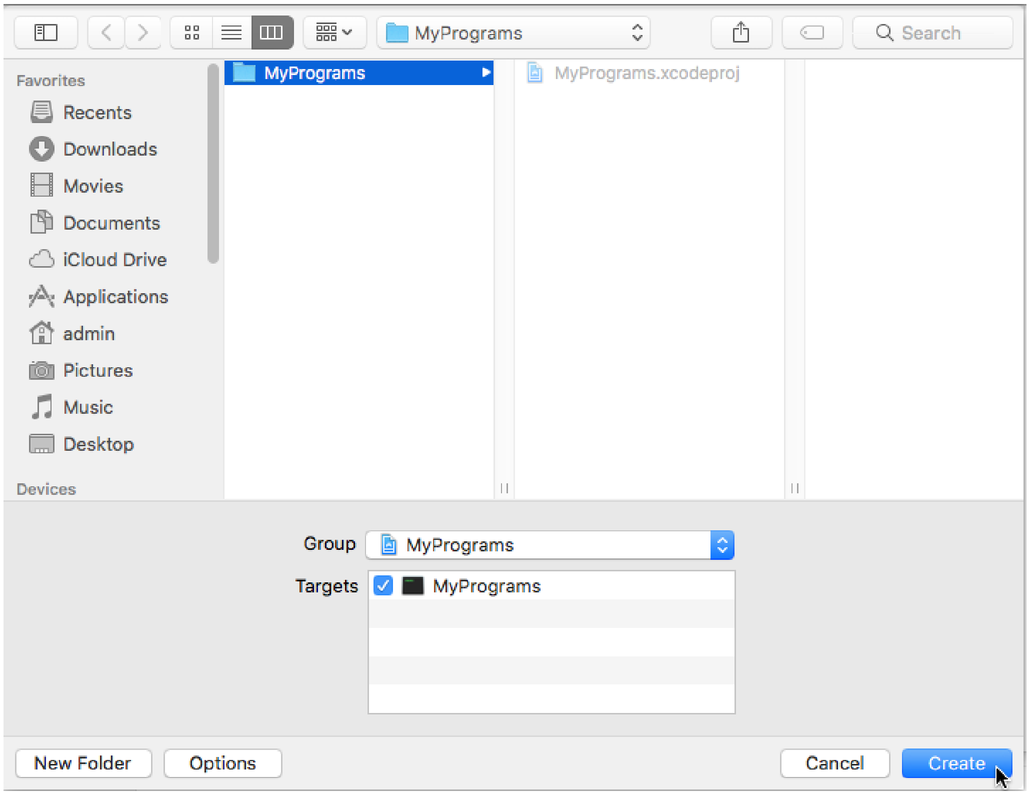
☐ Also create a header file

Figure 28

Continued on the next page...

Adding Code Files to a Project, continued

6. Click ***Create*** to close the window to create the file. The project window should now look as shown in Figure 30.



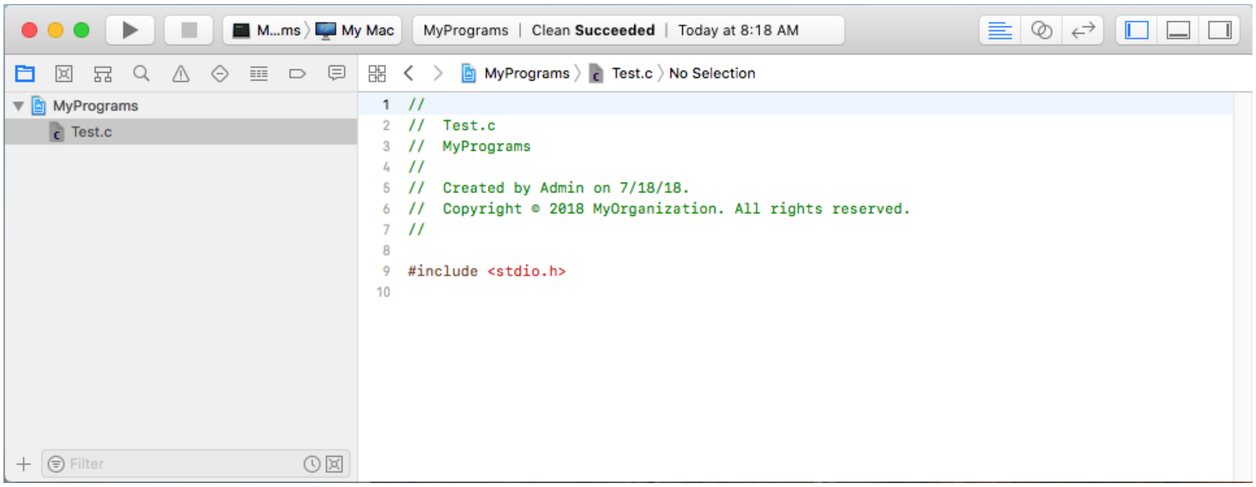
Group: MyPrograms

Targets: ☑ MyPrograms

Figure 29

7. This is the project window after file “Test.c” was created and added to the project. If the contents of the file are not visible simply select the file in the “Project Navigator” frame.

8. To create and add additional new files merely repeat the process.



M...ms 〉 My Mac
MyPrograms | Clean Succeeded | Today at 8:18 AM
MyPrograms 〉 Test.c 〉 No Selection
MyPrograms
Test.c
Filter

```
//
//  Test.c
//  MyPrograms
//
//  Created by Admin on 7/18/18.
//  Copyright © 2018 MyOrganization. All rights reserved.
//

#include <stdio.h>

```

Figure 30

Continued on the next page...

Adding Code Files to a Project, continued

To add an __existing source code file__ to the project:

1. Right-click the project name in the “Project Navigator” frame and select **Add Files to “MyPrograms”...** from resulting context menu, as shown in Figure 31 below. This opens the file selection window in Figure 32 on the next page.

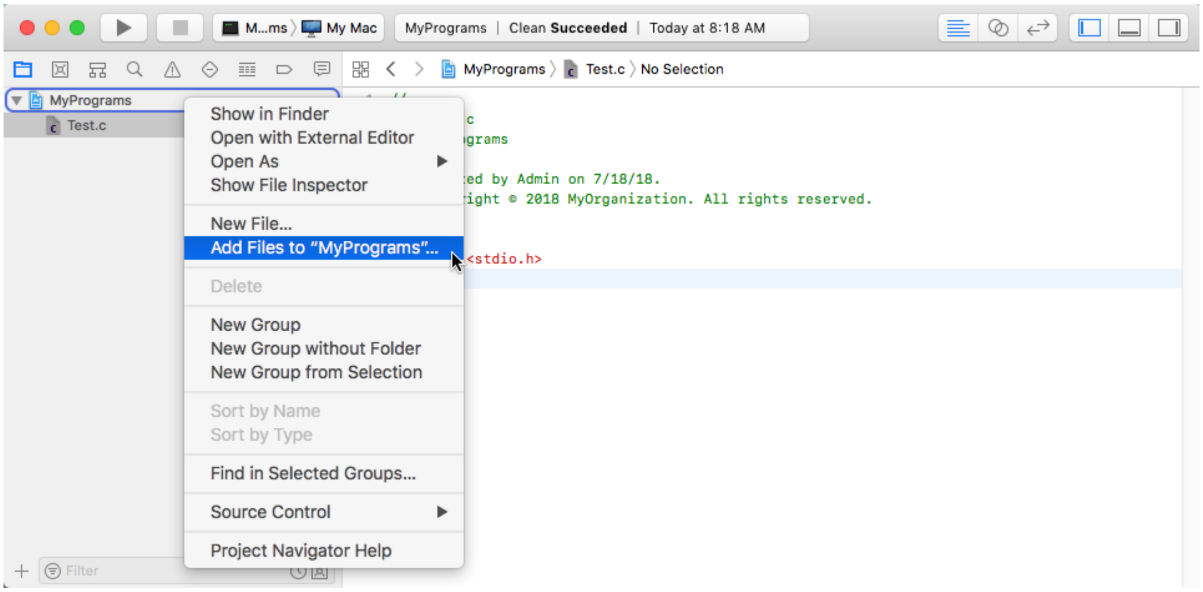
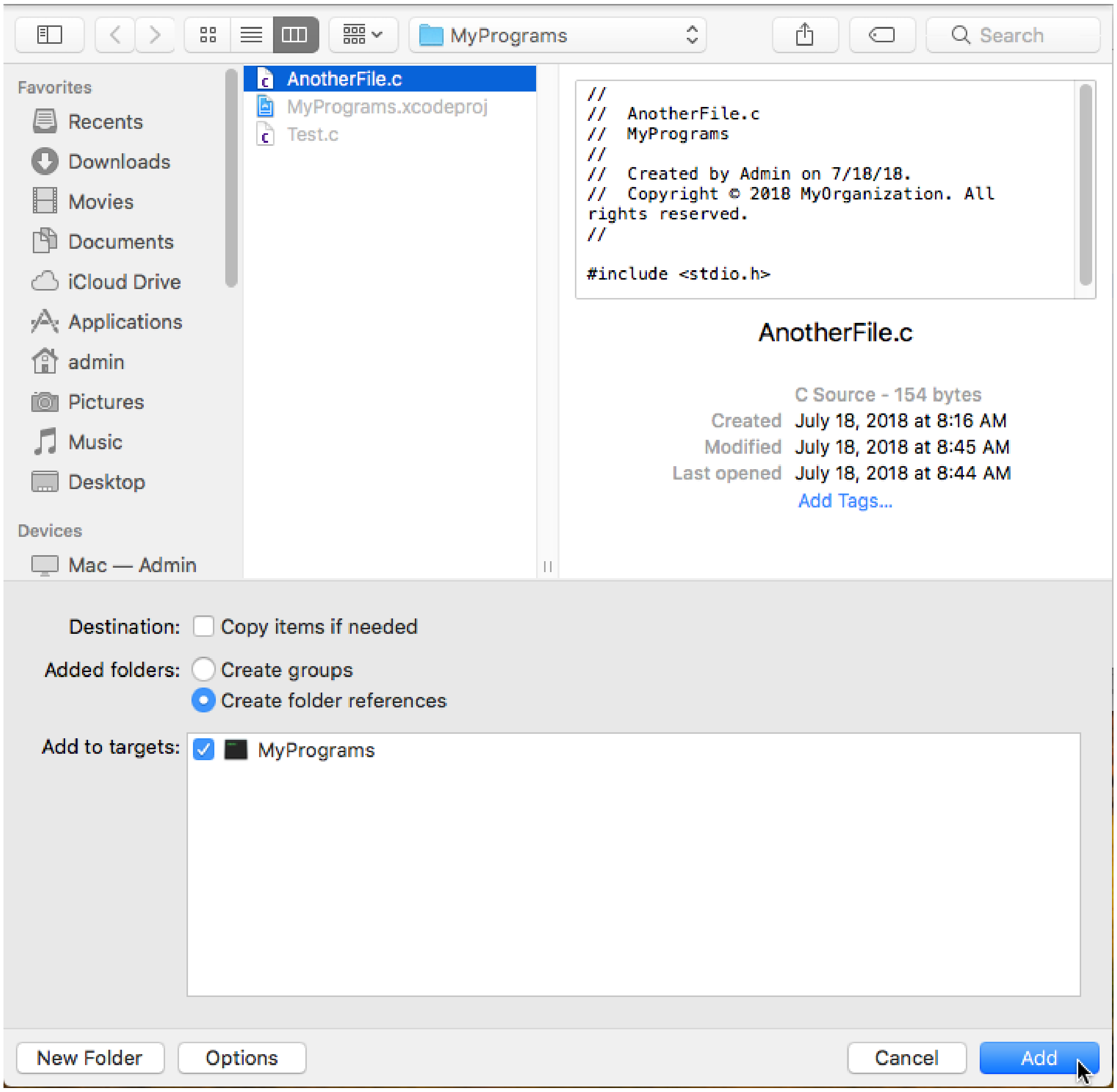


Figure 31

Continued on the next page...

2. In this example there is a file named "AnotherFile.c" that already exists from some earlier testing and I want to add it. To add one or more files merely select them and click ***Add***. This closes the window and the added files will be shown in the "Project Navigator" frame of the project window, as shown in Figure 33 on the next page.



```
//
//  AnotherFile.c
//  MyPrograms
//
//  Created by Admin on 7/18/18.
//  Copyright © 2018 MyOrganization. All
rights reserved.
//

#include <stdio.h>
```

AnotherFile.c

C Source - 154 bytes

Created	July 18, 2018 at 8:16 AM
Modified	July 18, 2018 at 8:45 AM
Last opened	July 18, 2018 at 8:44 AM

Destination: □ Copy items if needed

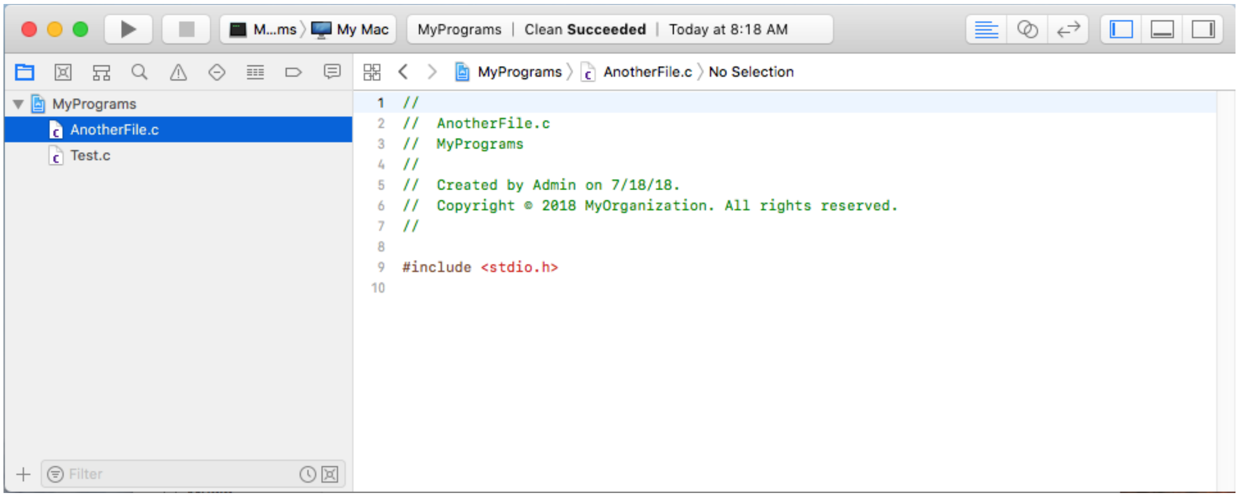
Added folders: ○ Create groups
● Create folder references

Add to targets: ☑ MyPrograms

Figure 32

Adding Source Code Files to a Project, continued

3. Figure 33 below illustrates how the project window looks after file “AnotherFile.c” has been added to the project named “MyPrograms” and selected. You may add whatever code you deem appropriate to these files at any time if you haven’t already done so.



```
//
//  AnotherFile.c
//  MyPrograms
//
//  Created by Admin on 7/18/18.
//  Copyright © 2018 MyOrganization. All rights reserved.
//

#include <stdio.h>

```

Figure 33

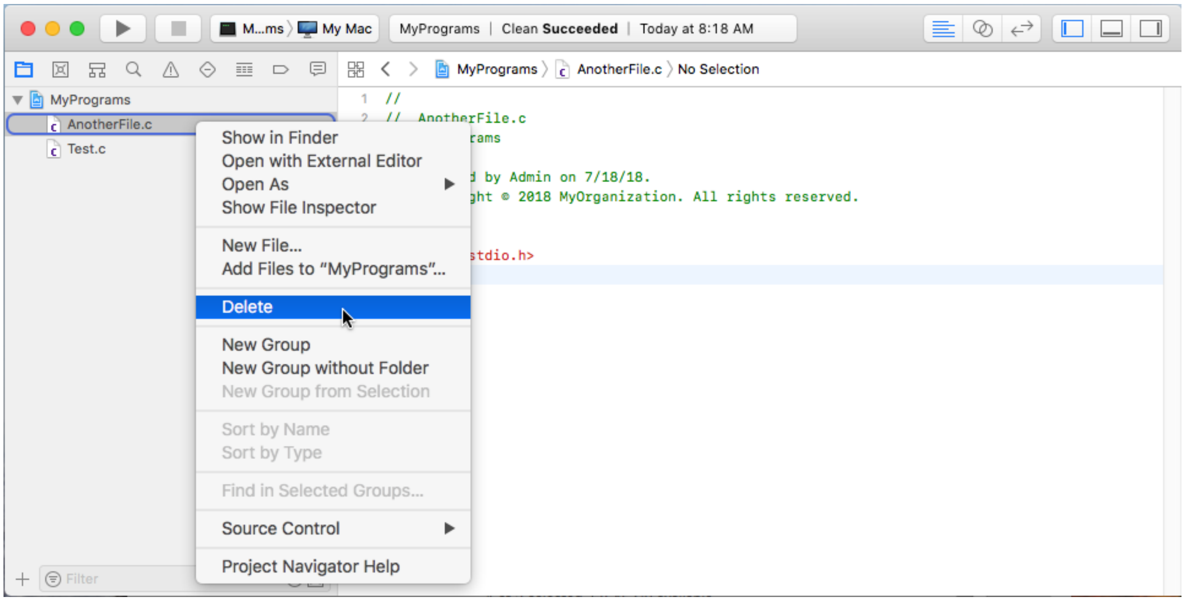
Using the Same Project for Every Exercise

For simplicity I recommend that you use the same project for all course exercises rather than creating a separate project for each. To do this you need simply exclude the file(s) for a previous exercise before adding those for the next. Excluded files will not be removed from the machine but will instead remain in your project folder in case you want to add them back into your project later or simply view their contents. See the next section on “Excluding Source Code Files from a Project” for information on how to do this very simply.

Excluding Source Code Files from a Project

Use this procedure to exclude files from a project or from the list of files (if any) at the top of the "Project Navigator" frame. Excluded files will remain in the project folder in case you want to add them back into your project later or simply view their contents.

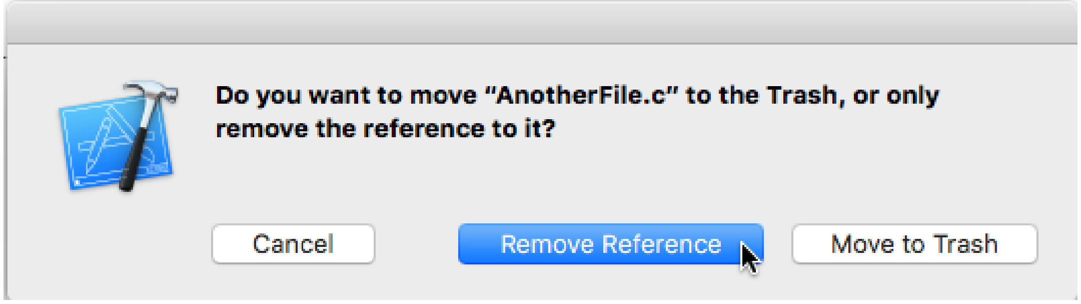
1. To only remove one file right-click it in the "Project Navigator" frame under the project name; to remove multiple files select all of them first, then right-click anywhere in the selection. Either way, select **Delete** from resulting context menu, as shown below in Figure 34. This opens the window in Figure 35.



```
1  //
2  //  AnotherFile.c
         rams

         d by Admin on 7/18/18.
         ght © 2018 MyOrganization. All rights reserved.

         stdio.h>
```

Figure 34

2. If you simply wish to exclude the selected file(s) from your project but keep them for possible future use (recommended), click ***Remove Reference***. However, if you wish to remove them from your computer entirely, click ***Move to Trash***. Either way the file(s) will be removed from the project.



Do you want to move "AnotherFile.c" to the Trash, or only remove the reference to it?

Figure 35

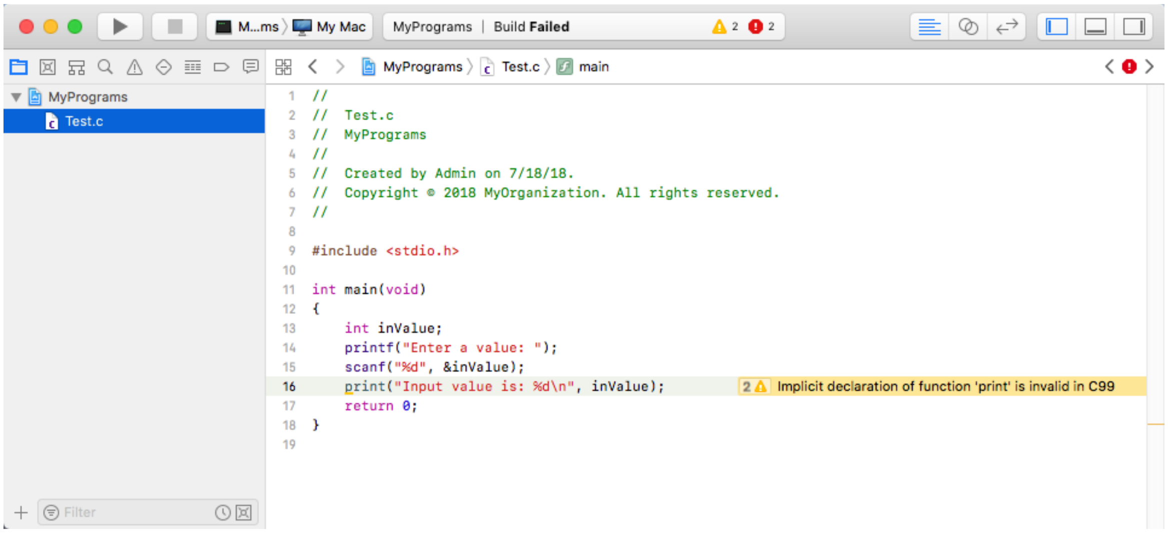
Compiling and Running a Program Using the IDE

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.

To Compile the Code and/or Build the Executable:

1. To compile any source code file(s) that have been modified since the previous build and create an executable program from them, select **Product → Build** from the Xcode menu (Figure 7).

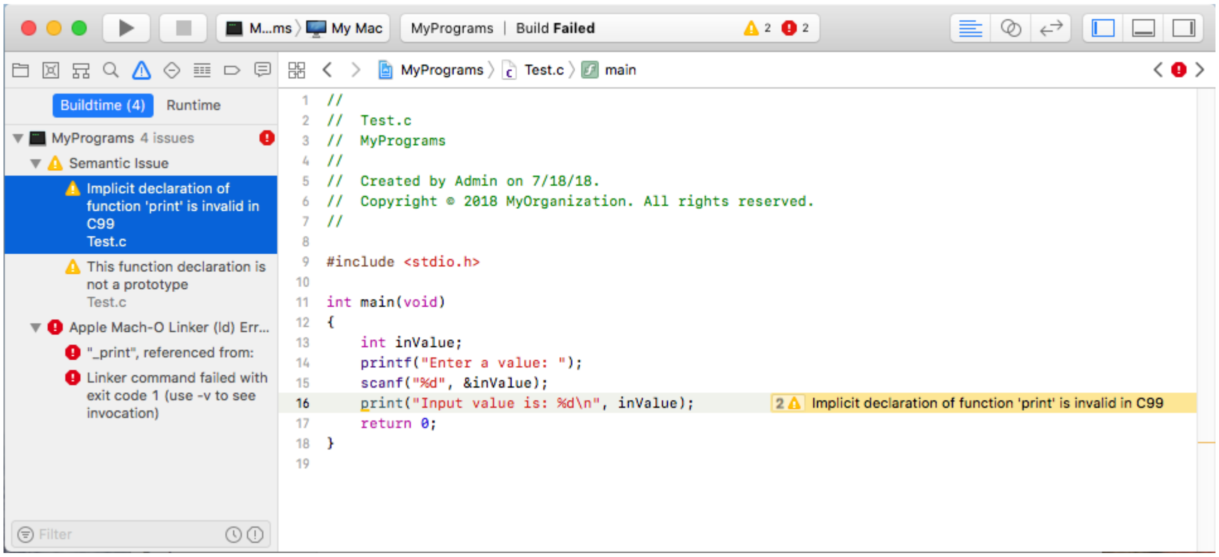
2. If there are any errors or warnings, as there are in the sample code in Figure 36 below, fix all of them. Although some error/warning messages might be displayed directly in the code as shown, others may not. To view all of them select **View → Navigators → Show Issue Navigator** from the Xcode menu (Figure 7), which displays the build results as shown in Figure 37.



```
//
//  Test.c
//  MyPrograms
//
//  Created by Admin on 7/18/18.
//  Copyright © 2018 MyOrganization. All rights reserved.
//

#include <stdio.h>

int main(void)
{
    int inValue;
    printf("Enter a value: ");
    scanf("%d", &inValue);
    print("Input value is: %d\n", inValue);     2 ⚠ Implicit declaration of function 'print' is invalid in C99
    return 0;
}
```

Figure 36

3. You may select whichever issue you would like to view in more detail.



MyPrograms 4 issues ❗
- Semantic Issue
  - ⚠ Implicit declaration of function 'print' is invalid in C99 Test.c
  - ⚠ This function declaration is not a prototype Test.c
- ❗ Apple Mach-O Linker (ld) Err...
  - ❗ "_print", referenced from:
  - ❗ Linker command failed with exit code 1 (use -v to see invocation)

```
//
//  Test.c
//  MyPrograms
//
//  Created by Admin on 7/18/18.
//  Copyright © 2018 MyOrganization. All rights reserved.
//

#include <stdio.h>

int main(void)
{
    int inValue;
    printf("Enter a value: ");
    scanf("%d", &inValue);
    print("Input value is: %d\n", inValue);     2 ⚠ Implicit declaration of function 'print' is invalid in C99
    return 0;
}
```

Figure 37

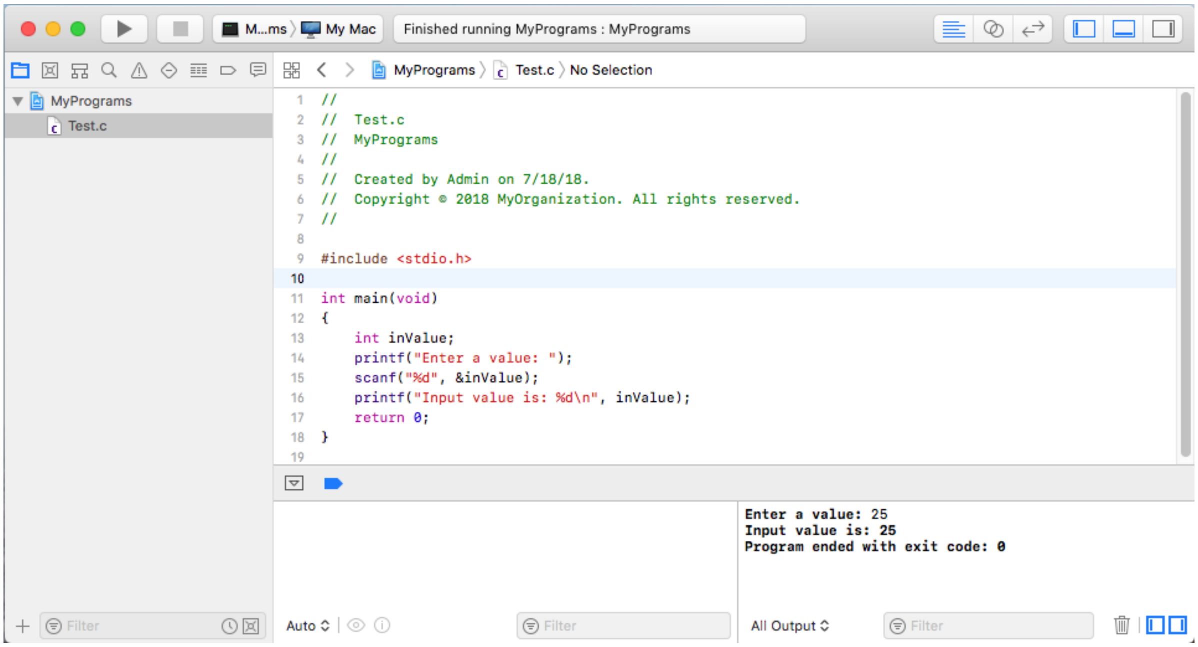
Continued on the next page...

Compiling and Running a Program Using the IDE, continued

To Run the Program:

After your code compiles/builds without errors or warnings you may run and test it.

1. To run your program select **Product → Run** from the Xcode menu (Figure 7). The results should be shown in the lower-right frame of the project window in Figure 38 below. If it isn't, select **View → Debug Area → Show Debug Area** from the Xcode menu (Figure 7). You may stretch it to whatever size you wish. If the program prompts the user for input he/she must enter the responses in that frame.



```
//
//  Test.c
//  MyPrograms
//
//  Created by Admin on 7/18/18.
//  Copyright © 2018 MyOrganization. All rights reserved.
//

#include <stdio.h>

int main(void)
{
    int inValue;
    printf("Enter a value: ");
    scanf("%d", &inValue);
    printf("Input value is: %d\n", inValue);
    return 0;
}

```

```
Enter a value: 25
Input value is: 25
Program ended with exit code: 0
```

Figure 38

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 39A and 39B show a portion of Xcode'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. Figure 40 shows icons that are located in the upper left of the project window. The arrowhead icon can be used for building then starting a program run while the square icon can be used to terminate it. Figure 41 shows some of the debugging icons, which are only visible when the program is running. These are located just below the source code window and are shown circled in red in Figure 44.

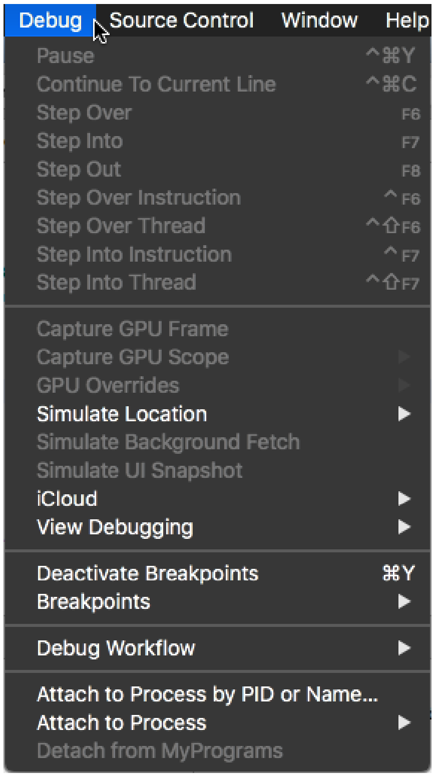


Figure 39A

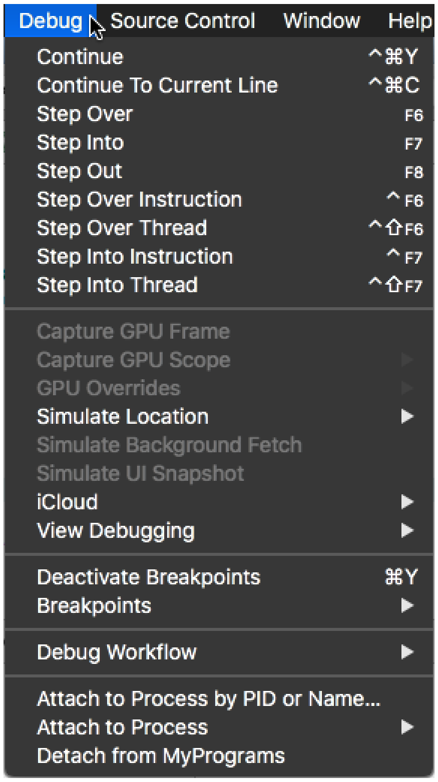


Figure 39B

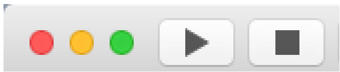


Figure 40

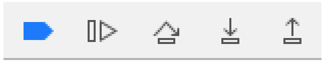
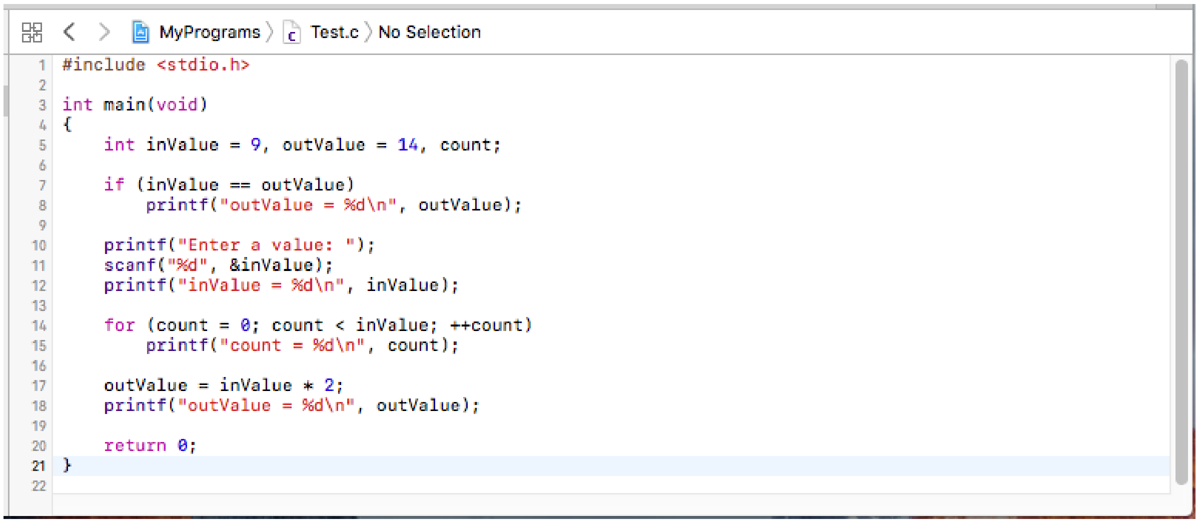


Figure 41

Using the IDE's Debugger, continued

The Program: Figure 42 below shows a typical program in an Xcode 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 and other expressions.

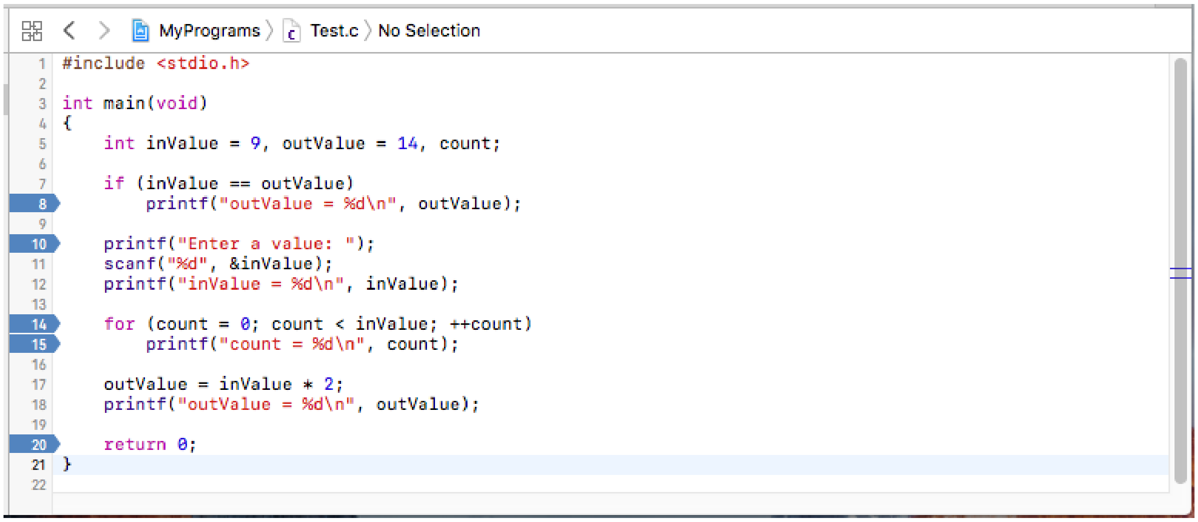


```c
#include <stdio.h>

int main(void)
{
    int inValue = 9, outValue = 14, count;

    if (inValue == outValue)
        printf("outValue = %d\n", outValue);

    printf("Enter a value: ");
    scanf("%d", &inValue);
    printf("inValue = %d\n", inValue);

    for (count = 0; count < inValue; ++count)
        printf("count = %d\n", count);

    outValue = inValue * 2;
    printf("outValue = %d\n", outValue);

    return 0;
}
```

Figure 42

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 margin just to the left of the line number, or you can place the cursor on that line and either press **Command+\** or select from the **Debug** menu. Five breakpoints have been set in Figure 43, as indicated by the blue arrows.

<u>IMPORTANT:</u> 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 35) to the next statement or set a breakpoint (page 33) on that next statement and continue (page 35).

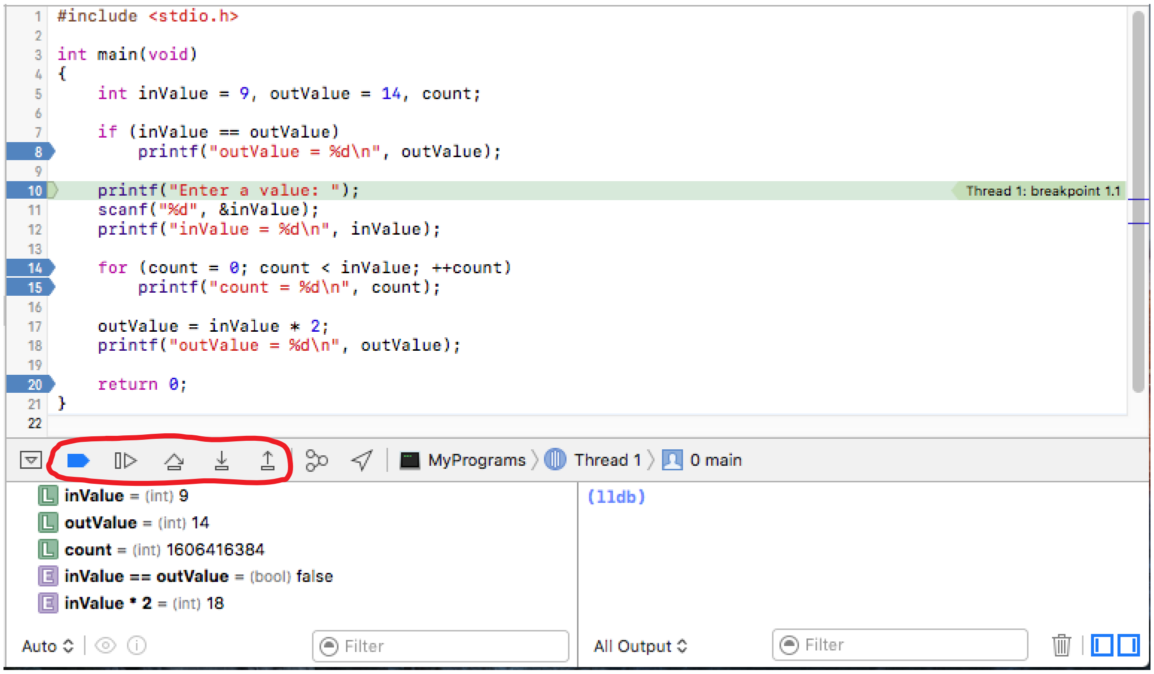


```c
#include <stdio.h>

int main(void)
{
    int inValue = 9, outValue = 14, count;

    if (inValue == outValue)
        printf("outValue = %d\n", outValue);

    printf("Enter a value: ");
    scanf("%d", &inValue);
    printf("inValue = %d\n", inValue);

    for (count = 0; count < inValue; ++count)
        printf("count = %d\n", count);

    outValue = inValue * 2;
    printf("outValue = %d\n", outValue);

    return 0;
}
```

Figure 43

Starting Debugging: An easy way to start the program is to press **Command+R**, but you may instead click the **Build and then run...** arrowhead icon in Figure 40 or select **Run** from the **Product** menu. A highlighted line indicates a debugging pause, at which point the values of variables can be examined.

Figure 44 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. You may delete them individually by right-clicking on them and selecting **Delete Breakpoint** from the resulting context menu. You can deactivate them individually by simply clicking on them, and deactivate all of them simultaneously when paused by pressing **Command+Y** or clicking on the blue arrow icon in the group of icons circled in red.



```
 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: ");                     Thread 1: breakpoint 1.1
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
```

```
L inValue = (int) 9
L outValue = (int) 14
L count = (int) 1606416384
E inValue == outValue = (bool) false
E inValue * 2 = (int) 18
```

```
(lldb)
```

Figure 44

Examining Variables and Other Expressions: Whenever a program is paused at a breakpoint or after a single-step you should see the values of all variables currently in scope displayed in the "Variables View" frame in the lower left of the window as shown in Figure 44. If not, press **Shift+Command+Y**. Other arbitrary expressions may also be placed there by right-clicking just below the last entry, selecting **Add Expression...** from the resulting context menu, and typing or pasting the desired expression into the resulting **Expression** entry box, which is what has been done in Figure 44.

<u>IMPORTANT:</u> 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 35) to the next statement or set a breakpoint (page 33) on that next statement and continue (page 35).

Note that in this example the values of variables **inValue** and **outValue** are 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 the values of listed items will be displayed during each pause, thereby allowing you to see how your code is affecting them. The value of a variable will also be displayed during a pause if you hover the mouse pointer over it in the code itself. The integer numeric values in this example are displayed in decimal, but by right-clicking on an item and selecting **View Value As** from the resulting context menu you may choose a different radix, such as octal or hexadecimal, or a different display format if appropriate.

Using the IDE’s Debugger, continued

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 **Command+.** or click the **Stop the running...** square icon in Figure 40 or select the **Stop** item in the **Product** menu.

The icons in Figure 45 below are from Figure 44 on the previous page and are repeated here just for convenience.

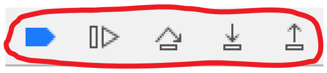


Figure 45

Continuing:

Press **Ctrl+Command+Y** or click the **Continue** icon (the 2nd icon in Figure 45) 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** – **F6** or the **Step Over** icon (the 3rd icon in Figure 45) 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** – **F7** or the **Step Into** icon (the 4th icon in Figure 45) 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** – **F8** or the **Step Out** icon (the 5th icon in Figure 45) 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:

7. Select **Product → Scheme → Edit Scheme...** from the Xcode menu (Figure 7) as shown in Figure 46. This will open the scheme editing window shown in Figure 47 on the next page.

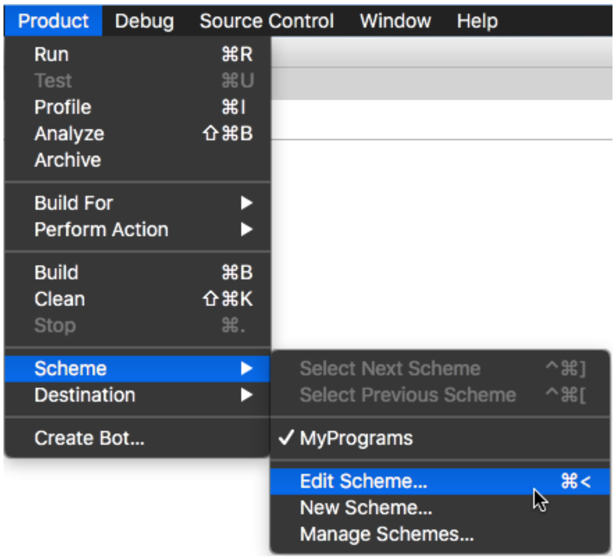


Figure 46

Continued on the next page...

Specifying Command Line Arguments from within the IDE, continued

8. As shown in Figure 47, select the ***Run Debug*** item in the left frame, the ***Arguments*** item at the top of the right frame, then expand the ***Arguments Passed On Launch*** item if necessary.

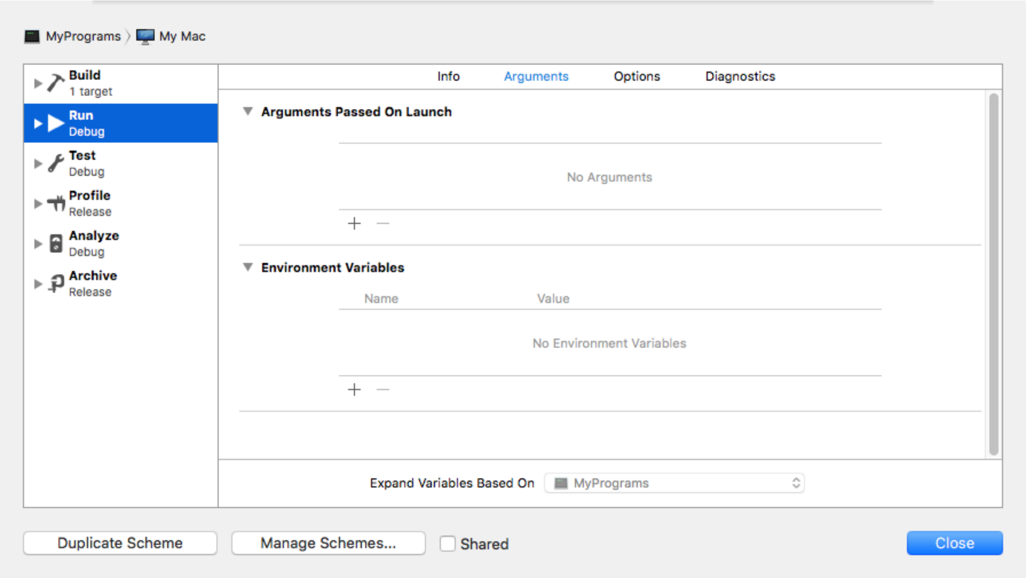


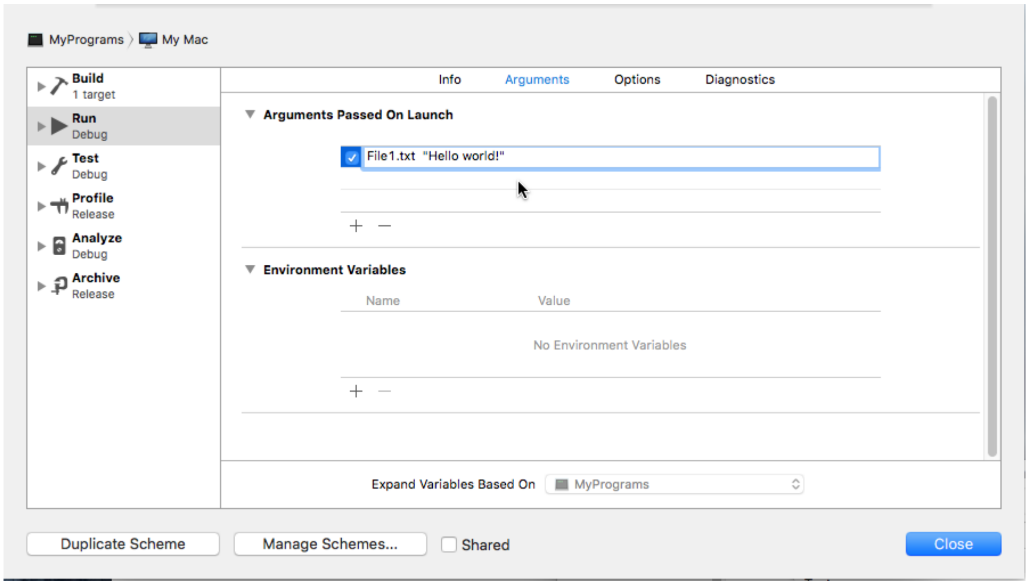
Figure 47

9. Click the plus sign in the ***Arguments Passed On Launch*** item to open the command arguments entry field, then enter the desired arguments into that field as shown in Figure 48.

   Note: For a string containing one or more spaces to serve as a single argument you must put double quotes around it. For example, ***"Mary Smith"*** instead of ***Mary Smith***

10. Click ***Close*** to close the window and accept the change, then run the program as usual.

BUG NOTE: 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, **should** cause all reads done by `scanf`, `getchar`, `cin >>`, `cin.get`, etc. to come from file ***InputFile.txt*** rather than the keyboard. However, I/O redirection does not appear to work properly from within the IDE. As an annoying workaround you can open an instance of "terminal" and run your program from there when I/O redirection is required.



☑ File1.txt "Hello world!"

Figure 48