

Chapter 5

Debugging with Xcode

Writing software is a difficult. Unless you're writing a very simple program, your code will have errors the compiler and static analyzer won't catch. Finding the errors by running your program can be difficult because the program runs too fast. It's like watching a DVD in fast forward mode.

A debugger lets you slow things down so you can see what is happening when your program runs. By using a debugger you can pause your program at any line of code, step through each line of code, and examine the values of your program's variables. A debugger helps immensely when you're trying to find out what's wrong with your code. For those of you new to programming, learning to use a debugger is an important skill to learn. In this chapter you'll learn how to use Xcode's debugger.

Before You Debug

If you create a new project, write the code for the project, and build the project, you should be able to debug your program with no problems. But there are steps you can take before you debug to make debugging go more smoothly.

- Configure Xcode for debugging.
- Choose a debugging format.
- Set your program's debugging options.
- Set environment variables for debugging.
- Use the debug versions of frameworks.
- Use the Guard Malloc library.
- Configure remote debugging, where you run your program on one computer and debug it on a second computer.

Configuring Xcode for Debugging

When configuring Xcode for debugging, your first task is to make sure the active build configuration is Debug, not Release. When you create a project in Xcode, the active build configuration should be set to Debug. If it's not, choose Project > Set Active Build Configuration > Debug.

As long as you're using the Debug build configuration, you should have no problems debugging. But to have the best debugging experience, examine the following build settings:

- Make sure the Generate Debug Symbols build setting is turned on. You need debug symbols to be able to debug your program.
- Set the Optimization Level build setting to None. Setting the optimization level to None ensures that every line of your code executes in the order you wrote it. During optimization the compiler may add, remove, or rearrange lines of code to make the program run faster, which makes debugging more difficult.
- Make sure the Fix and Continue build setting is turned on if you want to use Xcode's Fix and Continue feature. When you're debugging your program and you find an error in your source code, Fix and Continue lets you correct the error and continue debugging your program without leaving the debugger.

The Generate Debug Symbols, Optimization Level, and Fix and Continue build settings are in the Code Generation build settings collection. If you look in that collection, you'll see a Level of Debug Symbols build setting. You can ignore this setting. It works with the Stabs debugging format, which is no longer supported in Xcode.

Choosing a Debugging Format

If you look at the Debug Information Format build setting (look in the Build Options collection), you will see Xcode has three debugging format options for C, C++, and Objective-C programs: Stabs, DWARF, and DWARF with dSYM File. But Apple deprecated support for Stabs so you don't have much of a choice in debugging formats. You're going to be using DWARF. The decision to make is whether or not you want to create a dSYM file.

Should you decide to create a dSYM file, Xcode places all debugging symbols in a separate dSYM file. Using DWARF without a dSYM places the debugging symbols in the object files, which increases the size of the executable file. DWARF with dSYM is a good choice for release builds because you get a smaller executable file, but you have the debugging symbols available in case you release a product and users report problems. Plus, you don't have to worry about stripping debugging symbols out of the executable. For debug builds, the decision between DWARF and DWARF with dSYM is a matter of personal preference.

Setting Debugging Options for Your Program

Xcode has options to control how Xcode debugs your program. To set debugging options for your program's executable file:

1. Select Executables from the Groups and Files list.
2. Select the name of the executable file.
3. Click the Info button to open the inspector for the executable file.
4. Click the Debugging tab to show the debugging section of the inspector.

The debugging section is where you set the options related to debugging the executable file. At the top is a pop-up menu to choose the debugger to use. Xcode has three debuggers.

- `gdb` debugger. C, C++, and Objective-C programs should use the `gdb` debugger.
- Java debugger, which Java programs should use.
- AppleScript debugger, which AppleScript programs should use.

Xcode chooses the appropriate debugger based on the type of project you create so you shouldn't have to change debuggers. Below the When using pop-up menu is a group of options that apply to the debugger you're going to use.

Standard Input/Output

The first option you have is choosing the program to use for standard input and output. The options are the same for debugging as they are for running your program: pseudo terminal, system console, and pipe. In most cases pseudo terminal is the best choice. With pseudo terminal Xcode's debugger console works like a Unix shell window. From the console you enter `gdb` debugging commands and view the results of those commands.

If you want to save your program's output to a file, choose system console. The output goes to a log file instead of the debugger console. Do not use system console if you're going to enter commands in the debugger console or if your command-line program reads input from the user. Choosing system console prevents you from typing anything in the debugger console. If you're going to perform remote debugging, choose pipe for standard input and output.

Remote Debugging

If you're using `gdb` as your debugger, you can tell Xcode to debug the executable remotely. Remote debugging involves running your program on one computer and the debugger on another computer. Select the Debug execute remotely via SSH checkbox and enter the name of the computer you're connecting to in the Connect to text field. Refer to the section "Enabling Remote Debugging" later in this chapter for additional information about remote debugging.

Start Executable After Starting Debugger

The Start executable after starting debugger checkbox tells Xcode whether or not to start running your program after loading it in the debugger. If you deselect the checkbox, you must click the Restart button in the debugging window toolbar to run your program. The

advantage of selecting the checkbox is that your program starts running when Xcode loads it in the debugger, which is what you normally want to happen. The advantage of not selecting the checkbox is that waiting to run your program gives you time to set breakpoints.

Wait for Next Launch/Push Notification

Selecting the Wait for next launch/push notification checkbox tells the debugger to wait for the next launch/push notification. When you select this checkbox, the debugger will not attach to the binary when it launches. Waiting for the next launch/push notification helps debug push notifications on iPhone applications. Selecting this checkbox deselects the Start executable after starting debugger checkbox.

Break on Debugger() and DebugStr() Calls

The Break on Debugger() and DebugStr() checkbox tells Xcode whether or not to pause your program when it reaches a call to the `Debugger()` and `DebugStr()` functions in your code. The `Debugger()` and `DebugStr()` functions are part of the Carbon framework. The `Debugger()` function enters the kernel debugger, and the `DebugStr()` function prints text to the screen. If you don't call `Debugger()` or `DebugStr()`, the checkbox is irrelevant.

Auto-attach Debugger on Crash

Selecting the Auto-attach debugger on crash checkbox tells Xcode to launch the debugger and attach your program to it when your program crashes. In most cases auto-attaching the debugger is good. When your program crashes you'd like to see where the crash occurred in your code.

You should not select this checkbox if you're debugging fullscreen programs, like games, on a Mac with one monitor. If your program crashes and the debugger attaches, a major problem occurs. In a fullscreen program the screen is captured. With the screen captured, the only keyboard sequence that works is the Force Quit sequence (Cmd-Option-Esc). But the program is paused in the debugger, which prevents force quitting. Your Mac is effectively frozen, forcing you to restart it.

Adding Places to Look for Files

Finally, you can tell Xcode additional places to find source code files. To add a directory, click the + button and type the path name to the source code files. If you have your source code files in your project folder, you won't have to add places for Xcode to look. If you split your source code files into multiple folders and spread the folders all over your hard drive, you'll probably need to tell Xcode where to find the files.

Setting Environment Variables for Debugging

Mac OS X's malloc library has a collection of environment variables to help debug memory allocations. Table 5.1 lists the variables. To set environment variables in Xcode:

1. Select Executables from the Groups and Files list.
2. Select the name of the executable file.
3. Click the Info button to open the executable settings inspector.
4. Click the Arguments tab.
5. To add an environment variable, click the + button in the section Variables to be set in the environment.
6. Give the environment variable a name and a value. Giving an environment variable a value of zero disables the variable.

Table 5.1 Malloc Library Environment Variables

Variable Name	Value	Description
<code>MallocStackLogging</code>	Any integer > 0	Logs the chain of functions your program called to make the memory allocation.
<code>MallocStackLoggingNoCompact</code>	Any integer > 0	Does what <code>MallocStackLogging</code> does and remembers call stacks of memory allocations that no longer exist. Set this variable to remember every memory allocation made at a certain memory address.
<code>MallocScribble</code>	Any integer > 0	When your program frees memory, the operating system fills the memory with garbage values so your program can't accidentally access the memory again.
<code>MallocGuardEdges</code>	Any integer > 0	For large (over 4096 bytes) buffers the operating system places guard buffers on each edge of the buffer. If your program tries to read or write past the buffer, the program will crash.
<code>MallocDoNotProtectPrelude</code>	Any integer > 0	Works with <code>MallocGuardEdges</code> . The operating system won't place a guard buffer in front of the buffer.
<code>MallocDoNotProtectPostlude</code>	Any integer > 0	Works with <code>MallocGuardEdges</code> . The operating system won't place a guard buffer behind the buffer.
<code>MallocCheckHeapStart</code>	The number of allocations before checking the heap.	Checks the heap for corruption after <code>x</code> memory allocations, where <code>x</code> is the value you supply to <code>MallocCheckHeapStart</code> .

<code>MallocCheckHeapEach</code>	The interval to check the heap after the initial heap check.	Works with the <code>MallocCheckHeapStart</code> variable. After making the initial heap check with <code>MallocCheckHeapStart</code> , the operating system checks the heap every <code>x</code> memory allocations, where <code>x</code> is the value you supply to <code>MallocCheckHeapEach</code> .
<code>MallocCheckHeapSleep</code>	The number of seconds to sleep.	Your program goes to sleep when a heap corruption occurs.
<code>MallocCheckHeapAbort</code>	Any integer > 0	Aborts your program when a heap corruption occurs.
<code>MallocErrorAbort</code>	Any integer > 0	Aborts your program when it illegally frees memory or when an error occurs in a call to <code>malloc()</code> or <code>free()</code> .
<code>MallocCorruptionAbort</code>	Any integer > 0	Works similarly to <code>MallocErrorAbort</code> , but <code>MallocCorruptionAbort</code> does not abort when an out of memory condition occurs.
<code>MallocLogFile</code>	Path to file.	Error messages are written to a file instead of the console.

Using the Debug Version of Frameworks

Apple provides three versions of its frameworks: standard, debug and profile. The debug version writes debugging information to Xcode's debugger console as your program runs. You can debug your program with all three versions of Apple's frameworks. The debug version provides more debugging information. To switch to the debug versions of Apple's frameworks:

1. Select Executables from the Groups and Files list.
2. Select the name of the executable file.
3. Click the Info button to open the executable inspector.
4. Click the General tab.
5. Choose debug from the Use suffix when loading frameworks pop-up menu.

After switching to the debug version of Apple's frameworks, clean and rebuild your project for the switch to take effect.

Guard Malloc

Memory errors in your code are difficult to debug because the problems seem to occur randomly. Your program may run well one time and crash the next time you run it. The Guard Malloc library eliminates the randomness. When your program commits a memory access error, Guard Malloc crashes the program, which helps you locate the source of the error.

The downside of using Guard Malloc is that your program runs up to 100 times slower when running with Guard Malloc. Because of the slowdown, I recommend not using Guard Malloc the first time you debug your program. Correct the errors you find without Guard Malloc before debugging with Guard Malloc. To turn on Guard Malloc, choose Run > Enable Guard Malloc before you start debugging your program.

Enabling Remote Debugging

Remote debugging allows you to run your program on one computer while debugging it from a second computer. Fullscreen programs benefit the most from remote debugging. Without remote debugging, fullscreen programs are difficult to debug because there's no way to see the debugger's window when the program is running.

Setting up remote debugging requires a surprising amount of effort. I am going to assume you're going to be debugging on your Mac. I will refer to the computer you're going to run your program on as the host computer. You must perform the following tasks to set up remote debugging:

- The host computer must allow remote login so your Mac can connect to it.
- You must generate a key identifying your Mac and send the key to the host computer.
- You must configure your project's build folder so your Mac and the host computer can access the files in the folder.
- You must set your executable file to allow remote debugging.

Allowing Remote Login

If you're going to debug a program running on another computer, the host computer must be configured to allow you to login to it remotely. Go to the computer you want to login to and perform the following steps:

1. Choose Apple > System Preferences.
2. Choose Sharing from the System Preferences panel.
3. Select the Remote Login checkbox to activate remote login.

After turning on remote login, you have the option of allowing all users to login remotely or restricting remote login to certain users. Initially remote login is set so all users can login remotely. To restrict remote login, click the Only these users radio button in Allow access for group. Click the + button to add a user. A sheet will open with a list of connected users for you to choose from.

Generating ssh Keys

Xcode's remote debugging uses secure shell (**ssh**) to communicate between the two computers. You must generate a **ssh** key identifying your Mac so your Mac can communicate with the host computer.

Use the **ssh-keygen** command to generate **ssh** keys. Use the following command to generate the keys:

```
ssh-keygen -t dsa
```

When you run the **ssh-keygen** command, **ssh** asks you for a file name to save the keys. **ssh** provides a default file in parentheses with the path `/Users/Username/.ssh/Filename`. Press the Return key to use the default file. Next, **ssh** asks you for a password. Be careful typing your password. The shell window provides no feedback when typing the password so it looks like you're not typing anything, even though you are typing the password.

After providing a file location and password, **ssh** generates two keys. The public key is named `id_dsa.pub`, and the private key is named `id_dsa`. Keep the private key on your computer and send the public key to the host computer.

The public key you generated goes into the file `authorized_keys` inside the `.ssh` directory on the host computer. The `authorized_keys` file contains the keys of all the people authorized to access the host computer. Use the `scp` command to copy the public key to the host computer. The `scp` command takes the following form:

```
scp Filename Username@MachineName
```

The machine name takes the form `Name.local`. If I want to copy my key to the network server named `Server.local`, I would use the following command to copy my key:

```
scp id_dsa.pub mark@Server.local
```

After copying the public key file to the host computer, you must add the key to the `authorized_keys` file. The `cat` command appends a file to the contents of another file. Use the `cat` command to append the public key file, `id_dsa.pub`, to the list of keys in `authorized_keys`. After appending the public key you can remove the copy of `id_dsa.pub` on the host computer by using the `rm` command, as you can see in the following example:

```
cat id_dsa.pub > ~/.ssh/authorized_keys  
rm id_dsa.pub
```

Creating a Shared Folder for the Project's Build Products

For remote debugging to work, both computers must be able to access the project's build folder. Either use a network directory to store the project or make the build folder a shared folder. To make a folder a shared folder:

1. Choose Apple > System Preferences.
2. Choose Sharing from the System Preferences panel.
3. Select the File Sharing checkbox to make the folder a shared folder.
4. Click the + button under Shared Users to add the build folder.
5. Use the Users section to restrict the users that can access the build folder.

After setting up the build folder to let both computers access them, you must tell Xcode where the build products and intermediate files are.

1. Choose Project > Edit Project Settings.
2. Go to the Build Locations collection.
3. Set the locations for the build settings Build Products Path and Intermediate Files Path to the shared folder.

Turning on Remote Debugging

The final step to enabling remote debugging is to turn it on for your program. To turn on remote debugging:

1. Select Executables from the Group and Files list.
2. Select the name of the executable file.
3. Click the Info button to open the inspector.
4. Click the Debugging tab.
5. Select the Debug executable remotely via SSH checkbox.
6. Enter the name of the computer on which you want to run your program in the Connect to text field.

Launching the Debugger

The setup work is done. Let the debugging begin. Xcode provides several ways to launch the debugger.

- Choose Run > Debugger to open the debugger window. Launch your program by clicking the Build and Debug button on the debugger window toolbar. If the button says Build and Run, click the Breakpoints button to change the text of the button to Build and Debug.
- Choose Build > Build and Debug to build your program, open the debugger window, and launch your program.
- If you have built your program, choose Run > Debug to open the debugger window and launch your program.
- If your program is currently running, you can attach the debugger to it by choosing Run > Attach to Process > Program Name.

You can tell Xcode to automatically open the debugger through Xcode's debugging preferences. Use the On Start pop-up menu to tell Xcode what to do when you start debugging.

Figure 5.1 shows the debugger window. The window has five sections.

- Toolbar, which has buttons for the most common debugging commands.
- Call stack viewer.
- Variable viewer.
- Editor, which is identical to the editor you use to enter your source code.
- Console, which displays gdb commands along with the input and output of command-line programs.

If you don't see the console, choose Run > Console. The console is a separate window if you're not using Xcode's All-In-One project window layout.

The debugger initially uses a horizontal layout. Choosing Run > Debugger Display > Vertical Layout changes the display to a vertical layout. In the vertical layout, the variable viewer is under the call stack viewer and the editor is next to the call stack and variable viewers. The vertical layout lets you see more lines of code in the editor.

Call Stack Viewer

The call stack viewer lets you view your program's call stack, which is the chain of function calls that led your program to its current location. Selecting a function from the list takes you to its location in the editor, assuming you wrote that function. Sometimes low-level system calls appear in the call stack; selecting them won't do anything.

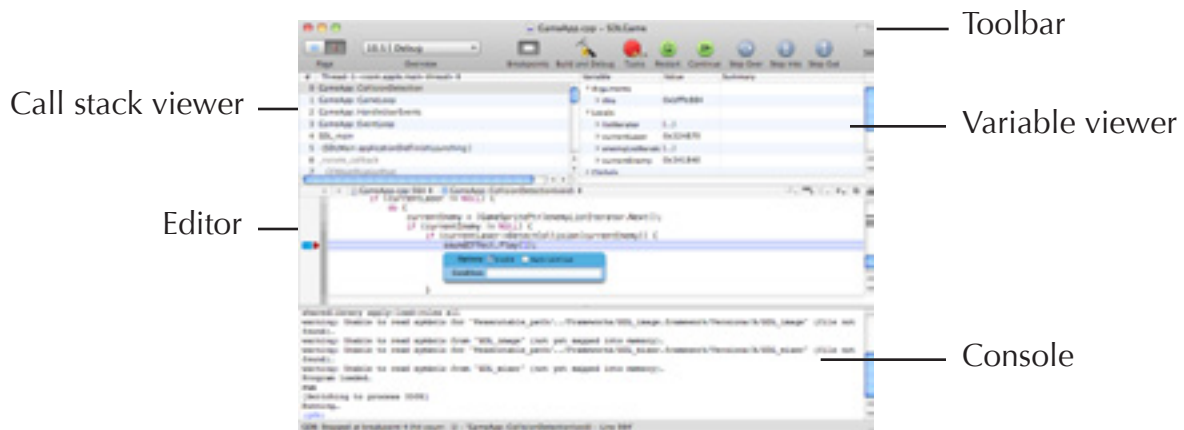


Figure 5.1

Debugger window

The call stack viewer has a pop-up menu that lets you select a thread to display in the viewer. Your code will have at least two threads, one for your program and one for the debugger. In this case Xcode displays the call stack for your program's main thread. If you write code with multiple threads, use the thread pop-up menu to examine your program's threads.

Variable Viewer

Use the variable viewer to look at your program's variables. It initially has three columns of information for each variable.

- The variable name.
- The variable's value.
- A summary, which provides additional information about the variable. If you have a variable of type `Rect`, the Summary column tells you the values of the rectangle's four corners. Many variables have an empty Summary column.

Sometimes a variable's summary has the message "out of scope". You will get this message when the debugger is in a section of code where the variable is inaccessible. Suppose you have the following code inside a function:

```
if (error) {
    NSString* errorMessage;
    // Do something with the error message
}
```

The scope of the `errorMessage` variable is the `if` block because the variable is declared inside the `if` block. Outside of the `if` block, `errorMessage` is inaccessible. The `errorMessage` variable will have the Summary value "out of scope" until the debugger is inside the `if` block.

Choosing Run > Variables View > Show Types adds a fourth column to the variable viewer. This column shows the variable's data type.

Xcode groups variables in the following categories:

- Arguments, which contains any arguments your program passed to the current function.
- Locals, which contains the function's local variables.
- File Statics, which contains any static variables in the source code file. Constants are a major source of static variables.
- Globals, which contains your program's global variables.
- Registers, which contains the contents of your Mac's registers. *Registers* are memory units in the CPU.

If a variable has multiple fields of information, such as C structs, C++ classes, Objective-C classes, or arrays, it will have a disclosure triangle next to it. Click the disclosure triangle to expand the variable to see the other fields. These fields may in turn have more fields; you can end up doing a lot of disclosure triangle clicking.

When you're debugging you normally have a set of variables you're most interested in viewing. You can place these variables in a separate window so you don't have to click as many disclosure triangles. Select the variables you want to view from the variable viewer and choose Run > Variables View > View Variable in Window.

Double-clicking a variable's value lets you edit the value by typing in a new one. Suppose you want to test your program's handling of null pointers. By setting a pointer variable to `NULL` in the variable viewer, you force your null pointer handling code to execute.

Setting Watchpoints

Watchpoints stop your program's execution when an expression changes value. Most of the time the expression you're interested in is a variable. To set a watchpoint, select the variable from the variable viewer and choose Run > Variables View > Watch Variable. The variable you set a watchpoint on will have a magnifying glass icon next to it to indicate that it is a watchpoint.

Custom Data Formatters

Custom data formatters let you customize what appears in the Value and Summary columns for each variable. Data formatters are especially useful for displaying the data structures you create for your programs. By using data formatters, you can display the most important data structure information in the Summary column. For data formatters to work, they must be enabled in Xcode. Choose Run > Variables View and make sure the Enable Data Formatters menu item has a check mark next to it.

To create a custom data formatter, double-click the Value or Summary column for a variable and enter a format string. Any literal text you enter will appear in the debugger window exactly as you type it, which makes the literal text good for labels. To refer to the variable itself, use the value `$VAR`. If the variable is a data structure, you can refer to the structure's individual members by placing the character `%` before and after the member name.

`%MemberName%`

If one of your data structure's members is another data structure, use the dot operator. Suppose you have a data structure with a member named `area`, which is of type `Rect`. You want to show the left edge in the format string. You would type the following format string:

```
%area.left%
```

Let's walk through a simple example of using data formatters. Suppose you have a data structure for 3D vectors named `Vector3D` with members `x`, `y`, and `z`. You would like the Summary column to show the `x`, `y`, and `z` components of the vector. Select the `Vector3D` variable in the debugger window and double-click the Summary column. Enter the following text in the Summary column:

```
x=%x%, y=%y%, z=%z%
```

Now every `Vector3D` variable in your program will show the `x`, `y`, and `z` components in the Summary column. If `x` has a value of 1, `y` has a value of 3, and `z` has a value of 5, the Summary column shows the following output:

```
x=1, y=3, z=5
```

Data formatter strings are stored in the following location:

```
/Users/YourUsername/Library/Application Support/Apple/
Developer Tools/CustomDataViews/CustomDataViews.plist
```

Viewing Shared Libraries

Choose `Run > Show > Shared Libraries` to open the Shared Libraries window, which you can see in Figure 5.2. At the top of the window are two pop-up menus that let you specify the default level of debugging symbols to show for system and user libraries. Most of the shared libraries in your programs are going to be system libraries. Only libraries that your project's targets produce are user libraries.

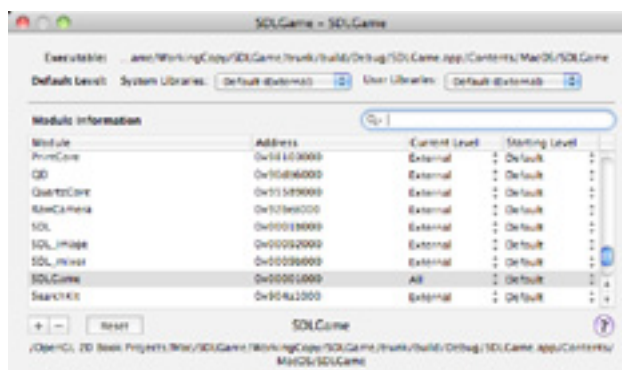


Figure 5.2

Shared libraries window

There are three levels of symbol showing. You can tell the debugger to show all symbols, no symbols, or external symbols, symbols declared external in the library's code. Showing external symbols is the default. The debugger loads all external symbols and loads other symbols when necessary.

Why wouldn't you want to show all debugging symbols? Showing debugging symbols is a balance between the ability to debug and speed of debugging. Mac OS X applications use a lot of shared libraries, and these libraries have lots of symbols. Loading every one of these symbols takes time and make debugging slower. Loading external symbols balances your need for information with your need for speed.

Below the pop-up menus is a list of libraries. The Shared Libraries window displays the following information for each library:

- The name of the library.
- The library's memory address. If there is no memory address, the library has not been loaded.
- The current level of debugging symbols to display for the library.
- The starting level of debugging symbols to display for the library.

Each library has a menu to set the current and starting level of debugging symbols. If you have some libraries where you want to see more debugging symbols, you can set the starting level for those libraries. Setting the current level helps if you want to see more debugging symbols at certain times when you're debugging.

Clicking the Reset button sets the starting level of debugging symbols back to the default value you set in the pop-up menus. Select a library and click the minus button to remove a library from the Shared Libraries window. Click the + button to add a library to the window.

Viewing Global Variables

Xcode does not automatically display the contents of global variables, which can be a problem if you use global variables in your programs. You must tell Xcode the global variables it should show in the variable viewer. Choose Run > Show > Global Variables to open the global variables browser, which you can see in Figure 5.3. Clicking the disclosure triangle next to Globals in the variable viewer also opens the global variables browser.

The left side of the global variables browser contains all the libraries in your program. Selecting a library from the list fills the right side of the window with the library's global variables. Select the checkbox next to a global variable to tell Xcode to display it in the variable viewer.

Tracking Expressions

Xcode's expressions window lets you view and track the value of expressions. An expression is a piece of code that returns a value. The most common expressions are variables and the results of functions and methods.

If you read the section on the variable viewer, you remember that you can view variables in a separate window. Why would you use the expressions window to view variables? The expressions window lets you add variables when you want and have all the variables in one window. To show a group of variables in a separate variable window, you must add the variables at the same time. If you add another variable later, Xcode creates a new window for that variable. By using the expressions window, all the variables will be in one window.

To track a variable, select it from the variable viewer and choose Run > Variables View > View Variable As Expression. To track other expressions, open the expressions window by choosing Run > Show > Expressions. Type the expression in the Expression text field. If you're tracking a function, cast the function to its return type.

`(ReturnType)FunctionName`

The expressions you enter won't appear in the expressions window until your program reaches a breakpoint. Make sure you type the expression correctly. If you misspell the expression or enter a non-existent variable name, Xcode won't tell you about the error, except to provide the Summary value of "out of scope" in the expressions window.

To stop tracking an expression, select it from the expressions window and press the Delete key.

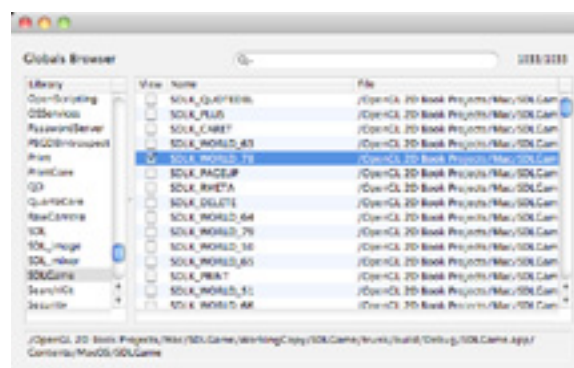


Figure 5.3

Global variables window

Viewing Dynamic Arrays

Many programs use pointers to create arrays when the size cannot be determined until the program runs. To create these dynamic arrays, you declare a pointer variable, then use the `malloc()` (in C) or `new()` (in C++) calls to make the pointer large enough to store the array. When you look at the pointer variable, you want to look at the array, not the variable. If you look at the pointer variable in the variable viewer of Xcode's debugging window, you'll see only the pointer, not the array. How do you view the array?

Open the expressions window and enter an expression in the Expressions text field. The expression should take the following form:

```
*ArrayName@Length
```

If you wanted to see the first 25 elements of the array `myArray`, you would enter the following expression in the Expressions text field:

```
*myArray@25
```

One thing I noticed when experimenting with dynamic arrays was the length I entered did not matter. As long I entered a length, the expressions window showed every element of the array. The benefit of this behavior is you don't have to know the size of the array to view the whole array.

In-Editor Debugging

Xcode lets you debug your program without leaving the editor. Debugging from the editor is easy. Open a file and start debugging. A debugger strip, shown in Figure 5.4, will appear in the editor underneath the toolbar. The debugger strip has three sections: a thread selector, navigation controls, and a call stack viewer. The navigation controls section contains seven buttons.

- A breakpoints button that activates and deactivates breakpoints.
- A pause/continue button that pauses or resumes running the program.
- A step over button.
- A step into button.



Figure 5.4

Debugger strip

- A step out button.
- A show debugger button. Click it to open the debugger window.
- A show console button. Click it to show the console.

If you don't see any debugging controls in the editor, go to Xcode's debugging preferences. Make sure the In-Editor Debugger Controls checkbox is selected.

Mini Debugger

The mini debugger is a HUD (heads up display) window that floats over all other windows. It lets you debug a program without having to bring the Xcode debugger window to the front.

You must start debugging your program before you can open the mini debugger window. Choose Run > Mini Debugger to open the mini debugger window.

If you want to use the mini debugger all the time, open Xcode's preferences. In the debugging section, choose Show Mini Debugger from the On Start pop-up menu. Make sure the In Editor Debugging Controls checkbox is selected.

When you reach a breakpoint in your program, the mini debugger will contain an editor window for the source file where the breakpoint occurred. One change is you will see a debugging strip at the top of the window that lets you perform the most common debugging tasks. For more information on the debugging strip, read the "In-Editor Debugging" section earlier in this chapter.

Datatypes

If you read the sections on the mini debugger and in-editor debugging, you may be wondering how to view the values of variables without going to Xcode's debugger window. This is where datatips come in. *Datatips* provide debugging information for a variable in an Xcode editor.

Turning on Datatips

To turn on datatips, choose Run > Debugger Display > Datatips. If you want to be able to step through your code in the content area of the editor, choose Run > Debugger Display > Step Controls.

Using Datatips

Moving your mouse over a variable in the editor displays basic information about the variable: its name, data type, value, and any summary information. Figure 5.5 shows an example of a datatip. If the variable has additional information, there will be a disclosure triangle on the left side. Move the mouse over the disclosure triangle to see the additional information. Classes, data structures, and arrays are the variables most likely to have additional information.

When the information on a variable is visible in the editor and you move the mouse cursor over the variable, a pop-up button cell (it looks like a tiny set of up/down arrows) appears next to the variable. Click the cell to open a menu. Use the menu to print a description of the variable or to view the variable in the memory browser.

Clicking a variable's value lets you change its value.

Using Step Controls

Moving the mouse in the gutter next to a line of code brings up a step control. If you're at the current line of code, the step control is a Step Over button. For other lines of code, the step control is a Continue button that tells the debugger to continue running until it reaches the line of code you clicked the Continue button on.

Breakpoints

To get any real debugging done, you need to use breakpoints. Technically, you can click the Pause button in the debugger window then step through your code. But Mac OS X programs with a GUI spend most of their time waiting for user input. When waiting for input your program isn't doing anything so there's nothing to debug.

Breakpoints tell the debugger to pause execution of your program at a specific line of code in your program. From there you can step through the code to find out what's wrong. By using breakpoints you can focus on a small section of code.

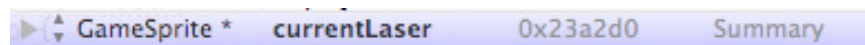


Figure 5.5

Datatip

Setting Breakpoints

The easiest way to set a breakpoint is to open one of your source code files and click the gutter next to the line of code where you want your program to pause. The gutter will have a blue arrow next to it. Clicking a breakpoint arrow in the gutter disables the breakpoint. Click the disabled breakpoint arrow to enable the breakpoint. Drag the arrow out of the gutter to delete a breakpoint.

If you want to see a list of all the breakpoints you've set, open the breakpoints window by selecting Breakpoints from the Groups and Files list or choose Run > Show > Breakpoints. Figure 5.6 shows what the breakpoints window looks like. There are seven pieces of information for each breakpoint.

- An icon identifying the type of breakpoint: line, symbolic (pause when a function is reached), or C++ exception.
- The location of the breakpoint: filename and line number.
- A checkbox that tells you if the breakpoint is enabled.
- A condition that tells Xcode when to pause your program. A blank condition tells Xcode to pause your program every time you reach the breakpoint.
- An ignore count that tells Xcode to ignore the breakpoint the number of times that you specify.
- An auto-continue checkbox that tells Xcode to continue execution after reaching the breakpoint.

The auto-continue checkbox requires some additional explanation. Deselecting the checkbox is the equivalent of pressing the Pause button on a DVD player. It pauses your program. If you want to step through your code after reaching the breakpoint, make sure the auto-continue checkbox is not selected.

Selecting the auto-continue checkbox is the equivalent of pressing the Pause button on a DVD player followed immediately by pressing the Play button. When should you select the auto-continue button? You should select the auto-continue button if you attach a breakpoint action to the breakpoint. When your program reaches the breakpoint, Xcode performs the action and goes back to running your program. Refer to the next section for information on breakpoint actions.

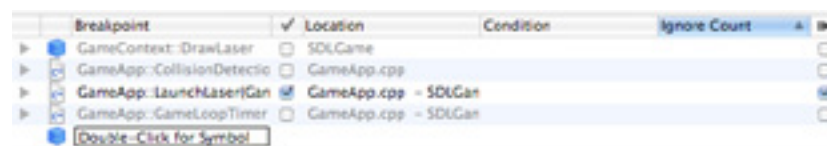


Figure 5.6

Breakpoints window

If Xcode isn't hitting your breakpoints when running your code in the debugger, make sure you're using the Debug build configuration. If you're using the Debug configuration and still can't hit breakpoints, open Xcode's debugging preferences. Deselect the Load Symbols Lazily checkbox. Loading symbols lazily tells Xcode to load debugging symbols when they're needed. Doing so improves debugging performance, but loading symbols lazily can cause breakpoints to not be hit. Turning off lazy symbol loading might solve the problem.

Breakpoint Actions

If you have the breakpoints window open, clicking the disclosure triangle next to a breakpoint lets you add an action to take when the breakpoint is reached. Refer to Figure 5.7 for a breakpoint action example. Click the + button. There are five breakpoint actions you can perform.

- Debugger commands execute `gdb` commands.
- Log writes a message to the console or speaks the message.
- Sound plays a system sound.
- Shell commands execute a Unix shell command. Use a shell command to run shell scripts when you reach a breakpoint.
- AppleScript executes an AppleScript script.

Debugger Commands

Enter the command in the text field. Selecting the Log checkbox tells Xcode to log the output of the command.

Log

Enter the message you want to log in the text view. If you want to log the value of a variable, wrap it in @ tags.



Figure 5.7

Breakpoint actions

@VariableName@

Select the Speak radio button to tell Xcode to speak the message instead of logging it.

Sound

Playing a sound helps when debugging a fullscreen program. You can play a sound when reaching an area of code. Use the pop-up menu to choose a sound.

Shell Commands

Shell commands can be very powerful. You can even run command-line programs using shell commands. You could check for memory leaks when you reach a breakpoint by running the `leaks` command and supplying the name of your program.

If you add a shell command breakpoint action, two text fields appear in the breakpoints window. Enter the command in the first text field. If you've written a shell script that you want to use, click the Choose button. Enter the arguments the shell command takes in the second text field.

If the auto-continue checkbox is selected for the breakpoint, Xcode will not wait for the shell command to execute before continuing to run your program. Select the Wait until done checkbox to tell Xcode to wait for the shell command to finish before resuming execution of your program.

AppleScript

Enter your script in the text field. Use the Compile and Test buttons to make sure your script works before using it as a breakpoint action.

Logging to a File

When you log the output of a debugging command or create a log action, the log data appears in the debugger console. How do you log the data to a file? The easiest thing to do is copy the output from the console and paste it into an empty text file.

Another way to log the data to a file is to turn on `gdb` file logging. Open Xcode's Debugging preferences. Select the checkbox underneath GDB Log. Now Xcode will log everything `gdb` does to a file when you debug. Logging everything `gdb` does usually results in too much information, which makes finding what you need difficult. The excessive information generated by the `gdb` log is why copying and pasting from the debugger console is easier.

To read the `gdb` log, open the Debugging preferences and Click the Open Log button. A window will open in the Finder. The log file has a filename similar to the following:

`Xcode###-AppName`

Where `###` is a number and `AppName` is the name of the program you're debugging. Xcode stores one log file for each program you're debugging. When you debug a program, Xcode overwrites any existing log files for that program. If you want to save the log file permanently, move it out of the folder.

When you open a log file from Xcode, you'll notice the log is in an obscure location. To change the location of the logs, enter a path in the GDB Log text file. All logs will be written to this file. The path you enter must be the path to a file. If you supply a path to a folder, no log file will be created.

Breakpoint Templates

Xcode ships with a set of breakpoint templates that perform common breakpoint actions. To use the breakpoint templates, right-click the line of code in the gutter to open a contextual menu. Choose Built-in breakpoints to get the list of breakpoint templates. If the line of code does not have a breakpoint, Xcode adds a breakpoint for you. There are six breakpoint templates.

- Log breakpoint and arguments, which logs the breakpoint number and the arguments to the current function.
- Log breakpoint and hit count, which logs the breakpoint number and the number of times the breakpoint has been hit.
- Log stack trace, which logs the call stack.
- Sound out, which plays a sound when you reach the breakpoint.
- Print self, which prints a description of an Objective-C object.
- Speak breakpoint and hit count.

All of the breakpoint templates auto-continue. They perform the action and resume executing your program.

The output of the log and print breakpoint actions appears in the debugger console. If you can't see the console, choose Run > Console.

Choosing another breakpoint template replaces the action from the old template with the new one. To completely turn off a breakpoint template for a particular breakpoint, open the breakpoints window, click the disclosure triangle next to the breakpoint, and click the minus button.

Stepping Through Your Code

When debugging you will spend a lot of time stepping through your code. Stepping means executing one line of code at a time, letting you pinpoint problems in your program.

The debugging toolbar has three buttons for stepping: Step Over, Step Into, and Step Out. To demonstrate stepping I will use a small C function.

```
void ExampleFunction(void)
{
1   int routinesCalled;

2   routinesCalled = 0;
3   FirstRoutine();
4   routinesCalled++;

5   SecondRoutine();
6   routinesCalled++;
}
```

Assume you've paused the program at `ExampleFunction()`. The debugger moves past line 1 and stops at line 2. For this line of code, the Step Over and Step Into buttons do the same thing. They execute the line of code, giving `routinesCalled` the value zero, and move to line 3.

If you click the Step Over button again, the debugger executes all the code in the function `FirstRoutine()` and moves to line 4. The debugger steps over the function call and moves to the next line of code. Clicking Step Over a second time moves the debugger to line 5.

Clicking the Step Into button at this point moves the debugger inside `SecondRoutine()`. From there you can step through the code in `SecondRoutine()`, which I have not bothered to list. The debugger steps into the function you're calling. After executing all the lines of code in `SecondRoutine()`, the debugger moves to line 6. If you don't want to walk through every line of code in `SecondRoutine()`, click the Step Out button when you're finished looking at `SecondRoutine()`. The debugger will step out of `SecondRoutine()` and take you back to `ExampleFunction()`.

On a line of code that calls another function, clicking Step Over executes the code in the function. Clicking Step Into takes you inside the called function. Clicking Step Out takes you out of the called function.

Viewing Memory

To look at the contents of memory, open the memory browser by choosing Run > Show > Memory Browsers. The memory browser, shown in Figure 5.8, has two combo boxes and two pop-up menus to control what appears in the memory browser and control how the memory appears. The Address combo box lets you choose the starting memory address to appear in the browser. You can enter an address in the text field, choose from a list of previously viewed addresses, or use the little arrows to choose the starting address. Selecting a variable from the debugger window and choosing Run > Variables View > View in Memory Browser displays the variable's memory contents in the memory browser.

The Bytes combo box determines the amount of memory the browser displays. The combo box has initial values ranging from 512 bytes to 64 KB. You can also type in the amount of bytes to show.

The arrows next to the Address combo step through memory. The amount to step is determined by the value of the Bytes combo box. If the Bytes combo box has the value of 512 bytes, the arrows move up or down 512 bytes from the current address in the browser.

The Word Size pop-up menu determines how many bytes of memory appear in one column of the browser. The possible word sizes are 1, 2, 4, and 8 bytes. The Columns pop-up menu determines how many columns appear in one row of the browser. The browser can display 1, 2, 4, 8, 16, or 32 columns.

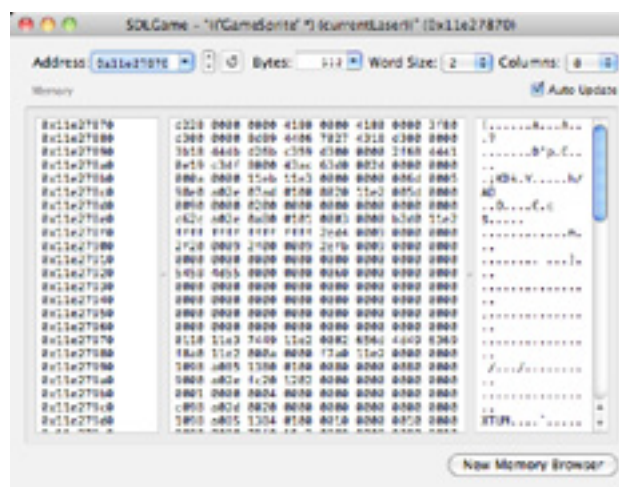


Figure 5.8

Memory browser

Each row in the memory browser displays the following information:

- The starting address for the row.
- The contents of memory, displayed as hexadecimal numbers.
- The ASCII character corresponding to the value of each byte of memory in the row.

Selecting the Auto Update checkbox in the memory browser tells the browser to update its contents when you step through code.

Viewing Disassembled Code

Initially the Xcode debugger is set to display only your source code, which is fine in most cases. If you need to view the disassembled code, choose Run > Debugger Display > Source and Disassembly. The editor pane will split, with the left pane showing your source code and the right pane showing the disassembled code, the corresponding assembly language statements.

Fixing Your Code While Debugging

The purpose of debugging is to find the source of your program's errors so you can correct them. With Xcode you can correct errors without leaving the debugger by using Fix and Continue.

NOTE

Fix and Continue is not being maintained by Apple. It works only for applications. If you're debugging a library or framework, you can't use Fix and Continue. Fix and Continue may not work for iPhone applications or Cocoa applications that use garbage collection.

Because Fix and Continue is not being maintained by Apple, there is no guarantee it will work properly for you. If Fix and Continue doesn't work for you, you're out of luck for now. However, Fix and Continue should work for minor code changes.

To use Fix and Continue, change your code in the editor and choose Run > Fix. A dialog box opens, asking if you want to save the changes you made to the file. Save them or else Fix and Continue won't work. Xcode compiles the file you modified and resumes execution where you last stopped it.

When you make a correction to your code, you want the debugger to stop where you made the correction so you can make sure your fix worked. The line of code you fixed may not be where the program counter, the current line of code, is. You can manually manipulate the program counter to make Xcode resume your program where you want after fixing. The program counter has a red arrow in the gutter, and Xcode highlights the line of code in red. You may have a hard time seeing the arrow if you set a breakpoint at that line of code. Drag the arrow to the line of code where you want Xcode to stop after compiling your corrections. This line of code will now be highlighted in red. Choose Run > Fix, and Xcode stops where you specified after recompiling the program.

Fix and Continue works on only one file at a time. If you find errors in two source code files, you must correct the errors in one file, click Fix, correct the errors in the second file, and click Fix again. Fix and Continue working on one file at a time is not as big a limitation as you might think. You should fix one error at a time, and one error rarely spreads across multiple files.

Using the gdb Console

The Xcode debugger handles basic debugging tasks; you can step through your code line by line, examine the values of variables, and set breakpoints. Many of you will be able to do all your debugging from the Xcode GUI. But if you want to do anything more advanced, you must use the `gdb` console and type commands in it. Xcode uses the `gdb` debugger for C, C++, and Objective-C programs.

NOTE

You cannot use the `gdb` console for Java and AppleScript programs because neither language uses `gdb` for debugging. Java programs use the Java debugger and AppleScript programs use the AppleScript debugger.

Using the console doesn't mean you have to abandon the Xcode GUI. You can use the debugger window toolbar to step through your code and view your program's variables in the debugger window while using the console to perform tasks the Xcode GUI can't handle.

Displaying the console is easy in Xcode. Choose Run > Console or drag the splitter bar at the bottom of the debugger window. Before you can type commands in the console, you must pause your program's execution in the debugger. If you've set breakpoints in your program, you can wait for your program to reach one of them. Otherwise you can click the Pause button on the debugger window toolbar to start entering commands in the `gdb` console.

Stopping Program Execution

There are three ways to stop your program's execution without explicitly pausing your program: breakpoints, watchpoints, and catchpoints. As I mentioned earlier in this chapter, breakpoints stop your program's execution at a specific line of code in your program. You can set breakpoints in the `gdb` console the way you would from the Xcode GUI, but `gdb` provides even more options. From the console you can set conditional breakpoints, which take effect only if it meets a condition you specify, and set breakpoints that execute only one time.

Watchpoints stop your program's execution when an expression changes value. Most of the time the expression you're interested in is a variable.

Catchpoints stop your program's execution when a C++ exception occurs. Exceptions provide a way of handling run-time errors in C++ programs. The C++ statements `try`, `catch`, and `throw` deal with exceptions. Obviously, only C++ programs can use catchpoints.

Setting Breakpoints

The `break` command, which you can abbreviate by typing `b`, sets breakpoints in the console. There are many ways to set them in `gdb`. If you supply no arguments to `break`,

```
break
```

`gdb` sets a breakpoint at the current line of code. Supplying a line number tells `gdb` to set a breakpoint at that line number in the current source code file. The following command:

```
break 447
```

Tells `gdb` to set a breakpoint at line 447 in the current file. To set a breakpoint at a specific line in a source code file, supply the file name, a colon, and the line number. The following command:

```
break main.m:15
```

Tells `gdb` to set a breakpoint at line 15 in the file `main.m`. Rather than setting a breakpoint at a line of code, you may prefer to set a breakpoint at the start of a function. Supply the function name when calling `break`. If you had a C function `PlayMusic()` that you wanted to set a breakpoint at, you would enter the following command:

```
break PlayMusic
```

Object-oriented languages like C++ and Objective-C complicate matters slightly. You must also supply a class name along with the function name because multiple classes can have a function with the same name. If you had a C++ class named `TMusic` with a member function `Play()`, you would enter the following command:

```
break TMusic::Play
```

If you had an Objective-C class named `TMusic` with a `play:` method, you would enter the following command:

```
break -[TMusic play:]
```

To set a breakpoint that stops your program only once, call `tbreak`. The `t` in `tbreak` stands for temporary. The `tbreak` command takes the same arguments as `break` does. The following command:

```
tbreak main.m:15
```

Sets a breakpoint at line 15 in the file `main.m` then deletes the breakpoint when your program stops at it.

Setting Watchpoints

The `watch` command sets watchpoints in your program. You supply an expression, which in most cases is a variable name. When the expression's value changes, `gdb` pauses your program.

```
watch x
```

Setting Catchpoints

The `catch` command sets catchpoints in your program. There are two ways to call `catch`.

```
catch catch  
catch throw
```

The first statement stops your program when it catches a C++ exception, and the second statement stops your program when it throws a C++ exception.

Examining Your Breakpoints

To view a list of all the breakpoints, watchpoints, and catchpoints, use the `info breakpoints` command. For each breakpoint, watchpoint, and catchpoint, `gdb` displays the information shown in Table 5.2.

Table 5.2 Information `info breakpoints` Provides

Information	Description
Breakpoint number	Breakpoints start at 1. Many breakpoint-related <code>gdb</code> commands take breakpoint numbers as parameters.
Type	Tells you whether you have a breakpoint, watchpoint, or catchpoint.
Disposition	Specifies what should happen when your program hits the breakpoint. The value <code>keep</code> tells <code>gdb</code> to keep the breakpoint. The value <code>dis</code> tells <code>gdb</code> to disable the breakpoint. The value <code>del</code> tells <code>gdb</code> to delete the breakpoint.
Enabled	Is the breakpoint enabled? If so, it will have the value <code>y</code> . If not, it will have the value <code>n</code> .
Address	The breakpoint's memory address.
What	The function name, source code file, and line number of the breakpoint.
Condition	If the breakpoint is conditional, the condition will reside here. If the breakpoint has been hit before, the number of times it's been hit will be here as well.

Setting Conditional Breakpoints

Conditional breakpoints give you the power to control when a breakpoint pauses your program. Supply a condition when you set the breakpoint. When your program hits the line of code where you set the conditional breakpoint, `gdb` pauses the program if the condition you supply is true.

There are two ways to set conditional breakpoints. First, supply a condition when you create the breakpoint with the `break if` command. The condition can be any Boolean expression, an expression that can be either true or false. Suppose you had the following code in a program:


```
enum DirectionType {
    NORTH, SOUTH, EAST, WEST
};

void Move(DirectionType directionToMove);
```

The following breakpoint executes when the program moves north:

```
break Move if directionToMove == NORTH
```

You can also use `if` to set conditional watchpoints, but you won't use conditional watchpoints as much as conditional breakpoints. Watchpoints pause your program when an expression changes value. The changing of value is what you're normally interested in so regular watchpoints are usually sufficient. Use a conditional watchpoint to pause your program when an expression changes to an interesting value.

```
watch x if x > 50
```

Second, use the `condition` command to turn a regular breakpoint into a conditional one. With the `condition` command you can set your breakpoints in Xcode and use the console to make certain ones conditional. Supply a breakpoint number and a condition with the `condition` command. The example below shows how you would use the `condition` command to pause when the `Move()` function moves north, assuming the breakpoint number is 1.

```
condition 1 directionToMove == NORTH
```

You can also use the `condition` command to make watchpoints and catchpoints conditional. Use conditional catchpoints to pause your program when it raises a specific exception.

To make a conditional breakpoint, watchpoint, or catchpoint unconditional, use the `condition` command. Supply a breakpoint number and no condition, and `gdb` removes the condition from the breakpoint, as you can see in the example below.

```
condition 1
```

Closely related to conditional breakpoints is `gdb`'s ignore count. The ignore count lets you skip a breakpoint when your program reaches it. Normally the ignore count is zero, meaning you won't skip any breakpoints, but you can use the `ignore` command to ignore a breakpoint. You supply a breakpoint number and a desired ignore count, and `gdb` will ignore that breakpoint the number of times you specify. The following example skips breakpoint 2 seven times:

```
ignore 2 7
```


When you reach a breakpoint in your program, you can set the ignore count for that breakpoint when you resume if you use the `continue` command from the console instead of clicking the Continue button in the Xcode debugger window. The following example sets the ignore count to nine:

```
continue 9
```

Disabling and Deleting Breakpoints

Let's say you set a breakpoint inside a loop. If the loop runs hundreds of times, you may not want to step through your code every time through the loop. Disabling the breakpoint turns it off but keeps it in the breakpoints list so you can turn it back on later.

The `disable` command disables breakpoints, watchpoints, and catchpoints. You supply a breakpoint number or a range of breakpoint numbers. The `info breakpoints` command shows your program's breakpoint numbers.

```
disable 4  
disable 3-6
```

To enable disabled breakpoints, watchpoints, and catchpoints, use the `enable` command. Like the `disable` command, you supply either a breakpoint number or a range of numbers.

```
enable 2  
enable 8-13
```

Calling `enable` like I did in the previous example causes the breakpoint to execute every time your program hits the breakpoint. Calling `enable` with the `once` option disables the breakpoint after it stops your program once.

```
enable once 1  
enable once 5-6
```

Calling `enable` with the `delete` option deletes the breakpoint after it stops your program once.

```
enable delete 9  
enable delete 2-7
```

To delete a breakpoint, watchpoint, or catchpoint, use the `delete` command. Like the `enable` and `disable` commands, you can supply a breakpoint number or a range of numbers.

```
delete 16
delete 11-14
```

If you want to delete the breakpoint you've just reached, use the `clear` command. Supply no arguments to delete the breakpoint you reached. You can also supply a line number or a function where you want to delete the breakpoint.

```
clear      // Clear the current breakpoint
clear main.m:15 // Clear a breakpoint at a line
clear PlayMusic // Clear a C function breakpoint
clear TMusic::Play // Clear a C++ member function
breakpoint
clear -[TMusic play:] // Clear an Objective-C method
breakpoint
```

Command Lists

Command lists allow you to run a series of commands when the debugger hits a breakpoint, watchpoint, or catchpoint. A command list adheres to the following format:

```
commands [Breakpoint Number]
    Insert the commands you want to execute here
end
```

If you create the command list immediately after creating the breakpoint (or watchpoint or catchpoint) you do not have to supply a breakpoint number; `gdb` attaches the command list to the breakpoint you just created. Use the `info breakpoints` command to get the numbers of all your program's breakpoints, watchpoints, and catchpoints.

When you start building a command list by typing commands, the prompt changes from `(gdb)` to `>` and will not revert back to `(gdb)` until you finish building the list by typing `end`. Many command lists have `silent` as the first command. The `silent` command suppresses the printing of the message that you hit a breakpoint, which helps if your command list prints information to the console. For the `silent` command to work, it must be the first command in a command list.

You can execute as many commands as you want in a command list, and you can use any `gdb` command in the list. The following command list writes the contents of an integer variable `y` to the console and continues the program's execution when it reaches the first breakpoint in the program:

```

commands 1
    silent
    printf "y = %d \n", y
    continue
end

```

You can also specify a command list when you create a breakpoint. The following example demonstrates creating a command list after creating a conditional breakpoint:

```

break main.m:24 if y > 5
commands
    silent
    printf "y = %d \n", y
    Continue
end

```

NOTE

You don't have to indent the commands in the `gdb` console. I indented the examples to make them easier to read.

Examining Data

Xcode's debugger window lets you view your variables and their values. The memory browser lets you view the contents of memory. From the `gdb` console you can view dynamic arrays. You can also tell the console to show the values of variables automatically when your program stops execution.

Examining Dynamic Arrays

Many programs use pointers to create arrays when the size cannot be determined until the program runs. To create these dynamic arrays, you declare a pointer variable, then use the `malloc()` (in C) or `new()` (in C++) calls to make the pointer large enough to store the array. When you look at the pointer variable, you want to look at the array, not the variable.

Artificial arrays solve the pointer problem. The `@` operator defines the array. The array goes to the left of the `@` operator, and the length of the array goes to the right of the array. Use the `print` command to look at the contents of the array.

```
print *array@length
```

The length is the number of elements you want to display. Suppose you have a 1000 element array, `myArray`, and you want to look at the first 50 elements. You would view the 50 elements with the following command:

```
print *myArray@50
```

If you want to start looking from a position other than the start of the array, use the following notation:

```
print *(array + starting point)@length
```

If you want to skip the first 100 elements and look at the next 25, you would run the following command:

```
print *(myArray + 100)@25
```

Displaying Data Automatically

When you're debugging, there's a few variables you're especially interested in. `gdb` allows you to place these variables in a display list, which `gdb` displays every time your program stops. Display lists keep you from having to click a series of disclosure triangles in the variable viewer to look at an important variable.

To add a variable to the display list, use the `display` command. Supply a variable name, and `gdb` adds it to the list.

```
display x
```

You can also supply any of the viewing formats in Table 5.3 to tell `gdb` to display the data the way you want. The following example displays a variable as a floating-point number:

```
display/f y
```

Table 5.3 Data Viewing Formats

Format	Description
x	Display as a hexadecimal (base 16) integer. This is the default for viewing memory.
d	Display as a signed decimal integer.
u	Display as an unsigned decimal integer.
o	Display as an octal (base 8) integer.
t	Display as a binary (base 2) integer.
a	Display as hexadecimal address and as an offset from the nearest symbol.
c	Display as a character constant.
f	Display as a floating-point number.

To see a list of all the variables you're automatically displaying, use the `info display` command. It tells you the following pieces of information for each variable:

- The ID number, which you can use to turn off automatic display.
- Whether or not it's enabled, y for yes and n for no.
- The expression to display.

To temporarily disable a variable from the automatic display list, use the `disable display` command. You supply ID numbers (separated by spaces) to `disable display`, which you can see in the following examples:

```
disable display 5    // Disable one variable
disable display 1 3 6 // Disable multiple variables
```

The `enable display` command enables variables you disabled with the `disable display` command. Supply ID numbers to `enable display`. The following examples enable the variables I disabled in the previous example:

```
enable display 5
enable display 1 3 6
```

To permanently remove variables from the automatic display list, use either the `undisplay` or `delete display` commands. Like the `enable display` and `disable display` commands, you supply a list of ID numbers separated by spaces, which you can see in the following examples:

```
undisplay 3
undisplay 2 4 5 9
delete display 7
delete display 2 6 13
```

You can place memory locations in automatic display lists so you can look at important memory addresses every time your program stops. Use the `display` command, supplying the memory address. Table 5.4 lists the memory units you can use to display memory. The following are some examples of using display lists:

```
display/64b myPtrVariable // Display 64 bytes automatically
display/16i 0x23459900 // Display 16 machine instructions
display/100fg 0x12123400 // 100 giant words as floating-point
```

Table 5.4 Memory Units

Unit	Letter You Pass to gdb	Bytes
Bytes	b	1
Halfwords	h	2
Words	w	4
Giant Words	g	8
Machine Instructions	i	N/A
String	s	N/A

Executing Shell Commands

The `shell` command lets you execute a Unix shell command in `gdb`. It takes the following form:

```
shell CommandName
```

With the `shell` command you can run another program in the `gdb` console. The following command runs the `heap` program that shows your program's memory heap:

```
shell heap AppName
```

The `shell` command also lets you run shell scripts, giving you enormous power if you know how to write Unix shell scripts. You can also place shell commands in a command list to execute scripts of shell commands when your program hits a breakpoint.

Defining Your Own Commands

If you find yourself entering a sequence of commands repeatedly, define a command to save yourself some typing. A user-defined command consists of a series of `gdb` commands. User-defined commands are similar to command lists. The difference is command lists execute when your program reaches a breakpoint while user-defined commands execute when you submit the command.

The first command in a user-defined command is the `define` command, which gives the command a name. The `define` command takes the following form:

```
define CommandName
```

After entering the `define` command, the console prompts you to enter the `gdb` commands you want to place in the user-defined command. A user-defined command can have up to ten arguments, with the first argument having the name `$arg0` and the tenth argument having the name `$arg9`. The `end` command is the equivalent of `}` in a C or Java program. Use the `end` command to end your user-defined command and return the prompt in the console to normal.

Let's look at an example of a simple user-defined command. Suppose you want to create a command that prints the contents of a dynamic array. You would define the following command:

```
define PrintArray
    print *($arg0 + $arg1)@$arg2
end
```

In this example `$arg0` is the pointer to the array, `$arg1` is the first element you want to display, and `$arg2` is the number of elements to display. To display the first 20 elements of an array called `myArray`, you would call the `PrintArray` command you created.

```
PrintArray myArray 0 20
```

`gdb` cannot alert you to errors in your user-defined command as you're entering the `gdb` commands. Running the command is the only way to know you wrote the command correctly. You have the responsibility of making sure you entered the `gdb` commands correctly.

Conditional Commands

You may want some of the `gdb` commands in your user-defined command to execute multiple times and other commands to execute conditionally. `gdb` has `if` and `while` commands that work similarly to `if` and `while` statements in C. Use the `if` and `while` commands to create conditional commands in your user-defined command.

The `if` command executes a series of `gdb` commands if a condition is true. It takes the following form:

```
if Condition
    Commands
else
    Commands
end
```

To make the `PrintArray` command example check for a negative starting point and for a negative number of items, you would use the following command:

```
define PrintArray
    if ($arg1 < 0) || ($arg2 < 0)
        printf "ERROR! \n"
    else
        print *($arg0 + $arg1)@$arg2
    end
end
```

The `while` command executes a series of `gdb` commands repeatedly until a condition is false. It takes the following form:

```
while Condition
    Commands
end
```

The example below prints the first `x` elements of a static array one element at a time, where `x` is a number you supply.

```
define PrintStaticArray
    set $array = $arg0
    set $index = 0
    while ($index < $arg1)
        print $array[$index]
        printf("\n")
        set $index = $index + 1
    end
```



```
end
```

The variables `$array` and `$index` are convenience variables. Convenience variables store values in `gdb` so you can use the values later in the debugging session. They have no effect on your program. In the `PrintStaticArray` example the `$index` convenience variable keeps track of where `gdb` currently is in the array. Convenience variables start with `$`. The name of the convenience variable cannot be a number.

```
$3 // Invalid convenience variable name
```

`gdb` uses variable names with numbers as a value history of variables you print to the console. When you print a variable, `gdb` creates a value history variable and displays the value history variable in the console. You can refer to value history variables in other `gdb` commands.

Documenting Your Commands

If you create commands for other programmers to use, you should document the commands so the other programmers know what the command does. The `show user` command displays the `gdb` commands that make up a user-defined command. If you supply no arguments, the `show user` command displays the `gdb` commands for all user-defined commands. Supply a command name to show the `gdb` commands for a single user-defined command.

```
show user CommandName
```

To create more documentation about a command, use the `document` command. It takes the following form:

```
document CommandName
```

After executing the `document` command, the console prompts you to enter the documentation. Press the Return key to end a paragraph of documentation. When you're finished entering the documentation, press the Return key and type `end`. Now when someone runs `help` on your command, the documentation you entered appears in the console.

Reading Commands from a File

Defining your own commands is a powerful tool, but there are problems if you define commands you want to use over a long period of time. Entering the commands every time you launch `gdb` becomes annoying after a while. Plus, you could make a typographical error, forcing you to reenter the command.

The solution is to place the commands you want to define in a file and load the file when you launch `gdb`. Place one `gdb` command per line. The character `#` signifies a comment. `gdb` ignores blank lines and tabs so you can indent commands to make them easier to read. You can also enter documentation for your commands in the file. Make sure to define the command first. The following example shows how the `PrintArray` command would look in a file:

```
define PrintArray
    print *($arg0 + $arg1)@$arg2
end

document PrintArray

The PrintArray command displays the contents of dynamic
arrays. It takes three arguments.

$arg0  The array.
$arg1  First element to display.
$arg2  Number of elements to display.

end
```

The `source` command executes commands from a file. Supply a file name.

```
source Filename
```

In the case of user-defined commands, running the `source` command loads the commands and documentation from the file. After loading the commands from the file, you can execute them when needed.

To have your user-defined commands loaded when `gdb` starts, place the commands in a file and perform the following steps:

1. Move the file you created to your home directory, `/Users/YourUsername`.
2. Run the Terminal program to reach the command line .
3. Change the name of your file to `.gdbinit` from the command line using the `mv` command.

The Finder won't let you start a file's name with a period because the operating system allows only system files to begin with a period. To start a file's name with a period, you must change the file's name from the command line.

When `gdb` launches it loads the `.gdbinit` file and executes any commands in the file. In the case of user-defined commands, `gdb` defines the commands on startup so you can start using your commands immediately.

Command Hooks

A command hook lets you execute a series of `gdb` commands before or after a command executes. You can create command hooks for every `gdb` command as well as user-defined commands. A command hook has the same structure as a user-defined command.

```
define
    Commands in hook
end
```

When defining a command hook to execute before the command, the `define` command takes the following form:

```
define hook-CommandName
```

When defining a command hook to execute after the command, the `define` command takes the following form:

```
define hookpost-CommandName
```

The following example prints a header for the `PrintArray` command example:

```
define hook-PrintArray
    printf "Array Contents\n\n"
end
```