Xcode Debugging Guide

Tools > Xcode

2009-01-06

Contents

Figures and Tables

Introduction

Finding and eliminating bugs in your code is a critical phase of the development process. Xcode provides advanced debugging facilities, which include debugging from the text editor so that you don't stray far from your code, and using the mini debugger, which provides a graphical debugging experience that is less intrusive on the running application than other methods. You can also use a more traditional, specialized Debugger window, or the GDB debugger console.

This document describes the Xcode debugging environments. You'll learn how to take advantage of the capabilities each environment provides.

Organization of This Document

This document contains the following chapters:

- "Debugging Essentials" (page 9) provides a high-level summary of the Xcode debugging environments.
- "Debugging in the Text Editor" (page 11) describes the debugging environment Xcode provides in the text editor.
- "Debugging in the Mini Debugger" (page 17) explains how to use the mini debugger to debug programs unobtrusively.
- "Debugging in the Debugger Window" (page 19) describes the Debugger window.
- "Debugging in the Console" (page 25) discusses GDB Console window.
- "Managing Program Execution" (page 27) describes the mechanisms used to control and monitor the execution of programs.
- "Viewing Variables and Memory" (page 37) talks about the various ways in which you can view the values of variables as you debug your programs.
- "Modifying Running Code" (page 45) shows how you can modify your executable while it is running.
- "Debugging Programs Remotely" (page 49) describes how to debug a program running on another computer.
- "Low-Level Debugging Support" (page 53) talks about the Mac OS X debugging facilities that can help you in your debugging tasks.
- "Debugging Preferences" (page 55) describes the Debugging preferences pane.

Debugging Essentials

When you test your program by running it from Xcode, you may observe problems in its behavior. To pinpoint the cause of these problems, you may need to examine particular source code lines. Xcode lets you analyze your code line by line to view your program's state at a particular stage of execution. This process is called **debugging**. The process being debugged is known as the **inferior**.

To debug a program, you run it under the control of a **debugger**, which lets you pause programs and examine their state. Xcode provides an interface to several debuggers.

When you debug a product, Xcode opens a debugging session with a debugger and your product's binary. Before you can debug a program, however, it should contain as much debugging information as possible, so that Xcode can provide you with accurate and useful information during your debugging. See "Building for Debugging" in *Xcode Project Management Guide* for information about how to build a program to debug.

You may also debug a running process (not launched by Xcode). This operation is known as **attaching** to a process. You can attach to a program running under Xcode or to any process for which you have a process ID. To attach to a running program use Run > Attach to Process.

This menu lists currently running programs launched from Xcode, identified by the program name and the name of the corresponding Xcode project. (This menu also lists other programs running on the computer.) You can use the menu's Process ID option to attach to any process using its process ID, which you obtain using UNIX utilities such as `top`.

Note: When you launch a program from Xcode, you can also have Xcode automatically attempt to attach the debugger to the process when the program crashes. See "Executable-Environment Debugging Information" in *Xcode Project Management Guide* for details.

You can perform debugging operations in several ways:

- In the text editor.

  The text editor allows you to perform many debugging operations, such as stepping through code lines and viewing the contents of variables. Debugging in the text editor lets you be close to your code as you debug it.

- In the mini debugger.

  The mini debugger is a floating window that lets you perform debugging operations in way that is less intrusive to your code than debugging within Xcode. The mini debugger lets you debug your application in an environment that closely resembles the way you customers use your program.

- In the Debugger window.

  The Xcode debugger window provides a rich, traditional debugging experience.

- In the console window.

  Some debugging operations produce textual output, which is displayed in the console window. This window can also be used to issue commands directly to the debugger instead of through the Xcode debugging interface.

The rest of this chapter describes requirements and essential concepts of the Xcode debugging environment.

Building a Product with Debugging Symbols

Before you can take advantage of the source-level debugger, the compiler must collect information for the debugger. To generate debugging symbols for a product, enable the Generate Debug Symbols (`GCC_GENERATE_DEBUGGING_SYMBOLS`) build setting and build the product. This setting is enabled by default in the Debug build configuration. If you use this build configuration as the active build configuration when you build your target, the compiler generates the necessary debugging information.

To view the build settings that are set in the Debug build configuration:

1. In the project window, select the target or targets you want to build and bring up the target editor.
2. Click Build to bring up the Build pane.
3. Choose Debug (or Development) from the Configuration pop-up menu at the top of the pane. You should see the Generate Debug Symbols setting in the table of build settings. Make sure that this setting is turned on; if it is, a checkmark appears in the Value column for this setting. Otherwise, turn on the setting by clicking the checkbox in the Value column.

Important: When building your product using a build system other than Xcode, you must ensure that lazy-symbol loading is turned off. See "Debugging Preferences" (page 55) for more information.

For more information about building and running programs, see *Xcode Project Management Guide*.

DWARF Debug Information Format

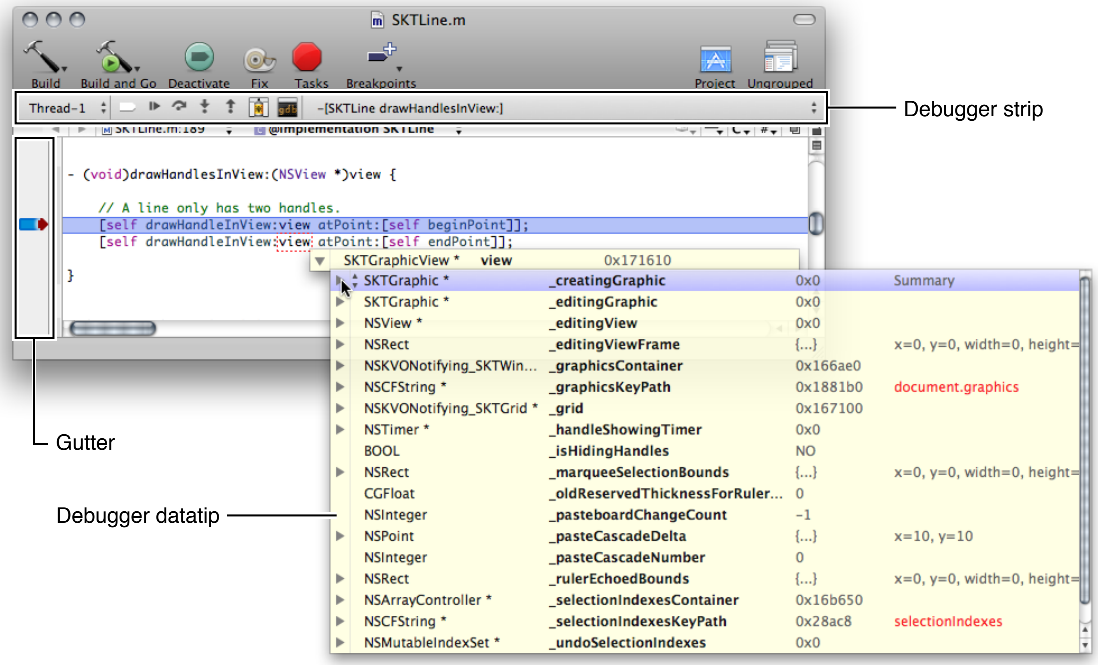
Xcode uses the DWARF debug information format to manage your programs debugging data. The Stabs format, supported in earlier releases of Xcode, is no longer supported.

Xcode can generate a separate file with your application's debug information. This feature considerably reduces the size of the binaries you create for the purpose of debugging applications.

Debugging in the Text Editor

You can perform many debugging tasks in the text editor, including controlling program flow, managing breakpoints and watchpoints, and viewing program memory. Figure 2-1 shows the text editor and the debugging controls it provides.

Figure 2-1 Debugging in the text editor



```
- (void)drawHandlesInView:(NSView *)view {

    // A line only has two handles.
    [self drawHandleInView:view atPoint:[self beginPoint]];
    [self drawHandleInView:view atPoint:[self endPoint]];

}
```

```
SKTGraphicView *   view      0x171610
```

SKTGraphic *	_creatingGraphic	0x0	Summary
SKTGraphic *	_editingGraphic	0x0	
NSView *	_editingView	0x0	
NSRect	_editingViewFrame	{...}	x=0, y=0, width=0, height=
NSKVONotifying_SKTWin...	_graphicsContainer	0x166ae0	
NSCFString *	_graphicsKeyPath	0x1881b0	document.graphics
NSKVONotifying_SKTGrid *	_grid	0x167100	
NSTimer *	_handleShowingTimer	0x0	
BOOL	_isHidingHandles	NO	
NSRect	_marqueeSelectionBounds	{...}	x=0, y=0, width=0, height=
CGFloat	_oldReservedThicknessForRuler...	0	
NSInteger	_pasteboardChangeCount	-1	
NSPoint	_pasteCascadeDelta	{...}	x=10, y=10
NSInteger	_pasteCascadeNumber	0	
NSRect	_rulerEchoedBounds	{...}	x=0, y=0, width=0, height=
NSArrayController *	_selectionIndexesContainer	0x16b650	
NSCFString *	_selectionIndexesKeyPath	0x28ac8	selectionIndexes
NSMutableIndexSet *	_undoSelectionIndexes	0x0	

These are the controls identified in Figure 2-1:

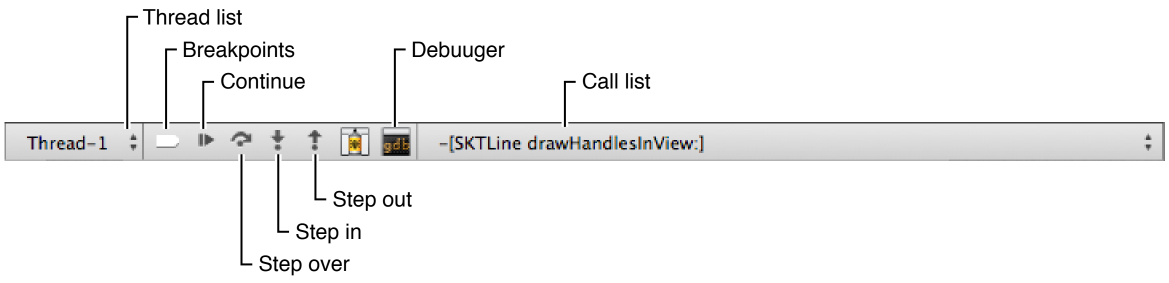
- **Debugger strip.** This is a small control strip that appears above the editor's content pane while debugging.
- **Gutter.** The gutter's shortcut (contextual) menu gives you access to a few debugging commands.
- **Debugger datatips.** Hovering the pointer over a variable displays the variable's value and lets you modify it.

This chapter describes these controls in detail.

Debugger Strip

The **debugger strip** (Figure 2-2) is a small control strip that appears above the content pane. It lets you perform several debugging tasks.

Figure 2-2 Debugger strip



These are the items in the debugger strip:

- **Thread list:** List of the threads in the inferior.
- **Breakpoints:** Activates/deactivates breakpoints.
- **Continue:** Continues execution of a paused inferior.
- **Step over:** Steps over the current code line. The process counter (PC), identified by the red arrow in the gutter, moves to the next code line to be executed in the current file.
- **Step in:** Steps into a function or method in the current code line. If possible, the editor shows the source file containing the called routine an the process counter appears in the code line to be executed next.
- **Step out:** Steps out of the current function or method. The editor shows the source file containing the caller.
- **Debugger:** Opens the Debugger window.
- **Call list:** List of the called routines, also known as the *call stack* or *stack frame*.

Gutter Shortcut Menu

The text editor gutter includes several shortcuts to debugging facilities. The code line indicated by the pointer at the time you choose the shortcut is the **action line**. These include:

- **Continue to Here:** Continues program execution up to the action line.
- **Add Breakpoint:** Adds a breakpoint to the action line.
- **Add & Edit Breakpoint:** Adds a breakpoint to the action line and opens the breakpoints window.
- **Built-in Breakpoints:** Adds a predefined breakpoint to the action line.
- **Reveal in Breakpoints:** Opens the action line's breakpoint in the breakpoints window.
- **Breakpoints On:** Activates breakpoints for the current debugging session.

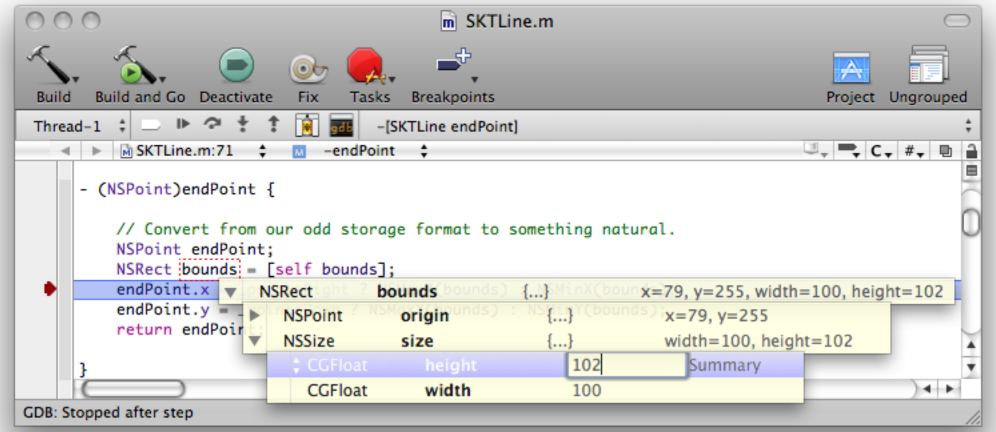
- **Breakpoints Off:** Deactivates breakpoints for the current debugging session.

Debugger Datatips

As you debug your program in the text editor, you may need to analyze the contents of the program's variables as you step through code lines. Xcode provides *debugger datatips* to let you view and change your program's variables. A **debugger datatip** is a control that provides access to the contents variables using a progressive disclosure mechanism driven by hovering the pointer over the control. You can also modify the contents of mutable variables using datatips.

Figure 2-3 shows a debugger datatip showing the contents of the `bounds` variable. In addition to viewing the variable's value, the figure shows how you can modify the value. In this case, after double-clicking the value of the `height` field of the `size` structure, you can change it to another value before executing the code line that uses the `bounds` variable.

Figure 2-3 Changing variables with debugger datatips in the text editor



```
- (NSPoint)endPoint {

    // Convert from our odd storage format to something natural.
    NSPoint endPoint;
    NSRect bounds = [self bounds];
    endPoint.x
    endPoint.y =
    return endPoi

}
```

```
▼ NSRect    bounds    {...}   x=79, y=255, width=100, height=102
  ▶ NSPoint   origin    {...}   x=79, y=255
  ▼ NSSize    size      {...}   width=100, height=102
      CGFloat   height    102   Summary
      CGFloat   width     100
```

GDB: Stopped after step

As you move the pointer over a disclosure triangle in a datatip row, the contents of the field the row represents are disclosed bellow that row. (You can turn off this behavior, as explained later.) When you hover the pointer to the right of the disclosure triangle control with two small triangles appears. Clicking that control shows the datatip menu. The datatip menu provides the following commands.

- **Print Description:** Prints the description of the current datatip field in the console.
- **Open in Window:** Opens a window containing the data of the current datatip field.
- **View as Memory:** Shows the contents of the current datatip field in the memory viewer window. See "Browsing Memory" (page 41) for more information.
- **Jump to Definition:** Opens the file that declares the data type of the current datatip field.
- **Jump to Documentation:** Opens the reference for the data type of the current datatip field.
- **Show Types:** Toggles the display of the data type of the datatip fields.

- **Show Data Formatters:** Toggles the display of data formatters for the datatip fields.
- **Sort by Name:** Sorts datatip fields by field name.
- **Sort by Type:** Sorts datatip fields by field data type.
- **No Sort:** Applies no sorting to datatip fields.
- **Auto Expand:** Toggles the autoexpansion of datatip fields that represent structures when the pointer hovers over the corresponding disclosure triangle.

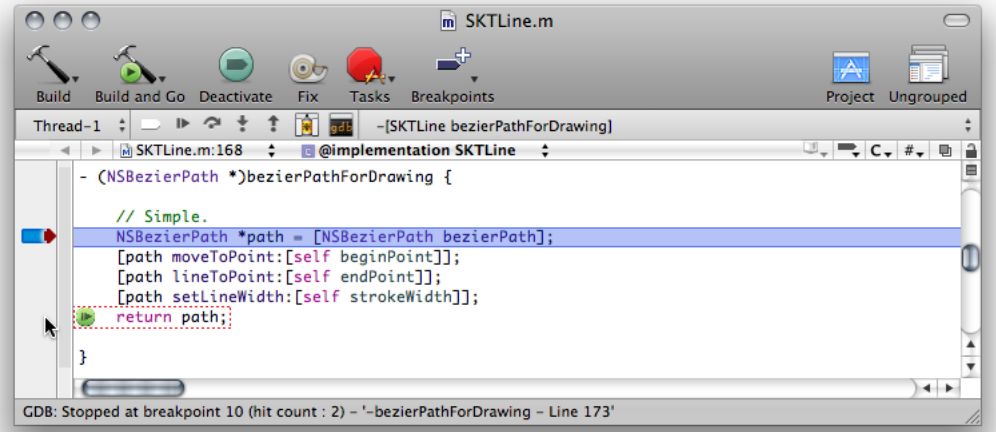
Debugger datatips also provide program-flow–control controls called *step controls*. **Step controls** allow you to perform perform Continue, Step In, and Step Over commands from the content pane of the text editor. To turn on step controls, use Run > Debugger Display > Datatips > Step Controls.

These are the commands step controls provide:

- **Continue to Here**

  To perform this command, hover the pointer over the gutter identifying the action line. The continue to here icon appears on the left margin of the line (Figure 2-4). Clicking this icon continues program execution up to the action line.

  Figure 2-4 Continuing execution to a particular code line

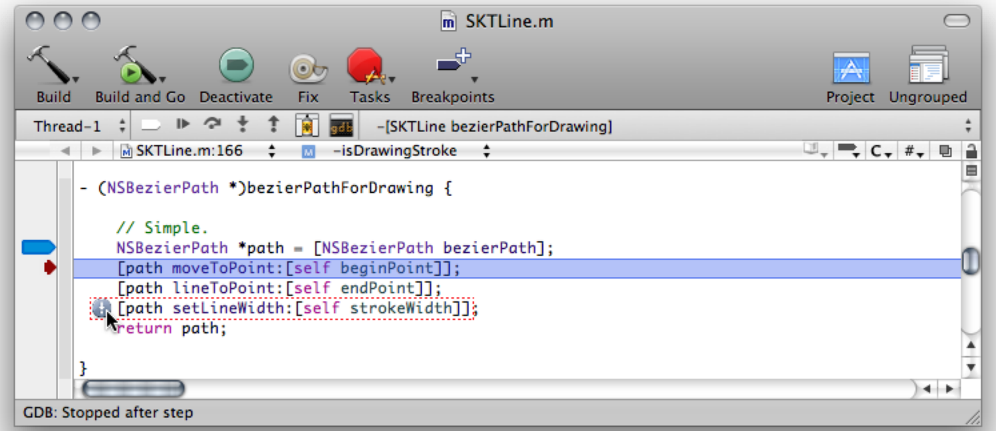
```
- (NSBezierPath *)bezierPathForDrawing {

    // Simple.
    NSBezierPath *path = [NSBezierPath bezierPath];
    [path moveToPoint:[self beginPoint]];
    [path lineToPoint:[self endPoint]];
    [path setLineWidth:[self strokeWidth]];
    return path;

}
```

  GDB: Stopped at breakpoint 10 (hit count : 2) - '-bezierPathForDrawing - Line 173'

- **Step In** and **Step Over**

  To perform these commands, hover the pointer over the action line in the gutter or the content pane until the Step In or the Step Over icon appears on the left margin of the line (Figure 2-5). Click the icon to perform the command.

Figure 2-5 Stepping into a call



```
- (NSBezierPath *)bezierPathForDrawing {

    // Simple.
    NSBezierPath *path = [NSBezierPath bezierPath];
    [path moveToPoint:[self beginPoint]];
    [path lineToPoint:[self endPoint]];
    [path setLineWidth:[self strokeWidth]];
    return path;

}
```

GDB: Stopped after step

Debugging in the Mini Debugger

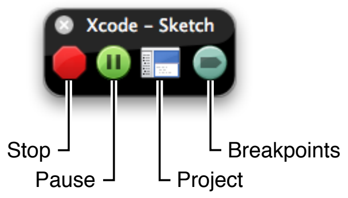
Sometimes, debugging your program under Xcode may present subtle differences in the program's running environment that make it difficult to reproduce problems your customers experience. In this case, you need an unobtrusive debugging experience. The Xcode mini debugger provides that experience.

The mini debugger is a window that floats over other windows and that provides debugging controls equivalent to those present in the text editor and the Debugger window. The mini debugger has two modes, depending on whether the inferior is running or stopped. When the inferior is running, the mini debugger provides controls to pause and stop the inferior, display the inferior's Xcode project, and activate/deactivate breakpoints. When the inferior is stopped (after reaching a breakpoint, for example), the mini debugger provides a debugging experience similar to the one provided by the text editor. In fact, when the your program is stopped, the mini debugger's entire content is made up of a text editor, although you cannot edit the source files it displays.

To show the mini debugger while running an application, choose Run > Mini Debugger.

Figure 3-1 shows the mini debugger with a running inferior.

Figure 3-1 Mini debugger with the inferior running



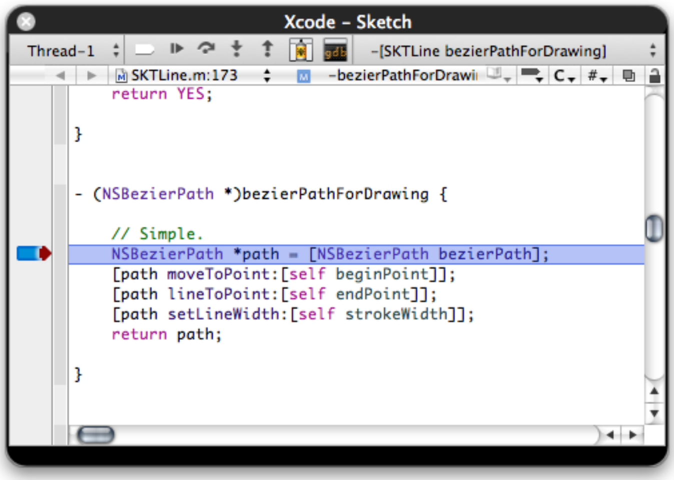
You can tell Xcode to always open the mini debugger when you debug a product using the On Start menu in Xcode Preferences > Debugging. See "Debugging Preferences" in *Xcode Workspace Guide* for more information.

These are the icons the mini debugger shows when the inferior is running:

- **Stop:** Terminates the inferior and closes the mini debugger.
- **Pause:** Pauses the inferior and engages the mini debugger.
- **Project:** Opens the inferior's Xcode project.
- **Breakpoints:** Activates/deactivates breakpoints.

Figure 3-2 shows the mini debugger after the inferior stops.

Figure 3-2 Mini debugger: Engaged



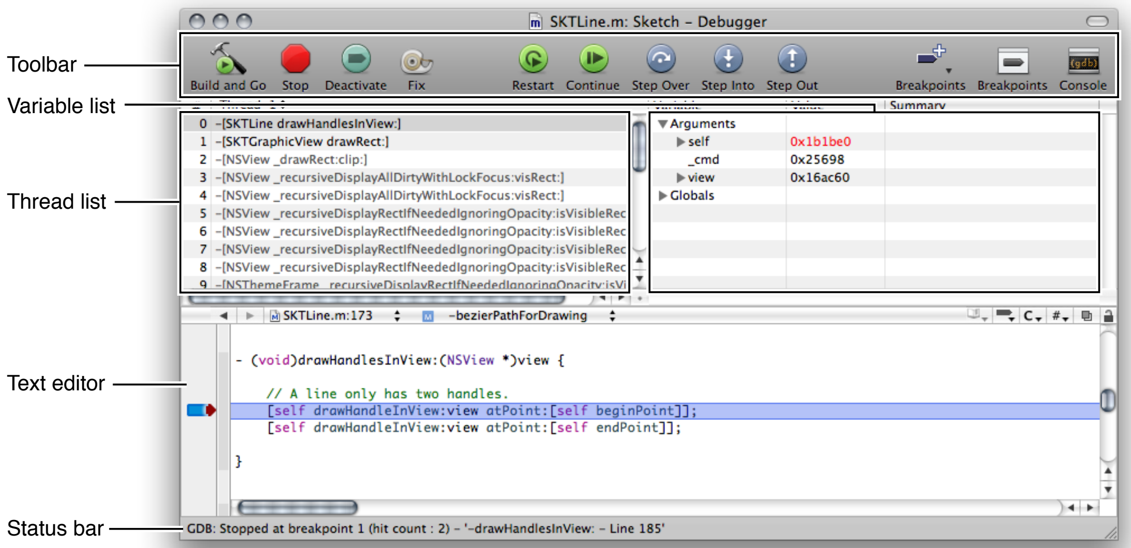
```
    return YES;

}


- (NSBezierPath *)bezierPathForDrawing {

    // Simple.
    NSBezierPath *path = [NSBezierPath bezierPath];
    [path moveToPoint:[self beginPoint]];
    [path lineToPoint:[self endPoint]];
    [path setLineWidth:[self strokeWidth]];
    return path;

}
```

For more information about debugging in the mini debugger, see "Debugging in the Text Editor" (page 11). Keep in mind that the text editor in the mini debugger doesn't allow you to edit source files.

Debugging in the Debugger Window

The Xcode Debugger window offers a traditional but rich debugging experience. Figure 4-1 shows the Debugger window.

Figure 4-1 The Debugger window



```
0  -[SKTLine drawHandlesInView:]
1  -[SKTGraphicView drawRect:]
2  -[NSView _drawRect:clip:]
3  -[NSView _recursiveDisplayAllDirtyWithLockFocus:visRect:]
4  -[NSView _recursiveDisplayAllDirtyWithLockFocus:visRect:]
5  -[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRec
6  -[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRec
7  -[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRec
8  -[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRec
9  -[NSThemeFrame _recursiveDisplayRectIfNeededIgnoringOpacity:isVi
```

Variable	Value
self	0x1b1be0
_cmd	0x25698
view	0x16ac60

```
- (void)drawHandlesInView:(NSView *)view {

    // A line only has two handles.
    [self drawHandleInView:view atPoint:[self beginPoint]];
    [self drawHandleInView:view atPoint:[self endPoint]];

}
```

```
GDB: Stopped at breakpoint 1 (hit count : 2) – '-drawHandlesInView: – Line 185'
```

Here's what the Debugger window contains:

- **Toolbar**
  - ❏ **Build and Go:** build and run the product.
  - ❏ **Stop:** Terminates the inferior.
  - ❏ **Activate/Deactivate:** Toggles breakpoints.
  - ❏ **Fix:** Compiles a single file fix and modifies your executable to run the changed code without stopping the current debugging session. For more information on how to use this feature, see "Modifying Running Code" (page 45).
  - ❏ **Restart:** Runs the product the same it was run the last time.
  - ❏ **Continue:** Continues program execution.
  - ❏ **Step Over:** Steps over the current code line.
  - ❏ **Step Into:** Steps into the current code line's call.
  - ❏ **Step Out:** Steps out of the current method or function.

- ❏ **Breakpoints (+):** Adds a predefined breakpoint.
- ❏ **Breakpoints:** Opens the breakpoints window.
- ❏ **Console:** Opens the console window.

- **Thread list**

  Displays the call stack of the current thread. The pop-up menu above this view lets you select different threads to view when debugging a multi-threaded application.

- **Variable list**

  Shows the variables defined in the current scope and their values. This section also shows the current state of all processor registers when the disassembly view is enabled (see "Viewing Disassembly Code and Processor Registers" (page 24) for more information).

- **Text editor**

  This pane displays the source code you are debugging. When execution of your program is paused, the debugger indicates the line at which execution is paused by displaying the PC indicator, which appears as a red arrow. The line of code is also highlighted. You can change the color used to highlight the currently executing statement with the Instruction Pointer Highlight color well in Xcode > Preferences > Debugging.

  The text editor pane offers alternate ways to execute debugging commands. See "Gutter Shortcut Menu" (page 12) and "Debugger Datatips" (page 13) for details.

- **Status bar**

  Displays the current status of the debugging session. For example, in the window shown above, Xcode indicates that GDB has just finished loading symbols for a single shared library.

You can change the layout of the Debugger window by choosing one of the following options:

- Run > Debugger Display > Horizontal Layout
- Run > Debugger Display > Vertical Layout

Troubleshooting debugger display:

If the debugger does not display source code, try the following:

- Make sure you have the source. Apple's frameworks and many third-party libraries don't include source code
- Make sure the product contains debugging information. See "Building for Debugging" in *Xcode Project Management Guide* and "Executable-Environment Debugging Information" in *Xcode Project Management Guide*.
- If the file is in the Groups & Files list, make sure its name is not in red (red means Xcode can't find the file).
- If the file is not in the Groups & Files list and your target may need to process it, add the file to the project. See Files in a Project.
- If the file is for a library or framework that was built for you, do one of the following:
  - Place the source file in the same location used by the person who built the library or framework. When someone builds a debuggable binary, the compiler stores the paths of its source files in the binary.
  - Add the directory that contains the file to the source directories list. Enter the directory's pathname in:

    Executable environment editor: Debugging > Additional directories to find source files in

The Debugger Display menu also lets you specify whether to show source code and/or disassembly code. See "Viewing Disassembly Code and Processor Registers" (page 24) to learn more about viewing disassembly code.

Viewing Stack Frames in the Debugger Window

For each function call that your program makes, the debugger stores information about that call in a **stack frame**. These stack frames are stored in the **call stack**. When execution of your program is paused in the debugger, Xcode displays the call stack for the currently running process in the Thread list and puts the most recent call at the top.

Selecting any function call in the call stack displays the stack frame for that function. The stack frame includes information on the arguments to the function, variables defined in the function, and the location of the function call. Xcode displays the frame's variables in the Variable list and displays its currently executing statement in the text editor with the process counter (PC)—a red arrow. If a stack frame is grayed out, no source code is available for it.

To learn about viewing the stack frame in the text editor, see "Debugger Strip" (page 12).

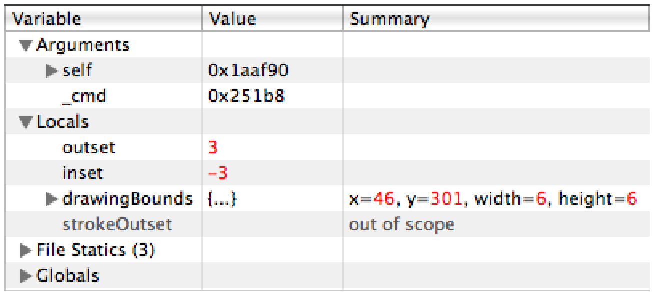
Viewing Variables in the Debugger window

The variables view shows information—such as name, type and value—about the variables in your program. Variables are displayed for the stack frame that is currently selected in the Debugger window. The variables view, shown in Figure 4-2 (page 22), appears in upper-right portion of the Debugger window by default. The variables view can have up to four columns:

1. The Variable column shows the variable's name.
2. The Type column displays the type of the variable. This column is optional. To display it, choose Debug > Variables View > Show Types.
3. The Value column shows the contents of the variable. If a variable's value is in red, it changed when the application was last active. You can edit the value of any variable; the changed value is used when you resume execution of your program.
4. The Summary column gives more information on the contents of a variable. It can be a description of the variable or an English language summary of the variable's value. For example, if a variable represents a point, its summary could read "(x = *x value*, y = *y value*)." You can edit the summary of a variable by double-clicking the Summary column or choosing Debug > Variables View > Edit Summary Format. For a description of how you can format variable summaries, see "Using Data Formatters" (page 37).

You can choose which columns the debugger shows in the Variable view; Xcode remembers these columns across debugging sessions.

Figure 4-2 Variable list



Variable	Value	Summary
▼ Arguments		
▶ self	0x1aaf90	
_cmd	0x251b8	
▼ Locals		
outset	3	
inset	-3	
▶ drawingBounds	{...}	x=46, y=301, width=6, height=6
strokeOutset		out of scope
▶ File Statics (3)		
▶ Globals		

Variables in the Variable view are grouped by category, as shown in Figure 4-2. To view variables in any of these groups, click the disclosure triangle next to that group. These groups are:

- The Arguments group contains the arguments to the function that is currently selected in the call stack.
- The Locals group contains the local variables declared in the function that is currently selected in the call stack.
- The Globals group shows global variables and their values. By default, there are no global variables in this section; you must select those variables you want to track in the Globals Browser, described in "Viewing Global Variables" (page 23).
- The File Statics group shows file statics, if any. This group is not shown if none are present.

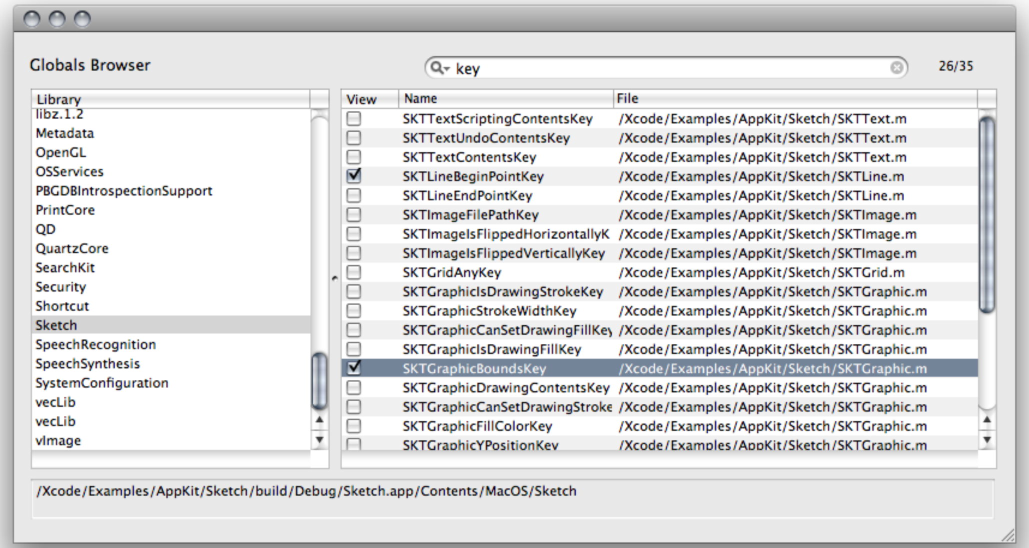
To view the contents of a structured variable, click the disclosure triangle beside the variable's name. You can also use a data formatter to display a variable's contents in the Summary column, as described in "Using Data Formatters" (page 37), or you can view a variable in its own window. Viewing a variable in its own window is particularly useful for viewing the contents of complex structured variables. To open a variable in its own window, double-click the variable's name or select it and choose Debug > Variables View > View Variable in Window.

To learn how to view variables in the text editor, see "Debugger Datatips" (page 13).

Viewing Global Variables

The Variable list contains a Globals group, which is initially empty. You can choose which global variables to display in the Globals group using the Globals Browser, shown in Figure 4-3. The Globals Browser lets you search for global variables by library.

Figure 4-3 Globals Browser



Library
libz.1.2
Metadata
OpenGL
OSServices
PBGDBIntrospectionSupport
PrintCore
QD
QuartzCore
SearchKit
Security
Shortcut
Sketch
SpeechRecognition
SpeechSynthesis
SystemConfiguration
vecLib
vecLib
vImage

View	Name	File
☐	SKTTextScriptingContentsKey	/Xcode/Examples/AppKit/Sketch/SKTText.m
☐	SKTTextUndoContentsKey	/Xcode/Examples/AppKit/Sketch/SKTText.m
☐	SKTTextContentsKey	/Xcode/Examples/AppKit/Sketch/SKTText.m
✓	SKTLineBeginPointKey	/Xcode/Examples/AppKit/Sketch/SKTLine.m
☐	SKTLineEndPointKey	/Xcode/Examples/AppKit/Sketch/SKTLine.m
☐	SKTImageFilePathKey	/Xcode/Examples/AppKit/Sketch/SKTImage.m
☐	SKTImageIsFlippedHorizontallyK	/Xcode/Examples/AppKit/Sketch/SKTImage.m
☐	SKTImageIsFlippedVerticallyKey	/Xcode/Examples/AppKit/Sketch/SKTImage.m
☐	SKTGridAnyKey	/Xcode/Examples/AppKit/Sketch/SKTGrid.m
☐	SKTGraphicIsDrawingStrokeKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicStrokeWidthKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicCanSetDrawingFillKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicIsDrawingFillKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
✓	SKTGraphicBoundsKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicDrawingContentsKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicCanSetDrawingStroke	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicFillColorKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m
☐	SKTGraphicYPositionKey	/Xcode/Examples/AppKit/Sketch/SKTGraphic.m

/Xcode/Examples/AppKit/Sketch/build/Debug/Sketch.app/Contents/MacOS/Sketch

To open the Globals Browser, choose Run > Show > Global Variables.

The debugger must be running and execution of the program being debugged must be paused for this item to be available. If you attempt to open of the Globals group when it's empty, Xcode automatically opens the Globals Browser.

The Library list, on the left of the Globals Browser, lists the available libraries, including system libraries and your own libraries. To see a library's global variables, select that library in the list; the global variables defined by that library are shown in the table to the right. In the globals table, you can see:

- The name of the global variable
- The file in which the global variable is defined

You can use the search field at the top of the Globals Browser window to filter the contents of the global variables table. To the right of the search field, Xcode displays the number of global variables currently visible, as a fraction of the total number of global variables in the currently selected library.

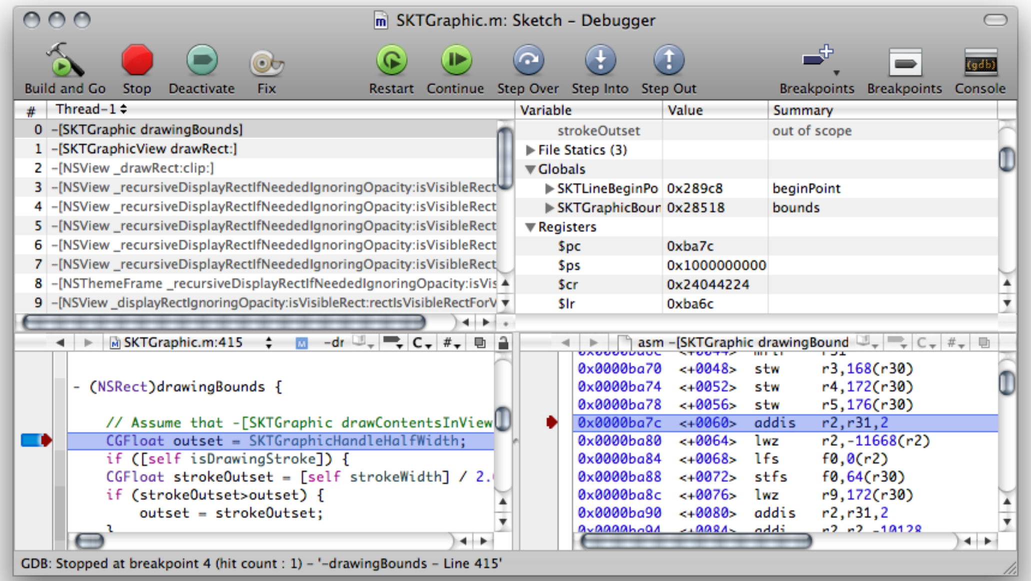
To add a global variable to the Globals list in the Debugger window variable view, select the checkbox in the "View" column next to the global variable.

When you select a library in the Library list, the full path to that library is displayed below the list.

Viewing Disassembly Code and Processor Registers

When you choose to display disassembly code or when no source code is available for the function or method selected in the Thread list, Xcode displays a pane with disassembled code in the debugger. Figure 4-4 shows disassembled code in the debugger after choosing to view source and disassembly code.

Figure 4-4 Disassembled code in the Debugger window



#	Thread-1
0	–[SKTGraphic drawingBounds]
1	–[SKTGraphicView drawRect:]
2	–[NSView _drawRect:clip:]
3	–[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect
4	–[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect
5	–[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect
6	–[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect
7	–[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect
8	–[NSThemeFrame _recursiveDisplayRectIfNeededIgnoringOpacity:isVis
9	–[NSView _displayRectIgnoringOpacity:isVisibleRect:rectIsVisibleRectForV

Variable	Value	Summary
strokeOutset		out of scope
▶ File Statics (3)		
▼ Globals		
▶ SKTLineBeginPo	0x289c8	beginPoint
▶ SKTGraphicBour	0x28518	bounds
▼ Registers		
$pc	0xba7c	
$ps	0x1000000000	
$cr	0x24044224	
$lr	0xba6c	

```
- (NSRect)drawingBounds {

    // Assume that -[SKTGraphic drawContentsInView
    CGFloat outset = SKTGraphicHandleHalfWidth;
    if ([self isDrawingStroke]) {
    CGFloat strokeOutset = [self strokeWidth] / 2.
    if (strokeOutset>outset) {
        outset = strokeOutset;
```

```
0x0000ba70   <+0048>   stw     r3,168(r30)
0x0000ba74   <+0052>   stw     r4,172(r30)
0x0000ba78   <+0056>   stw     r5,176(r30)
0x0000ba7c   <+0060>   addis   r2,r31,2
0x0000ba80   <+0064>   lwz     r2,-11668(r2)
0x0000ba84   <+0068>   lfs     f0,0(r2)
0x0000ba88   <+0072>   stfs    f0,64(r30)
0x0000ba8c   <+0076>   lwz     r9,172(r30)
0x0000ba90   <+0080>   addis   r2,r31,2
```

GDB: Stopped at breakpoint 4 (hit count : 1) – '-drawingBounds – Line 415'

When the disassembly pane is shown, the Variable list contains a Registers group containing all the processor registers.

Debugging in the Console

Xcode's graphical interface for GDB, the GNU debugger, lets you perform most necessary debugging tasks. You may, however, encounter situations—such as working with watchpoints in GDB—that require you to interact directly with the debugger on the command line. Using the console window, you can:

- View the commands that Xcode sends to GDB or the Java command-line debugger
- Send commands directly to GDB or the Java command-line debugger
- View the debugger output for those commands
- See debugging messages printed to `stderr` by your program or by system frameworks
- Debug a command-line program that requires input from `stdin`

If you are debugging a command-line program that requires input from `stdin`, you must use the Console to communicate with your program when it is running in the debugger. This window is only available when your program is running under the debugger.

To open the Console window, choose Run > Console.

To enter commands, click in the console window and type at the `gdb` or JavaBug prompt. To get help with GDB and Java debugging commands, enter `help` at the console. To learn more about command-line debugging with GDB, see *Debugging with GDB* in Tools Compilers & Debuggers Documentation.

To make the Console text easily readable, Xcode lets you choose the text colors and fonts used in the console window. You can use different fonts and colors for the text you type in the console, the text the debugger writes to the console, and the debug console's prompt. To change the colors used for text in the console window, use the Fonts and Colors group in Xcode > Preferences > Debugging. See "Debugging Preferences" in *Xcode Workspace Guide* for more information.

Note that Xcode uses an executable environment to determine how to launch your program. To specify command-line arguments and environment variables to use when launching your program from Xcode, edit the executable environment; you cannot alter this environment from the `gdb` command-line in the console window. See "Configuring Executable Environments" in *Xcode Project Management Guide* to learn more about configuring an executable environment.

Managing Program Execution

Xcode provides several mechanisms to control and monitor the execution of an inferior (a program being debugged). While debugging a program you can:

- Pause execution as a result of an event. You may also perform actions such as logging and emitting sounds
- Monitor the value of variables or data items
- Pause execution through special Core Services debugging functions

This section describes the facilities Xcode provides to let you control and monitor the execution of your program.

Using Breakpoints

Breakpoints let you perform actions on certain events, such as when program execution reaches a certain point in the code. Table 6-1 lists the three types of breakpoints.

Table 6-1 Breakpoint types

Breakpoint type	Trigger
File line	Execution reaches a specific code line.
Symbolic	A specific routine is called.
C++ exception	A particular C++ exception is thrown.

You can add one or more *breakpoint actions* to a breakpoint. A **breakpoint action** is an operation that Xcode performs when the breakpoint is triggered. The default action (when the breakpoint has no breakpoint action associated to it) is to pause program execution. However, Xcode also supports:

Managing Breakpoints

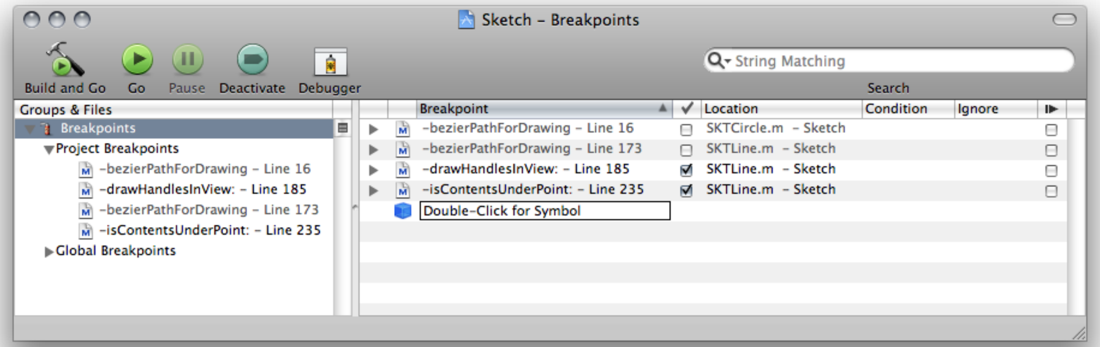
The following sections show how to view, add, activate/deactivate, customize, and remove breakpoints.

Viewing Breakpoints

The breakpoints window lets you view and modify breakpoints set in the current project, as well as any defined for the current user. To open the breakpoints window, choose Run > Show > Breakpoints.

Figure 6-1 shows the breakpoints window.

Figure 6-1 Breakpoints window



Breakpoint	✓	Location	Condition	Ignore	
-bezierPathForDrawing – Line 16	☐	SKTCircle.m – Sketch			☐
-bezierPathForDrawing – Line 173	☐	SKTLine.m – Sketch			☐
-drawHandlesInView: – Line 185	✓	SKTLine.m – Sketch			☐
-isContentsUnderPoint: – Line 235	✓	SKTLine.m – Sketch			☐
Double-Click for Symbol					

Breakpoints are collected in the Breakpoints smart group. This smart group contains two subgroups: the Project Breakpoints group contains breakpoints specific to the current project and the Global Breakpoints group contains breakpoints defined for the current user. Project breakpoints are stored in the project's user file; that is, they are saved for the current user and the current project. Global breakpoints are available to you in any project that you open.

You can select the Project Breakpoints group or the Global Breakpoints group to see only those breakpoints in the detail view; you can also open the group to see its contents in the outline view. Selecting the Breakpoints smart group displays all available breakpoints in the detail view.

Here is what the detail view for breakpoints contains:

- The disclosure triangle shows and hides any actions associated with the breakpoint.
- The icon indicates the type of the breakpoint. A file icon indicates that it is a file-line or C++ exception breakpoint. A blue cube indicates that it is a symbolic breakpoint.
- The Breakpoint column displays the title of the breakpoint. For file line breakpoints, this is the line number and surrounding context of the location at which the breakpoint is set. For symbolic breakpoints, this is the name of the function or method on which the breakpoint is set.
- The column with the checkmark indicates whether the breakpoint is turned on (selected) or off (unselected). When a breakpoint is turned on (and breakpoints are active), program execution stops when it reaches the specified line or routine call. Otherwise, program execution does not stop on it.

  A dash in this checkbox indicates that, while the breakpoint is turned on, the debugger has not yet resolved it. When you start debugging, these dashes change to checkmarks as the debugger resolves each breakpoint.
- The Location column shows the file in which file line breakpoints are located. For symbolic breakpoints, this column displays “Symbol.”
- The Condition column lets you specify an **execute condition** for the breakpoint. When you specify a condition, the breakpoing is triggered only if the condition (for example, `i == 24`) is true. You can use any variables that are in the current scope for that breakpoint. Note that you must cast any function calls to the appropriate return type.
- The Ignore column specifies an **ignore condition** for the breakpoint. Similar to an execute condition, when you specify an ignore condition, the breakpoint is triggered only if the condition is true.

- The column with the Continue symbol indicates whether to automatically continue execution of your program after hitting the breakpoint. If the checkbox in this column is selected, program execution stops on the breakpoint, Xcode performs any breakpoint actions, and then automatically resumes execution of the program. Otherwise, program execution stops on the breakpoint, Xcode performs any breakpoint actions, and does not resume until you give the command to continue.

You can delete, and turn off/on any existing breakpoint in your project in the breakpoints window. You can also create symbolic breakpoints. However, you can add a breakpoint to a specific line of code only from the editor or the console.

To view the source code for a breakpoint, double-click the breakpoint in the breakpoints window. This opens the source file in which the breakpoint is set (or the function is defined) in a separate editor window.

Adding File-Line Breakpoints

To add a file-line breakpoint, perform one of the following tasks:

- Select the line at which you want the breakpoint to be triggered and choose Run > Manage Breakpoints > Add Breakpoint at Current Line.
- Use the gutter shortcut (contextual) menu. See "Gutter Shortcut Menu" (page 12) for details.

You can easily move a file-line breakpoint from one code line to another line by dragging the breakpoint arrow to the new line within the code editor.

Adding Symbolic Breakpoints

To add a symbolic breakpoint, perform one of the following tasks:

- Select the trigger code line and choose Run > Manage Breakpoints > Add Symbolic Breakpoint.

  In the dialog that appears, enter the name of the method or function at which you want the breakpoint to be triggered.
- Open the breakpoints window, double-click on "Double-Click for Symbol," and enter the trigger routine name.

Note: When you set a breakpoint on an Objective-C method, you must include the brackets and a plus or minus sign. For example, to stop whenever a Cocoa exception is raised, enter `-[NSException raise]`.

Adding C++ Exception Breakpoints

To add a C++ exception breakpoint:

Select the trigger code line and choose Run > Manage Breakpoints > Add C++ Exception Breakpoint.

In the dialog that appears, enter the name of the trigger exception.

To stop on all C++ exceptions, select All Exceptions.

Importing and Exporting Breakpoints

Xcode allows you to import and export breakpoints. This feature lets you share sets of breakpoints with other developers working on the same project, or across projects.

To export breakpoints, select the breakpoints you want to export from the Breakpoints smart group in the Groups & Files list and choose Run > Manage Breakpoints > Export Breakpoints.

To import breakpoints into a project, open the Breakpoints smart group and select the group to which you want to add the breakpoints—either Project Breakpoints or Global Breakpoints and choose:

Xcode menu bar: Run > Manage Breakpoints > Import Breakpoints

Xcode adds a new subgroup containing the imported breakpoints to the selected breakpoints group.

Grouping Breakpoints

To group existing breakpoints, select them in the Breakpoints smart group and choose:

Groups & Files shortcut menu: Group

You can create your own groups within the Breakpoints smart group to organize breakpoints. To add a breakpoints group:

1. Select either the Project Breakpoints or the Global Breakpoints group, depending on which breakpoint realm you want to operate on.
2. Choose Project > New Group.

To rename an existing group, Control-click the group and choose Rename.

To delete a group, select it and press Delete.

Removing Breakpoints

You can remove breakpoints using the text editor of the breakpoints window.

To remove a fine-line breakpoint, open the breakpoint's file in the text editor and drag the breakpoint indicator out of the gutter. You may also select the code line with the breakpoint and choose Run > Manage Breakpoints > Remove Breakpoint at Current Line.

To remove symbolic and C++ exception breakpoints, open the breakpoints window, select the breakpoint, and press Delete.

Turning Breakpoints On/Off

In the course of debugging a program, you may find that you don't currently want the debugger to use a particular breakpoint that you have set, but you don't want to delete it entirely, either, in case you want to use it again at a later time. Instead of deleting the breakpoint from your project, you can simply turn it off. Breakpoints that are turned off are not triggered during program execution.

You can turn on/off any breakpoint in your project in the breakpoints window. To turn on/off a breakpoint, open the breakpoints window and click the checkbox beside the breakpoint you want to modify. A breakpoint is turned on when this checkbox is selected.

You can also turn on/off file-line breakpoints from the text editor by clicking the breakpoint in the gutter. Breakpoints that are turned off appear as a light-gray arrow in the gutter.

You can turn on/off more than one breakpoint—or even groups of breakpoints—at once. To do so:

1. Select the breakpoints you want to turn on/off in the Breakpoints smart group in the Groups & Files list. You can select the individual breakpoints or a breakpoint group.

2. Choose Enable Breakpoints or Disable Breakpoints from the gutter shortcut menu to turn them on or off, respectively.

To quickly turn on a set of breakpoints, disabling any other breakpoints for the project, select the breakpoints you want to turn on and choose Enable Only These Breakpoints from the gutter Option shortcut menu (Control-Option-click). Turning on/off a breakpoints group affects all breakpoints in that group and in all its subgroups.

Using Breakpoint Templates

Xcode provides a number of breakpoint templates that you can use to insert file line breakpoints that are preconfigured with various breakpoint actions. To add a breakpoint based on one of these templates, use:

Text editor gutter shortcut menu: Built-in Breakpoints

These are the breakpoint templates:

- **Log breakpoint with arguments and auto-continue.** This creates a breakpoint that prints the arguments of the currently executing function or method to the console and automatically continues.
- **Log breakpoint and hit count and auto-continue.** This creates a breakpoint that prints the number of times the breakpoint has been hit during the current debugging session to the console and automatically continues.
- **Log stack trace and auto-continue.** This creates a breakpoint that prints a backtrace of the entire stack for the current thread and automatically continues. The Debugger Command action uses the GDB command `bt` to generate the backtrace.
- **Sound out and auto-continue.** This creates a breakpoint that uses the Sound action to play an alert and automatically continues.
- **Print self and auto-continue.** This creates a breakpoint that prints a description of an Objective-C object and automatically continues.

- **Speak breakpoint and hit count and auto-continue.** This creates a breakpoint that uses the Log action to speak the breakpoint name and the number of times it has been encountered during the current debugging session and then automatically continues.

When you choose one of these templates, Xcode inserts a breakpoint at the current line of code and configures it with the appropriate action. You can edit the breakpoint action in the breakpoints window.

You can create your own breakpoint templates from any breakpoints that you have configured by placing them in a group called Template Breakpoints. Any breakpoints that you add to this group will appear in the Built-in Breakpoints gutter shortcut menu.

Conditionalizing Breakpoints

To have the debugger execute breakpoint actions only under certain circumstances, you can associate an execute condition or an ignore condition with a file-line or symbolic breakpoint. See "Viewing Breakpoints" (page 27) to learn more about these conditions.

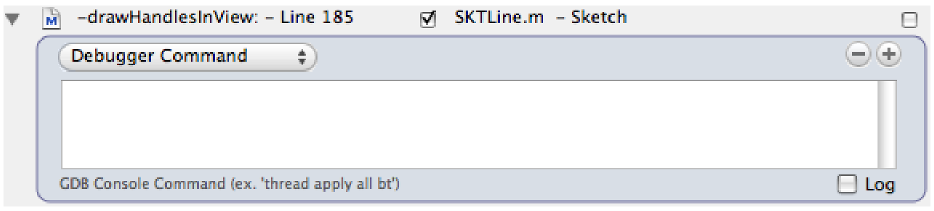
Each time it encounters the breakpoint, the debugger evaluates the expression. If the expression is true—that is, if it evaluates to a nonzero value—the debugger stops at the breakpoint and Xcode performs any breakpoint actions. Breakpoint conditions are currently supported only when using GDB.

Creating Breakpoint Actions

As described in "Using Breakpoints" (page 27), by default, breakpoints pause the execution of your program. However, you can have breakpoints perform other actions when they are triggered. For example, you can have a breakpoint log additional information to the console, execute a debugger or shell command, or alert you by playing a sound.

To associate a breakpoint action with a breakpoint, open the breakpoint in the breakpoints window. This displays the interface for specifying breakpoint actions, as shown in Figure 6-2.

Figure 6-2 Breakpoint action editor


-drawHandlesInView: - Line 185
SKTLine.m - Sketch
Debugger Command
GDB Console Command (ex. 'thread apply all bt')
Log

To add an action, click the plus (+) button. Choose the type of action you want Xcode to perform from the actions pop-up menu. Each action has a different interface for specifying the message to log, the sound to play, and so forth.

You can choose any of the following actions:

- **Debugger Command.** Xcode sends a command to the debugger. Type the command in the text field below the action menu. For example, if you are using GDB, you could specify a command such as `backtrace`.

Do not use the GDB commands `jump` or `continue` in a breakpoint action if you have set that breakpoint to automatically continue. If the first action associated with a breakpoint is a Debugger Command action, Xcode sets the command on the breakpoint using the GDB command `commands`, instead of pausing execution and then sending the command to GDB.

- **Log.** Xcode logs a message when it encounters the breakpoint. You can choose to have Xcode print a message to the console, speak the message using Text-to-Speech, or do both. Choose either Log or Speak to specify how Xcode logs the message.

  To specify the message that Xcode logs, type it into the text field below the action menu. You can include information about the current breakpoint in the message using the following formatters:

  - `%B`. The name of the breakpoint, as shown in the Breakpoint column of the Breakpoints window. By default, this is the name of the function or method in which the breakpoint is set, including the line number for file-line breakpoints. You can also specify your own name for the breakpoint, as described in "Viewing Breakpoints" (page 27).
  - `%H`. The number of times the debugger has encountered the breakpoint in the current debugging session.
  - `%C`. Any comment associated with the breakpoint, as shown in the Comments column of the Breakpoints window.

  For example, if you specify the message `Stopped at %B again. We've been here %H times. Don't you want to stop somewhere else instead?`, you would see a message similar to the following in the console:

  ```
  Stopped at TDialView::Draw() - Line 305 again. We've been here 5 times. Don't
  you want to stop somewhere else instead?
  ```

- **Sound.** Xcode plays a sound when it encounters the breakpoint. Use the second pop-up menu to specify which sound to play. You can choose any of the built-in system sounds.
- **Shell Command.** Xcode executes a command in your default shell when it encounters the breakpoint. Enter the name of the command or shell script file in the first text field. You can also click Choose to navigate to it in the Open dialog. Type any arguments to pass to the command in the second text field. Xcode displays a warning in the project window status bar when it cannot find the command or shell script file entered in the first text field.

  If you have set the breakpoint to automatically continue, Xcode does not, by default, wait until the shell command has finished executing before resuming execution of the program being debugged. To ensure that Xcode does not continue until the specified command has finished executing, select the "Wait until done" option.

  Xcode prints the exit value the command or shell script in the debugger console, whether "Wait until done" is selected.

- **AppleScript.** Xcode executes an AppleScript script when the breakpoint is encountered. Enter the script in the text field below the action menu. To compile your script, click the Compile button.

  You can use the same format specifiers in your AppleScript script to access information about the current breakpoint as for the Log action, described above.

Xcode supports these breakpoint actions for GDB, the Java debugger, and the AppleScript debugger. Note, however, that the debugger commands that you can pass to the Java and AppleScript debuggers using a debugger command action are limited.

You can use the value of GDB expressions in some breakpoint actions. These expressions can return the value of a local variable or the result of a function call. The breakpoint actions on which you can use GDB expressions are: Log, Shell Command, and AppleScript. To execute a GDB expression in a breakpoint action, surround the expression with `@` characters. For example, to print the value of the local variable `i`, use a Log action with the text `i = @i@`. To print the name of the running program, use a Log action with the text `Program name is @(char*)getprogname()@`.

Using Watchpoints

To monitor changes to the value of variables or data items, you can set *watchpoints*. A **watchpoint** pauses execution of the program whenever the value of the watched item changes. You can set a watchpoint on a variable only when execution of the program is halted. To set a watchpoint on a variable:

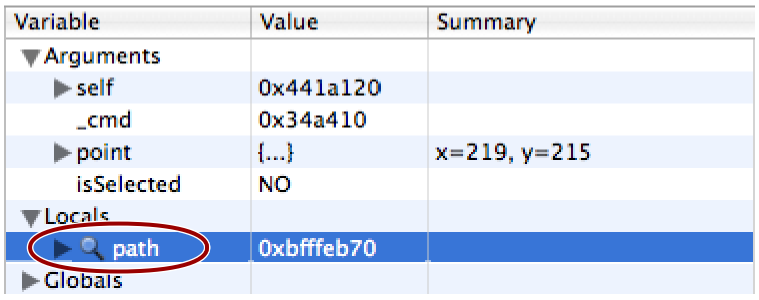
1. With execution of the program paused at a breakpoint, select the variable in the Variable list in the Debugger window. See "Viewing Variables in the Debugger window" (page 22) to learn more about the Variable list.

2. Choose one of the following:

   - Run > Variables View > Watch Variable
   - Watch Variable from the variable list shortcut menu

   Xcode displays a magnifying glass next to the variable to indicate that the variable is being watched, as shown in Figure 6-3.

Figure 6-3 Watched variable in the Variable list



Variable	Value	Summary
▼Arguments		
▶self	0x441a120	
_cmd	0x34a410	
▶point	{...}	x=219, y=215
isSelected	NO	
▼Locals		
▶ 🔍 path	0xbfffeb70	
▶Globals		

When the value of the variable changes, Xcode pauses execution of the program and displays a dialog showing the location of the program counter and the new value of the variable. If execution of the program moves beyond the scope of the current variable, Xcode deletes the watchpoint and pauses execution of the program.

Important: Watching local variables, located on the stack, can cause your program to crash if system calls are made in the current function.

Pausing on Core Services Debugging Functions

The Core Services framework includes functions, such as `Debugger` and `DebugStr`, that break into the debugger with a message. If your code contains calls to these functions, you can tell the Xcode debugger to stop when it encounters them.

You can turn on this feature for an individual executable, as described in "Executable-Environment Debugging Information" in *Xcode Project Management Guide*, or for all executables in the current project. To toggle this feature on/off, choose Run > Stop on Debugger()/DebugStr().

Note: Xcode implements this feature by setting the `USERBREAK` environment variable to `1`, which causes these functions to send a `SIGINT` signal to the current process, breaking into the debugger.

Viewing Variables and Memory

This chapter describes the various ways in which you can view the values of variables as you debug your programs.

Setting the Variable Display Format

You can view the value of a variable in a variety of formats, including hexadecimal, octal, and unsigned decimal. To display the value of a variable in a different numeric format:

1. Open the Debugger window and select the variable in the Variable list.
2. From Run > Variables View.

   choose one of these options:

   - Natural
   - Hexadecimal
   - OSType
   - Decimal
   - Unsigned Decimal
   - Octal
   - Binary

You can also cast a variable to a type that's not included in the menu. For example, a variable may be declared as `void *`, yet you know it contains a `char *` value. To cast a variable to a type, select the variable, choose Run > Variables View View Value As.

Using Data Formatters

Xcode allows you to customize how variables are displayed in debugger datatips and the Variable list in the debugger by specifying your own format strings for the Value or Summary columns. In this way, you can display program data in a readable format. Xcode includes a number of built-in **data formatters** for data types defined by various Mac OS X system frameworks. You can edit these format strings or create your own data formatters.

To turn on/off data formatters, use:

Xcode menu bar: Run > Variables View > Enable Data Formatters

Important: If you are debugging heavily threaded code, where more than one thread is executing the same code, data formatters may cause threads to run at the wrong time and miss breakpoints. To avoid this problem, disable data formatters.

Writing Data Formatters

You can provide your own data formatters to display data types defined by your program. To edit the formatter associated with a variable value or variable summary, select the variable in the Variable list in the debugger and double-click in the appropriate column of the Variable list in the debugger. You can also choose Run > Variables View > Edit Summary Format.

Data formatters can contain:

- Literal text.

  You can add explanatory text or labels to identify the data presented by the format string.

- References to values within a structured data type.

  You can access any member of a structured variable from the format string for that variable. The syntax for doing so is `%<pathToValue>%`, where `<pathToValue>` is the period-delimited path to the value you want to access in the current data structure.

  In C++, to access a member defined in the superclass of an object, the path to the member must include the name of the superclass. For example, `%Superclass.x%`.

- Expressions, including function or method calls.

  The syntax for an expression is `{<expression>}`. To reference the variable itself in `<expression>`, use `$VAR`. For example, to display the name of a notification—of type `NSNotification`—you can use `{(NSString *)[$VAR name]}`.

- References to columns in the Variable list.

  When Xcode resolves a data formatter, it replaces the member reference or expression with the value that was obtained by evaluating the reference or expression. This value is the same value found in the Value column of the Variable list. You can, however, specify that Xcode use the contents of any column in the Variable list—Variable name, Type, Value, or Summary. To do so, add `:<referencedColumn>` after the expression or member reference, where `<referencedColumn>` is a letter indicating which Variable-list column to access. So, the syntax for accessing a value in a structured data type becomes `%<pathToValue>%:<referencedColumn>`. Table 7-1 shows the possible values for referencing variable display columns.

  Table 7-1 Characters for referencing variable display columns

Reference	Variable view column
`n`	Variable (shows the variable name)
`v`	Value
`t`	Type

Reference	Variable view column
`s`	Summary

Note: Double quotation-mark characters in data formatters must be escaped, as in the following example:

```
{(NSString *)[$VAR valueForKey:@\"name\"]}:s
```

Memory management in data formatters: Xcode automatically allocates and deallocates the memory data formatters use. But if a data formatter returns a reference to data that resides elsewhere, the formatter would be responsible for managing that memory.

However, there's no mechanism to notify data formatters when they are no longer used, therefore, data formatters should not return dynamically allocated memory. They can, however, return static strings; for example, `"invalid value"`.

C++ data formatters: Your data formatters may take C++ pointers but not C++ references. With references, copies of the referenced objects are generated, which may cause undesired side effects.

Data Formatter Example

The following example uses the `CGRect` data type to illustrate how you can build format strings using member references and expressions. (Note that because Apple provides format strings for the `CGRect` data type, Xcode already knows how to display the contents of variables of that type). The `CGRect` data type is defined as follows:

```
struct CGRect { CGPoint origin; CGSize size; }; typedef struct CGRect CGRect;
```

Assuming that the goal is to create a format string that displays the origin and size of variables of type `CGRect`, there are many ways you can write such a format string. For example, you can reference members of the `origin` and `size` fields directly. Of course, each of these two fields also contains data structures, so simply referencing the values of those fields isn't very interesting; the values you want are in the data structures themselves. One way you can access those values is to include the full path to the desired field from the `CGRect` type. For example, to access the height and width of the rectangle, in the `height` and `width` fields of the `CGSize` structure in the `size` field you could use the references `%size.height%` and `%size.width%`. A sample format string using these references might be similar to the following:

```
height = %size.height%, width = %size.width%
```

You could write a similar reference to access the x-and y-coordinates of the origin. Or, if you already have a data formatter for values of type `CGPoint` that displays the x and y coordinates of the point in the Summary column of the Variable list—such as `(%x%, %y%)`—you can leverage that format string to display the contents of the `origin` field in the data formatter for the `CGRect` type. You can do so by referencing the Summary column for `CGPoint`, as in the following format string:

```
origin: %origin%:s
```

When Xcode evaluates this format string, it accesses the `origin` field and retrieves the contents of the Summary column for the `CGPoint` data type, substituting it for the reference to the `origin` field. The end result is equivalent to writing the format string origin: `(%origin.x%, %origin.y%)`.

You can combine this format string with the format string for the size field and create a data format string for the `CGRect` type similar to the following:

```
origin: %origin%:s, height = %size.height%, width = %size.width%
```

For example, a rectangle with the origin (1,2), a width of 3, and a height of 4 results in the following display:

```
origin: (1, 2), width=3, height=4.
```

You can also write a data formatter to display the same information using an expression such as the following:

```
origin: {$VAR.origin}:s, height = {$VAR.size.height}, width = {$VAR.size.width}
```

When Xcode evaluates this expression for a variable, it replaces `$VAR` with a reference to the variable itself. Of course, using an expression to perform a simple value reference is not necessary. Another example of an expression in a format string is `{(NSString *)[$VAR name]}:s` to display the name of a notification, of type `NSNotification`.

When you specify a custom data formatter for a variable of a given type, that format string is also used for all other variables of the same type. Note, however, that you cannot specify a custom format for string types, such as `NSString`, `char*`, and so on. Custom data formatters that you enter in the Debugger window are stored at:

```
~/Library/Application
Support/Developer/<Xcode_release>/CustomDataViews/CustomDataViews.plist
```

In addition to supplying custom format strings to display variables in the debugger, you can also write code that constructs descriptions for variables displayed in the debugger. These functions can be packaged as a bundle that is loaded into the process being debugged. These functions can then be invoked from data formatters.

Monitoring the Value of an Expression

Using the Expressions window, you can view and track the value of an expression. For example, you can track a global value or a function result over the course of a debugging session. To open the Expressions window, choose Run > Show > Expressions.

Type the expression you wish to track in the Expression field. Xcode adds the expression, evaluates it, and displays the value and summary for that expression. The display format of the value and summary information is determined by any data formatters in effect.

The expression can include any variables that are in scope at the current statement and can use any function in your project. To view processor registers, enter an expression such as `'$r0', '$r1'`.

In the debugger you can add a variable to the Expressions window by selecting the variable in the Variable list and choosing Run > Variables View > View Variable As Expression.

To remove an expression from the Expressions window, select it and press Delete.

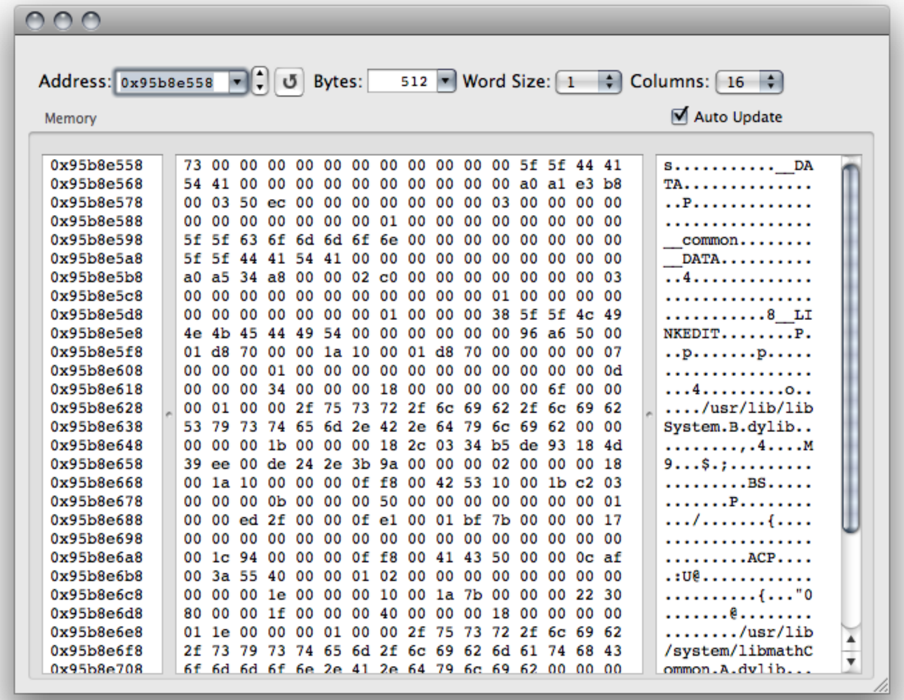
Tips on using the Expressions window:

- The expression is evaluated in the current frame. As the frame changes, the expression may go in and out of scope. When an expression goes out of scope, Xcode notes this in the Summary column.
- Always cast a function to its proper return type. The debugger doesn't know the return type for many functions. For example, use `(int)getpid()` instead of `getpid()`.
- Expressions and functions can have side effects. For example, `i++` increments `i` each time it's evaluated. If you step though 10 lines of code, `i` is incremented 10 times.

Browsing Memory

When execution of the current program is paused, you can browse the contents of memory using the Memory Browser. To open the memory browser, shown in Figure 7-1, choose Run > Show > Memory Browser.

Figure 7-1 Memory Browser



Address: 0x95b8e558 Bytes: 512 Word Size: 1 Columns: 16
Memory
☑ Auto Update

```
0x95b8e558  73 00 00 00 00 00 00 00 00 00 00 00 5f 5f 44 41  s...........__DA
0x95b8e568  54 41 00 00 00 00 00 00 00 00 00 00 a0 a1 e3 b8  TA..............
0x95b8e578  00 03 50 ec 00 00 00 00 00 00 00 03 00 00 00 00  ..P.............
0x95b8e588  00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00  ................
0x95b8e598  5f 5f 63 6f 6d 6d 6f 6e 00 00 00 00 00 00 00 00  __common........
0x95b8e5a8  5f 5f 44 41 54 41 00 00 00 00 00 00 00 00 00 00  __DATA..........
0x95b8e5b8  a0 a5 34 a8 00 00 02 c0 00 00 00 00 00 00 00 03  ..4.............
0x95b8e5c8  00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00  ................
0x95b8e5d8  00 00 00 00 00 00 00 01 00 00 00 38 5f 5f 4c 49  ...........8__LI
0x95b8e5e8  4e 4b 45 44 49 54 00 00 00 00 00 00 96 a6 50 00  NKEDIT........P.
0x95b8e5f8  01 d8 70 00 00 1a 10 00 01 d8 70 00 00 00 00 07  ..p.......p.....
0x95b8e608  00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 0d  ................
0x95b8e618  00 00 00 34 00 00 00 18 00 00 00 00 00 6f 00 00  ...4.........o..
0x95b8e628  00 01 00 00 2f 75 73 72 2f 6c 69 62 2f 6c 69 62  ..../usr/lib/lib
0x95b8e638  53 79 73 74 65 6d 2e 42 2e 64 79 6c 69 62 00 00  System.B.dylib..
0x95b8e648  00 00 00 1b 00 00 00 18 2c 03 34 b5 de 93 18 4d  ........,.4....M
0x95b8e658  39 ee 00 de 24 2e 3b 9a 00 00 00 02 00 00 00 18  9...$.;.........
0x95b8e668  00 1a 10 00 00 00 0f f8 00 42 53 10 00 1b c2 03  .........BS.....
0x95b8e678  00 00 00 0b 00 00 00 50 00 00 00 00 00 00 00 01  .......P........
0x95b8e688  00 00 ed 2f 00 00 0f e1 00 01 bf 7b 00 00 00 17  .../.......{....
0x95b8e698  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x95b8e6a8  00 1c 94 00 00 00 0f f8 00 41 43 50 00 00 0c af  .........ACP....
0x95b8e6b8  00 3a 55 40 00 00 01 02 00 00 00 00 00 00 00 00  .:U@............
0x95b8e6c8  00 00 00 1e 00 00 00 10 00 1a 7b 00 00 00 22 30  ..........{..."0
0x95b8e6d8  80 00 00 1f 00 00 00 40 00 00 00 18 00 00 00 00  .......@........
0x95b8e6e8  01 1e 00 00 00 01 00 00 2f 75 73 72 2f 6c 69 62  ......../usr/lib
0x95b8e6f8  2f 73 79 73 74 65 6d 2f 6c 69 62 6d 61 74 68 43  /system/libmathC
0x95b8e708  6f 6d 6d 6f 6e 2e 41 2e 64 79 6c 69 62 00 00 00  ommon.A.dylib...
```

You can also open the Memory Browser to the location of a particular variable:

1. In the debugger, select the variable.
2. Choose Run > Variables View > View As Memory.

In the Memory Browser, you can see:

- A contiguous block of memory addresses.
- The contents of memory at those addresses, represented in hexadecimal.
- The contents of memory at those addresses, represented in ASCII.
- The Address field, which controls the starting address of the memory displayed in the table below. You can enter an address, a variable name or expression. Hexadecimal values must be preceded by `0x`.
- The arrows next to the Address field, which "step" through memory; clicking the up arrow shows the previous page of memory, and clicking the down arrow returns the next page of memory.
- The Bytes field, which controls the number of bytes displayed in the memory browser. You can choose one of the options from the pop-up menu in the Bytes field or type a number directly into the field. However, the number of bytes will be rounded up to the next full row of memory fetched.
- The Word Size pop-up menu specifies the size of the memory chunks you want to view. For example, to analyze memory you know contains 32-bit integers, you would choose 4 from the menu.

  Note: On a big-endian system, the word size that you choose affects the spacing of the data displayed by the Memory Browser. On a little-endian system, however, the word size can also affect the byte order as well. If you are debugging on Intel-based computers, you should be careful to choose an appropriate word size for viewing memory. In most cases, the appropriate word size is one.

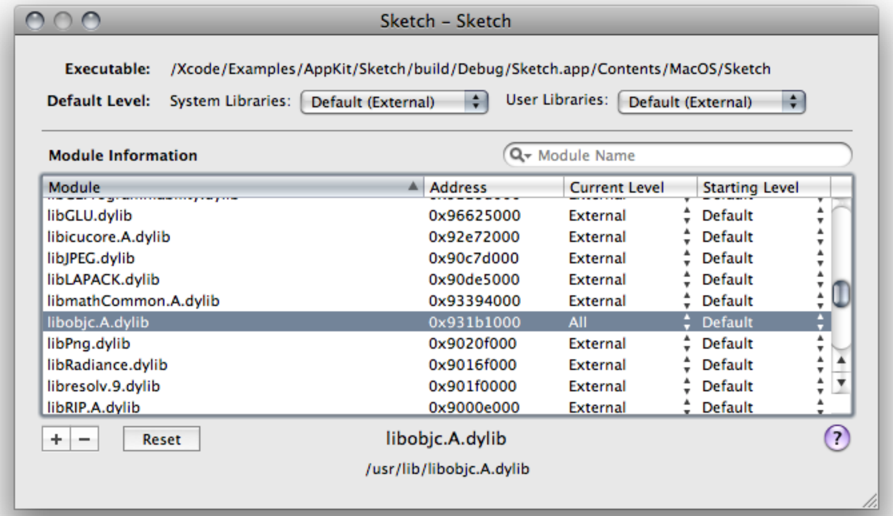
- The Columns pop-up menu specifies the number of columns the browser uses to display the identified memory.

Viewing Shared Libraries

You can see which libraries have been loaded by the inferior using the Shared Libraries window, shown in Figure 7-2. To open this window, choose Run > Show > Shared Libraries.

The Module Information table lists all the individual libraries the executable links against. In this table, you can see the name and address of each shared library, as well as the symbols the debugger has loaded for that library. The Starting Level column shows which symbols the debugger loads by default for a given library when the current executable is running. The Current Level column shows which symbols the debugger has loaded for the library during the current debugging session. When an entry has a value in the Address and Current Level columns, the library has been loaded in the debugging session.

Figure 7-2 Shared Libraries window



Sketch – Sketch

Executable: /Xcode/Examples/AppKit/Sketch/build/Debug/Sketch.app/Contents/MacOS/Sketch

Default Level: System Libraries: Default (External) User Libraries: Default (External)

Module Information

Module	Address	Current Level	Starting Level
libGLU.dylib	0x96625000	External	Default
libicucore.A.dylib	0x92e72000	External	Default
libJPEG.dylib	0x90c7d000	External	Default
libLAPACK.dylib	0x90de5000	External	Default
libmathCommon.A.dylib	0x93394000	External	Default
libobjc.A.dylib	0x931b1000	All	Default
libPng.dylib	0x9020f000	External	Default
libRadiance.dylib	0x9016f000	External	Default
libresolv.9.dylib	0x901f0000	External	Default
libRIP.A.dylib	0x9000e000	External	Default

libobjc.A.dylib

/usr/lib/libobjc.A.dylib

The path at the bottom of the window shows where the currently selected library is located in the file system. You can quickly locate a particular library by using the search field to filter the list of libraries by name.

Using the Shared Libraries window you can also choose which symbols the debugger loads for a shared library. This can help the debugger load your project faster. You can specify a default symbol level for all system and user libraries; you can also change which symbols the debugger loads for individual libraries.

For any shared library, you can choose one of three levels of debugging information:

- **All:** Loads all debugging information, including all symbol names and the line numbers for your source code.
- **External:** Loads only the names of the symbols declared external.
- **None:** Loads no information.

You can specify a different symbol level for system libraries and user libraries. User libraries are any libraries produced by a target in the current project. System libraries are all other libraries.

By default, the debugger loads only external symbols for system and user libraries, and automatically loads additional symbols as needed. Turning off the "Load symbols lazily" option, described in "Debugging Preferences" in *Xcode Workspace Guide*, changes the default symbol level for User Libraries to All. This is a per-user setting and affects all executables you define. You can also customize the default symbol level settings for system and user libraries on a per-executable basis, using the Default Level pop-up menus in the Shared Libraries window.

For some special cases—applications with a large number of symbols—you may wish to customize the default symbol level for individual libraries when running with a particular executable. To set the initial symbol level to a value other than the default, make a selection in the Starting Level column. While debugging, you can increase the symbol level using the Current Level column. This can be useful if you need more symbol information while using GDB commands in the console. Clicking Reset sets all of the starting symbol levels for the libraries in the Module Information table to the default value.

Modifying Running Code

A very powerful feature of Xcode is the ability to modify your executable while it is running and see the results of your modification. This feature is implemented through the Fix command.

This feature is useful when it takes a significant amount of time to reach the place in your application's execution cycle that you want to debug. You can use the feature to learn more about potential bug fixes or to see the immediate results of code changes.

Fix Command Overview

The Fix command in Xcode is a way to modify an application at debug time and see the results of your modification without restarting your debugging session. This feature can be particularly useful if it takes a lot of time to reach your application's current state of execution in the debugger. Rather than recompile your project and restart your debugging session, you can make minor changes to your code, patch your executable, and see the immediate results of your changes.

The Fix command is not intended as a replacement for building your product regularly. Instead, it is a convenience feature for viewing the effect of small changes without restarting your debugging session.

Important: Use of the Fix command is subject to certain requirements and restrictions, which are listed in "Fix Command Restrictions" (page 47).

Supported Fixes

Although there are many restrictions to what you can fix, there are also some features that are explicitly supported by the Fix command, including the following:

- Storing pointers to a patched function. GDB inserts code to jump from the old function to the newly patched function. Thus, despite your code having a pointer to the original function, using that pointer executes the patched version.
- You can add file-local static variables to a file, but remember that GDB does not execute any initialization code associated with them. Thus, if you declare a new class instance as a global static, GDB does not execute any constructors or static initializers for the instance. there are also limitations on how those variables behave after multiple patches.
- You can add new C++ classes as part of a patch.

To learn about restrictions on the Fix command, see "Fix Command Restrictions" (page 47).

GDB and the Fix Command

The process of fixing source files while debugging is tricky. GDB must manipulate your executable while it is running, and insert new code without disturbing the state of your program execution. The actual process involves compiling your code with special flags and rewriting portions of your binary to call the new code when appropriate. At all times, the Fix command modifies the in-memory image of your executable. It does not permanently modify the files of your original application binary.

When it receives a patched binary, GDB compares that binary against the code in the application's original binary. GDB checks for several modifications that cannot be patched into a running application. If it detects any of these modifications, it reports back to Xcode that it could not incorporate the patch. If this occurs, you can stop debugging and rebuild your application, or continue debugging without the patch.

If GDB does not report any problems with your patch, it integrates the patched code into your application's memory image. It does this by making the following modifications:

- Modifies functions in your original binary to jump to any patched versions
- Modifies any static or global variables in the patched file to point back to the versions in the original binary

Debugging with Patched Code

After patching your application, you should be able to continue debugging your code as before. The next time you encounter a patched function, you should see the changes you made appear in the debugger. However, there are some caveats to be aware of when working with patched code.

If you patch a function that is on the stack, you may not see the results of that patch immediately. GDB is capable of patching the function that is currently at the top of the stack. However, if you patch a function that is further down the calling chain, the patch does not take effect until the next time you call it. Thus, the function must return and be called again before it receives the patch. Until that time, the function on the stack continues to execute the original code.

While your program is paused in the debugger, you can move the program counter around and resume execution from any point in the current function. This feature lets you rerun a patched function from the beginning, or from any point, to account for the changes you made.

If you quit your debugging session for any reason, you must rebuild your program to acquire any changes made by patching. The effects of the Fix command are only applicable to your executable while it is active in the debugger. The reason is that the command does not modify your program's compiled object files. Instead, it creates temporary object files and loads them into the memory space of your process dynamically. When you quit the debugger, GDB discards the temporary object files containing the patches. Recompiling your project recreates your application's original object files from the patched code.

Using Fix and Continue

If you are running your executable in the debugger and you make changes to your source code, you can patch your executable by choosing Run > Fix.

If you change more than one source file, you must fix each file separately. Xcode compiles the changes, patches the executable to use the new code, and resumes execution from the location at which the program was paused. If the changes to your code appear before the line at which execution is set to resume, you will not see the effect of your change until that code is called again. To see the effect of your change, you can manually alter the location at which execution resumes: When your program is paused, drag the PC indicator—the red arrow pointing to the line of code where execution is paused—to the location at which you want to resume execution.

Xcode automatically locates the correct target to use when creating the fix bundle. For example, if you are debugging an application suite that relies on a framework you created, you can make a change in the framework code. When you execute the Fix command, Xcode automatically uses the correct framework target, instead of the target associated with the running executable, to create the fix. Xcode will also follow cross-project references to targets in other projects.

Fix Command Restrictions

The Fix command in Xcode is a powerful way to make small changes to a source file without restarting your debugging session. Although powerful, there are some things you need to do before you can take advantage of the Fix command. The command itself is enabled only when you are debugging an executable. In addition, you must make sure you build your program as follows:

- Build using native targets
- Compile your code with GCC version 3.3 or later
- Build your code without optimizations
- Build your code with debugging symbols enabled

In general, if you build using the Debug build configurations provided by Xcode, the correct settings are used. However, if you try to patch your executable and get a file-not-found error, check the build settings of your target and make sure that debugging symbols are turned on, as described in "Building for Debugging" in *Xcode Project Management Guide*. You can also check the build log to make sure that the patch bundle was created.

The Fix command works on only one file at a time. If you make changes to multiple source files, you must patch each file separately before continuing with your debugging session.

Important: If you attempt to patch multiple files, pause your application until you finish integrating all of the patches.

Fix Command Restrictions Reported by GDB

There are many types of code changes that cannot be patched. The GDB debugger reports an error if it cannot integrate any of your changes because of a known restriction. If GDB reports one of these errors, you must rebuild your program and restart your debugging session, or continue debugging the program without the changes.

GDB recognizes the following changes to your code and reports an error if you try to include them as part of a fix:

- Changes to the number or type of arguments in a function or method that is currently on the stack
- Changes to the return type or name of a function or method that is currently on the stack
- Changes to the number or type of local variables in a function or method that is currently on the stack
- Changes to the type of global or file-static variables
- Symbol type redefinitions, that is, changing a function to a variable or a variable to a function

Additional Fix Command Restrictions

In addition to the restrictions reported by GDB, there are additional restrictions that GDB currently does not check. If you attempt to include any of these changes in a patch, your application may crash or exhibit other undefined behavior when it encounters the code. The solution is to avoid using the Fix command for the change. Instead, rebuild your program and restart your debugging session.

The following is a list of changes that cannot be included as part of a fix:

- Changes to a nib file.
- Changes to the definition of a structure or union.
- The addition of an Objective-C class.
- The addition or removal of class instance variables.
- The addition or removal of class methods.
- The addition or removal of methods to an Objective-C category. These methods are not registered with the Objective-C runtime and thus cannot be called. (You can fix existing methods in a category.)
- Any reference to an unresolved external variable or function. Link errors of this nature cannot be resolved by the dynamic linker.
- The addition of a static variable across multiple patches in one session. GDB maintains a copy of the static in each patch; however, because there is no original variable to refer to, each variable remains separate from the others, which can lead to unpredictable results.
- The addition of a function to one file when it is called from a different patched file. New functions are private to the patch file in which they appear.
- The addition of a `try` block or the addition of a `catch` handler to an existing `try` block.
- The addition of a C++ template class specialization.
- Changes to functions that require two-level namespaces during linking to prevent known symbol conflicts across different libraries. GDB supports patching two-level namespace binaries but currently does so using flat namespace conventions.

Important: Be aware that other conditions may also cause patched code to fail or exhibit other undefined behavior. If you encounter such a problem, you should rebuild your program and start a new debugging session.

Debugging Programs Remotely

Remote debugging lets you debug a program running on another computer. This is good for programs that you cannot easily debug on the host on which they are running. For example, you may be trying to debug a full-screen application, such as a game, or a problem with event handling in your application's GUI. Interacting with the debugger on the same computer interferes with the execution of the program you are trying to debug. In these cases, you have to debug the program remotely.

With remote graphical debugging in Xcode, you can debug a program running on a remote computer, as you would any local executable, without resorting to the command-line.

Note: Standard input (`stdin`) does not work with remote debugging; if you have a command-line tool that requires user input, you must use the GDB command-line interface to debug your program remotely.

This chapter introduces remote debugging in Xcode and walks you through enabling remote debugging in Xcode. To set up your project for remote debugging, you must perform the steps described in the following sections:

- "Configuring Remote Log-in" (page 49) describes how to configure your local computer and the remote host to allow remote login using SSH public key authentication.
- "Creating a Shared Build Location" (page 51) describes how to set up a shared build directory that both computers can access via the same path.
- "Configuring an Executable for Remote Debugging" (page 51) describes how to configure the executable of the program you wish to debug for remote debugging in Xcode.

Configuring Remote Log-in

Remote debugging in Xcode relies on SSH public key authentication to create a secure connection with the remote computer. To facilitate authentication, Xcode integrates with `ssh-agent`. This lets you use encrypted private keys for added security without having to reenter your passphrase each time Xcode establishes a connection to the remote host. If you already use a third-party utility to set up the environment variables used by `ssh-agent`, Xcode attempts to use those settings. Otherwise, Xcode uses its own agent for authentication.

Before starting a remote debugging session, you need to be able to log in to the remote computer. To do this, you must:

1. Turn on remote login on the computer that will host the program being debugged. Select:

   System Preferences: Sharing > Remote Login

2. Ensure that you can connect to the remote host using SSH public key authentication. If you are unsure whether you are using SSH public key authentication, you can test this by logging in to the remote computer with `ssh`. If you are prompted for the user's password, you are not using public key authentication. If you are prompted for a passphrase—or for nothing at all—you are already using public key authentication.

If you are not set up to log in to the remote host using SSH public key authentication, you need to create a public/private key pair, and configure the local and host computers to use it. You can do so with the following steps:

1. Generate a public and private key pair using `ssh-keygen`. On the command line, type the following line:

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

   You should see output similar to the following:

   ```
   Generating public/private dsa key pair.
   Enter file in which to save the key (/Users/admin/.ssh/id_dsa):
   /Users/admin/.ssh/id_dsa already exists.
   Overwrite (y/n)? y
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in /Users/admin/.ssh/id_dsa.
   Your public key has been saved in /Users/admin/.ssh/id_dsa.pub.
   The key fingerprint is:
   ```

   Note: Do not leave the passphrase empty; if you do so, your private key will be unencrypted.

2. Copy the public key to the `authorized_keys` file on the remote computer. This file is usually stored at `~/.ssh/authorized_keys`. If the `authorized_keys` file already exists on the remote computer, be careful not to overwrite the file. You can add the public key, which is stored in the file you specified to `ssh-keygen` (`id_dsa.pub` by default), by entering the following on the command line:

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

3. Make sure that the `authorized_keys` file is not readable by anybody else. Change the permissions on the file by entering the following on the command line:

   ```
   chmod go-rwx ~/.ssh/authorized_keys
   ```

4. Test the connection by logging in to the remote computer using `ssh`. From the command-line, type `ssh <username>@<hostname>`. Ensure that you are not asked for the user's password. If you did not leave it empty in step 1, you should be prompted for your passphrase, as in the following example:

   ```
   Enter passphrase for key '/Users/admin/.ssh/id_dsa':
   ```

If you are debugging a GUI application, you must be logged in to the remote computer as the same user that you connect to using `ssh`. This user must have permission to read the build products.

Creating a Shared Build Location

For remote debugging to work, both computers—both the local computer running Xcode and the remote host running the program you are debugging—must have access to your project's build products and intermediate files. You can do this in either of two ways:

- Create a single shared location. This is easiest with a network home directory, although you can use any shared folder that both computers can access. In Xcode, set the build products and intermediates location to this shared folder. If necessary, create symbolic links to the build folder on the remote host so the path to the build products is the same on both computers. For details on how to set the location of the build folder in your Xcode project, see "Build Locations" in *Xcode Project Management Guide*.
- Copy the files to the remote host. Alternatively, you can copy the build products and intermediate files over to the remote host after each build, although this is considerably less convenient. These files must be located at the same path on the remote computer.

Configuring an Executable for Remote Debugging

After you have configured both computers to allow for remote login and have set up a common build products location, the last step is configuring the executable you want to debug remotely. You may consider creating a separate custom executable environment for remote debugging. Using separate executable environments, you can specify different options for debugging remotely and debugging locally and easily switch between the two modes.

See "Executable-Environment Debugging Information" in *Xcode Project Management Guide* to learn how to configure an executable for remote debugging.

To start a remote debugging session, make sure the active executable is correctly set, then build and debug your product as normal. Before it launches the executable, Xcode displays an authentication dialog that asks you to type in your passphrase. After you have authenticated once, Xcode does not prompt you for your passphrase again until the next time you initiate a remote debugging session after restarting Xcode.

If you are experiencing problems debugging on the remote host, look in the console for error messages. To view the console, choose Run > Console.

Low-Level Debugging Support

Many of the subsystems in Mac OS X include debugging facilities that can help you in your debugging tasks. You can use most of these debugging facilities along with Xcode. Many debugging facilities are enabled or disabled by setting an environment variable; you can modify the executable environment to set these environment variables from Xcode. Xcode also includes several options for enabling specific debugging options, such as `libgmalloc` (Guard Malloc), loading debug library variants, and stopping on Core Services debugging functions (described in "Pausing on Core Services Debugging Functions" (page 35)). For more on the many debugging facilities available in Mac OS X, see *TN2124: Mac OS X Debugging Magic*.

Accessing Mac OS X Library Debug Symbols

Many Mac OS X system frameworks include debug versions, in addition to the production version. These library variants are identified by their `_debug` suffix. Debug variants of the system frameworks usually include debugging symbols, extra assertions, and often extra debugging facilities. You can modify the executable environment to have Xcode use the debug variants for libraries that your program loads. To use the debug variant of a library, open the inspector for the executable environment that you use to run your program. In the General pane, choose "debug" from the menu "Use [*suffix*] suffix when loading frameworks."

Using Guard Malloc

Xcode also integrates Guard Malloc (`libgmalloc`) into the debugger interface. Guard Malloc helps you debug memory problems by causing your program to crash on memory access errors. Because Guard Malloc causes your program to crash, you should use Guard Malloc with the debugger. When a memory access error occurs and your program crashes, you can look at the stack trace, determine exactly where the error occurred, and jump to the location of the problem.

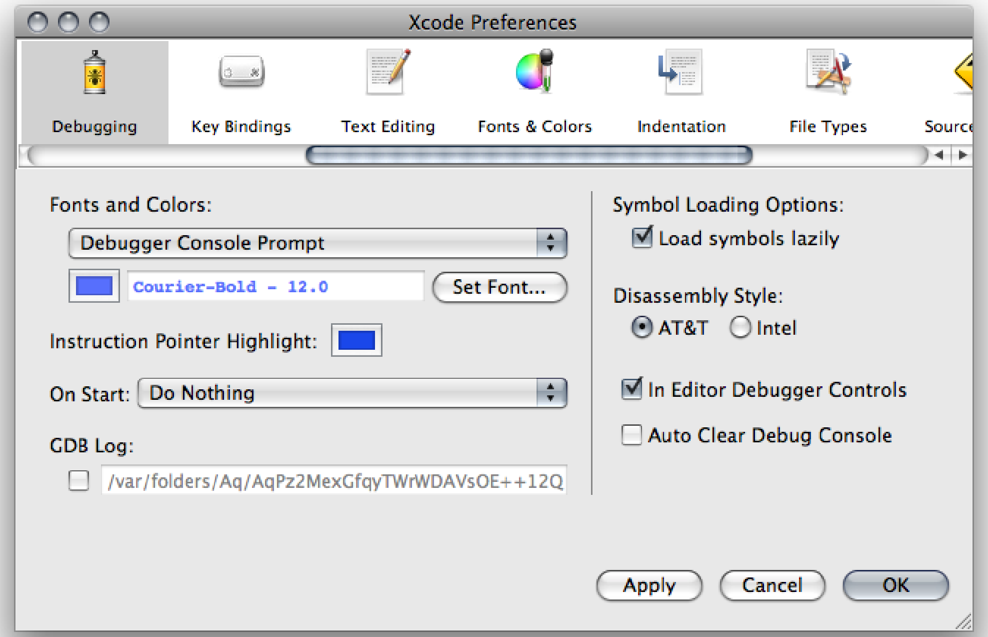
To enable debugging with Guard Malloc from Xcode, choose Debug > Enable Guard Malloc before starting the debugging session. You can also use Guard Malloc with GDB from the command line, by setting the `DYLD_INSERT_LIBRARIES` environment variable, as described in the man page for `libgmalloc`.

Guard Malloc has a number of additional options available. You can take advantage of these by setting the appropriate environment variables on the executable. In the inspector window for the executable, open the Arguments pane and add the environment variables to the environment variables table at the bottom of the window. See the man page for `libgmalloc` for additional details and information.

Debugging Preferences

The Debugging pane of Xcode preferences contains options for customizing the debugger console, the text editor debugging experience, and other debugging aspects. Figure 11-1 shows the Debugging preferences pane.

Figure 11-1 Debugging preferences pane



Fonts and Colors:
Debugger Console Prompt
🟦 Courier-Bold - 12.0

Instruction Pointer Highlight: 🟦

On Start: Do Nothing

GDB Log:
☐ /var/folders/Aq/AqPz2MexGfqyTWrWDAVsOE++12Q

Symbol Loading Options:
✓ Load symbols lazily

Disassembly Style:
[✓ AT&T] Intel

✓ In Editor Debugger Controls

☐ Auto Clear Debug Console

Here's what the pane contains:

- **Fonts and Colors:** Specifies the color and font used for text in the console.
- **Instruction Pointer Highlight:** Specifies the color used to highlight the location of the instruction pointer in the debugger window when execution of the current program is stopped.
- **On Start:** Specifies actions to perform when you launch a program from Xcode. The actions include showing the console, the debugger, or the mini debugger.
- **GDB Log:** Specifies a file into which GDB logs its activities.
- **Symbol Loading Options:** Specifies symbol loading behavior.

  - ❏ **Load symbols lazily:** When selected, the debugger defers loading symbols until they are needed. Otherwise, Xcode loads all symbols for the executable and its libraries when you launch it in the debugger. You can further customize which symbols are loaded in the Shared Libraries window. See "Viewing Shared Libraries" (page 42) for more information.

- **Disassembly Style:** Specifies the disassembly format used in the text editor and the debugger. See "Viewing Disassembly Code and Processor Registers" (page 24) for more information.
- **In Editor Debugger Controls:** Specifies whether the debugger strip appears in the text editor. See "Debugger Strip" (page 12) to learn about the debugger strip.
- **Auto Clear Debug Console.** Specifies whether to clear the debugging console at the start of a debugging session.

Document Revision History

This table describes the changes to *Xcode Debugging Guide*.

Date	Notes
2009-01-06	Made minor content changes.
	Added lazy-symbol—loading requirement when using non–Xcode build systems to "Building a Product with Debugging Symbols" (page 10).
2008-11-19	Made minor content addition and changes.
	Added "Debugging Preferences" (page 55) (previously published in *Xcode Workspace Guide*).
2008-10-15	Fixed content problems and typos.
	Removed 2048 key-size specification from instructions in "Configuring Remote Log-in" (page 49).
	Specified that the Stabs debugging information format is no longer supported.
	Performed minor content reorganization.
2008-05-16	New document that describes the Xcode debugging facilities and the recommended debugging techniques.