

[Xcode](#) / [Debugging](#) / Diagnosing and resolving bugs in your running app

Article

Diagnosing and resolving bugs in your running app

Inspect your app to isolate bugs, locate crashes, identify excess system-resource usage, visualize memory bugs, and investigate problems in its appearance.

Overview

Unit testing determines whether your code delivers results that meet your expectations, but it doesn't explain the cause when it doesn't. To diagnose an error, attach the debugger, reproduce the error, and then narrow down its root cause by inspecting your variables at key points in your code while the app is running with breakpoints. If you configure a scheme's run action for debugging by using the Debug executable checkbox in Info settings, the app will attach to the debugger automatically when the app uses the scheme. To attach the debugger to a process that is already running, choose Debug → Attach to Process, and select your app's process from the list. Follow this same process to diagnose and resolve errors in your code, crashes, memory leaks, and layout problems.

Pause the app to inspect variables and isolate bugs

To fix a bug, you first need to understand what is causing it. To narrow down the cause of a bug, develop a set of steps to reliably reproduce it:

1. Determine where the bug happens in your source code.
2. Pause your app with a breakpoint in your source code before the point at which you believe the bug occurs.
3. Look at your variables and confirm they have the values you expect. If they don't, begin again with step 1.

4. Step through your code and watch your variables change. Note where your variables have unexpected values.

5. Analyze your code to determine a fix.

After determining a potential fix for the bug, confirm the diagnosis by changing your code and retesting to reproduce it. If the change addresses the problem, you've resolved the bug. If your change doesn't resolve it, reconsider where the bug might be occurring, and repeat the steps to isolate and fix it.

For more information on setting breakpoints and inspecting variables, see [Setting breakpoints to pause your running app](#) and [Stepping through code and inspecting variables to isolate bugs](#).

Locate crashes, exceptions, and runtime issues

When your app experiences a crash, exception, or runtime issue, it can be challenging to pinpoint the code causing the problem because the stack trace for the crash doesn't always point to the line of code that causes the crash. Use the rubric below to identify the problem characteristics and then set the correct type of breakpoint:

- A crash that stops at `main` or highlights `AppDelegate` is frequently an Objective-C exception.
- A crash that is the result of a runtime issue also stops at `main` or highlights `AppDelegate`, and may have a message similar to: "Thread 8: EXC_BAD_INSTRUCTION (code=...)".
- A crash that stops at an uncaught or unhandled Swift error displays a fatal error message and indicates a Swift error.

Add a breakpoint to your code in a location based on problem characteristics, then when your app stops at the breakpoint, check the state of the code execution. For more information on setting breakpoints and identifying crashes, see [Setting breakpoints to pause your running app](#) and [Identifying the cause of common crashes](#).

Note

Your Swift code can receive an Objective-C exception when it uses code from a module that uses Objective-C.

Inspect variables and execution sequence without pausing

When you develop code, it's helpful to log actions and variable values so you understand how your code runs and what values your variables have at different points in your app. This is especially true when you develop *concurrent code*, or code that executes simultaneously across multiple queues or threads, because bugs can be intermittent and difficult to reproduce. Often, you

reproduce a bug in normal execution, but not when stepping through the debugger, because the timing is different between normal execution and debugging. The debugger provides tools to inspect variables without pausing and disturbing the timing of your concurrent code.

Developers commonly add `print` or `NSLog` statements to see variable values. While this technique works, it adds extra code that isn't useful after you finish development, and leaves your app with a noisy console that makes diagnosing subsequent bugs more difficult. Instead, use breakpoint actions to know when events in your app take place, and inspect variable values without pausing.

To determine whether your code executes with minimal effect to timing, use a breakpoint action to play a sound and continue executing. If the debugger reaches the breakpoint when you run the app, it plays the sound and confirms execution.

To log a variable value to the console without pausing, add a breakpoint with a Debugger Command action using `po` to print out the evaluation of an object, or `v` to print the value of a variable to the console. Select the Automatically continue after evaluating actions option for the breakpoint to prevent pausing.

The screenshot shows the Xcode breakpoint editor for a breakpoint named 'Enable Breakpoint in ListDataSource.swift:38'. The 'Name' field is empty, with a note stating 'A breakpoint name cannot start with numbers or contain any white space.' The 'Condition' field is also empty. The 'Ignore' field is set to '0' with a spinner, followed by the text 'times before stopping'. The 'Action' field is set to 'Debugger Command' with a dropdown arrow. Below this, a text box contains the command 'po indexPath'. To the right of the text box are '+' and '-' buttons. At the bottom, the 'Options' section has a checked checkbox for 'Automatically continue after evaluating actions'.

To log custom text to the console and add context to variable values, add a breakpoint with a Log Message action. Specify your custom text, and include expressions, the breakpoint name, or the breakpoint hit count to provide more information.

Note

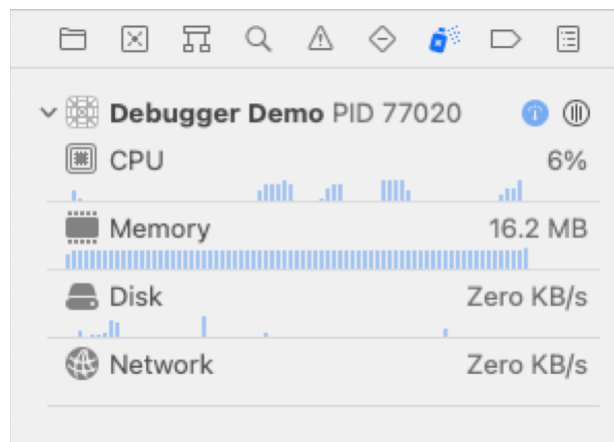
Because `po` compiles code dynamically to evaluate expressions, it takes more time to evaluate your variable and log it to the console. To reduce timing issues, use `v` to log variable values instead.

Use other breakpoint actions to execute an AppleScript or a shell script, or to capture a GPU frame.

For more information on inspecting variables, see [Setting breakpoints to pause your running app](#) and [Stepping through code and inspecting variables to isolate bugs](#).

Identify potential overuse of CPU and memory

An easily overlooked and common problem in development and testing is the overuse of CPU and memory. Xcode's debugger provides gauges in the Debug navigator to help investigate potential problems. Monitor the gauges while you're testing your app to uncover unusual usage. Click a gauge for a more detailed view.



The CPU gauge shows the amount of CPU the app requires to process instructions over time. When your app is drawing the user interface, processing data it retrieves from the network, or performing calculations, it's normal to see CPU usage increase to fairly high numbers for a short period of time. When those tasks are complete and your app is idle and waiting for the user to perform an action, CPU usage should be zero or very low. Do additional analysis if CPU usage is:

- Persistent at a level above zero when the app appears to be idle.
- Over 100% for more than very short periods of time.
- Very high and you see hitches in your user interface.

For more information on improving performance, see [Improving your app's performance](#).

The Memory gauge shows how much memory your app uses over time. It starts at a fairly small number, less than 10 MB, when you first launch your app, and then increases as people navigate through your user interface. It may also increase if you fetch, process, and store data from the network, or perform complex calculations. It then decreases when processing is complete. Watch the gauge as you navigate through your app, and note when memory usage goes up and down. Memory usage increases when you present modal views or add a view to a navigation controller, and decreases when you dismiss or navigate away from those views. If your usage continues to increase and doesn't ever decrease, investigate whether you have a memory leak or abandoned

memory. For more information on reducing memory use and resolving memory leaks, see the section below [Visualize and diagnose increasing memory usage](#) and [Reducing your app's memory use](#).

Detect high disk access and network use

Be aware of issues resulting from frequent access to resources on disk and over the network. You can monitor these resources using the gauges in Xcode's debugger as well. The Disk I/O gauge shows how much data your app reads and writes from disk over time. The gauge shows if you:

- Store data that the user generates in your app.
- Store data in user preferences.
- Fetch data from the network and store it.
- Read data from your app bundle or the app's directories.

Storing and reading data frequently from disk uses more power than doing so from memory, and it adds wear and tear to the user's device. To know whether disk usage is unusual, you need to understand the size of data you're storing and reading, and compare that to what you observe in the gauge. For example, if you download and store a 5 MB graphic file for display in a view that you use frequently and it writes over 50 MB of data, investigate whether the remote image changes frequently, or whether you need to configure networking to prevent redownloading the same image. If you're reading more data from disk than you expect, investigate whether a memory-caching solution might help, or whether you're initiating a data read from the wrong point in your app or view life cycle, and reading it too often. For more information on reducing disk writes, see [Reducing disk writes](#).

The Network I/O gauge shows how much data your app reads from and writes to the network over time. If your app only uses local resources, your app may not read or write any data from the network. Communicating data over the network uses energy and reduces the device's battery life, so minimize data transfer wherever possible. To understand your app's network usage, watch the Network I/O gauge when your app is sending or receiving data from the network. For example, if you implement a cache system for downloaded images and your network usage increases when accessing them, confirm that your cache settings are correct in the app and on the server. If you're uploading user-generated content and frequent upload failures during poor networking conditions lead to high network usage, implement a system to recover and restart failed uploads at the point of failure, rather than reuploading the entire file.

Visualize and diagnose increasing memory usage

Diagnose the cause of memory leaks and abandoned memory with the memory graph. The observable symptom of a memory leak is memory usage that continues to increase over time,

even when conditions in the app indicate that memory usage is decreasing. A memory leak can occur in a *retain cycle*, which is when objects maintain strong references to each other, but the app no longer references them. These objects remain in memory and the app can't remove them. *Abandoned memory* occurs when you create objects and your code still references them, but your app no longer needs them or uses them.

To see the memory graph in the debugger, pause your app at a breakpoint and click the Debug Memory Graph button in the debug bar. Alternatively, click the Debug Memory Graph button when the app is running to pause the app and show the memory graph.

The memory graph view replaces the stack trace in the Debug navigator with a list of types, organized by library, each with a list of instances called *nodes*. Select a node to view its memory graph.

A node's memory graph shows all the memory references to that node, and highlights strong references. Control-click any node in the graph to perform more actions, such as accessing Quick Look or printing the description to the console. Choose Focus on Node to show the graph for the selected node. Click a reference to see its details, including the name of the variable, the type of reference, and the source and destination objects in memory.

To resolve a memory leak for a retain cycle:

1. Observe the Memory gauge while you navigate the app.
2. Note when memory usage increases when your app instantiates an object, but doesn't decrease when you expect the system to deallocate the object.
3. Examine the memory graph to see if there are an unexpected number of instances of the object or inappropriate strong references to it.
4. If there is a strong reference to the object, Control-click the node with the strong reference and choose Focus on Node to view its graph. If the node also has a strong reference from the object, this is a retain cycle.
5. Resolve the retain cycle by changing one side of the relationship to use a weak declaration for the reference to the other object, or remove the reference altogether by removing any dependencies on the other object. Retest to confirm that the change fixes the issue.

To resolve abandoned memory issues, identify the time in your app's life cycle that it no longer needs the abandoned object and remove any references to it.

Inspect and resolve appearance and layout issues

Some issues in the appearance or layout of your app only appear when you configure the system with a particular interface style, dynamic text size or when your app uses particular accessibility features. Use environmental overrides when targeting iOS, iPadOS, macOS, and tvOS apps to test your interface in these environments. To understand issues that involve the position or size of your view, you might need to inspect them within the context of views in other layers. Use the view debugger when targeting iOS, iPadOS, macOS, tvOS, and watchOS apps, which displays a 3D representation of the view hierarchy in layers, to help diagnose these issue. Entities within a visionOS app and their surroundings sometimes interact with each other in ways you don't expect. Enable visualizations to represent coordinate axes, bounding boxes, and other information that is normally invisible, to help understand these interactions.

For information on using these features to debug the appearance of your app, see [Diagnosing issues in the appearance of a running app](#).