# Shortcuts

* Control (or Ctrl) ⌃
* Option (or Alt) ⌥
* Shift ⇧
* Command (or Cmd) ⌘

|  |  |  |
| --- | --- | --- |
| **Groups of Functions** | **Functions** | **Shortcuts** |
| **Code Editing** | Show completions | Ctrl + Space 🡪 Cmd + Space |
| Toggle single comment | Cmd + / |
| Multiple cursors | **Each time**: Ctrl + Shift + Click 🡪 Ctrl + Click  **All-in-one**: Alt + Drag |
| Move line(s) up / down | Alt + Cmd + [ / ] 🡪 Alt + ↑ / ↓ |
| **Navigation** | Jump forward | Ctrl + Cmd + → 🡪 Alt + → |
| Jump back | Ctrl + Cmd + ← 🡪 Alt + ← |
| Jump to line | Cmd + L 🡪 Cmd + G |
| Jump to file / ~~method~~ | Cmd + Shift + O |
| Switch to Header/Source | Alt + Cmd + ↑ 🡪 Alt + O |
| Related items (counterpart, include from, include by, etc.) | Ctrl + 1 |
| Go to definition / Go to caller(s) | Cmd + click |
| Open definition | Cmd + double‐click |
| Find in ﬁle | Cmd + F |
| Find in project | Shift + Cmd + F |
| Line beginning | Cmd + ← 🡪 Home |
| Line end | Cmd + → 🡪 End |
| File top | Cmd + ↑ 🡪 Cmd + Home |
| File bottom | Cmd + ↓ 🡪 Cmd + End |
| Select occurrences of current word | **Each time**: Cmd + + E then Cmd + + G  **All-in-one**: Alt + Cmd + E 🡪 Cmd + F2  **Redo**: |
| **Build and Run** | Build | Cmd + B |
| Run (after building) | Cmd + R |
| Profile | Cmd + I |
| Stop the running app | Cmd + . |
| **Debuging** | Step over | F6 |
| Step into | F7 |
| Step out | F8 |
| **Documentation** | Open pop‐up help window | On a symbol, press Alt + Click |
| **Other** | Save all | Alt + Cmd + S 🡪 Cmd + Shift + S |
| Add to bookmark | Cmd + D |
| Open Preferences / Settings | Cmd + , |

MAC OS

Switch program: Cmd + Tab (same as Alt + Tab on Win)

Next window: Cmd + `

Close window: Cmd + W (same as Win)

Delele files: Cmd + Delete

New folder: Cmd + Shift + N (same as Win)

Spotlight: Cmd + Space (same as Win + typing on Win)

Begining / Ending of line: Cmd + ←/ →

Begining / Ending of file: Cmd + ↑/ ↓

Minimize the front window to the Dock: Cmd + M

Minimize all windows of the front app: Alt + Cmd + M (same as Win + D on Win)

Hide the windows of the front app: Cmd + H

Hide the windows of other apps: Alt + Cmd + H

Show desktop: F11

Undo on document: Cmd + Shift + Z (same as Ctrl + Y on Win)

# Allocations and Leaks (Instruments)

## Allocations and Leaks

The allocation is a much broader term than the leak.

Both ***Allocations*** and ***Leaks*** tool would help you identify this.

* Abandoned memory: Is due to strong pointers from a persistent object that won’t get released while our app is running. For example, a class with a timer being strongly pointed by RunLoop, or a DispatchSourceTimer being strongly pointed by GCD, etc. ***Allocations*** will help you identify this. ***Leaks*** can't detect this.
* Cached memory: This is an interesting case. It's not a programming error; it's just a mistake. Suppose you cache images that aren't really needed, e.g. all photos of last year, while 90% of the users care about photos of last month only. ***Allocations*** will help you visualize the memory growth. ***Leaks*** won't because it's not a leak. Technically you could write an app with 0 leaks, but one that caches everything and consumes 10GB – Users will hate you for that!

You really need to understand that:

Strong reference cycle (a.k.a. leaks) warning is produced when there are two (or more objects) whose only strong references are between each other. Abandoned memory or cached memory is neither of them!

# Others

For the strong references (that create leaks, but aren't reference cycles) e.g. Timer, DispatchSourceTimer, DispatchWorkItem, the memory graph doesn't create a purple icon, I suspect it's simply because it doesn't find two objects pointing back to each other strongly. <https://stackoverflow.com/questions/56261915/does-xcode-memory-graph-offer-any-smart-visual-indicators-for-strong-references>

## How to use Allocation and Leaks:

<https://developer.apple.com/videos/play/wwdc2012/242/>

<https://www.swiftdevjournal.com/measuring-your-apps-memory-usage-with-instruments/>

<https://developer.ibm.com/tutorials/mo-ios-memory/>

## How to use Memory Graph Debugger:

<https://medium.com/zendesk-engineering/ios-identifying-memory-leaks-using-the-xcode-memory-graph-debugger-e84f097b9d15>

## Over-releasing

Despite your best efforts, you will occassionally accidentally release something twice instead of once, or maybe release something that you don't own. This will cause a crash. Fortunately, these bugs are pretty easy to find if you turn on **zombies** in Instruments. And I'm not talking about sexually arousing the walking dead inside of a clarinet, or anything like that.

Xcode 4 makes this pretty easy:

* Click and hold the "Run" button in the toolbar.
* Select "Profile" from the drop down menu.
* When Instruments pops up, select the "Zombies" instrument.
* Go to your running app and trigger the crash.
* Instruments will pop up a little box that says something like: "You messaged a zombie at 0xDEADBEEF." Click the little arrow in there.
* In the bottom pane, Instruments will show you every single retain, release and autorelease that ever happened to the object, so you can figure out the problem from that.

## Retain Cycles

Normally you retain an ivar in a setter or an initialiser method, then you release it in dealloc. That way, when an object is deallocated there is a cascading effect. The root object releases its children, then it's children release their children, until the whole data structure is fully released.

If you have a situation where object X owns object Y, and object Y also owns object X then you have a problem. X and Y will never be deallocated while they own each other, because they are keeping each others retain counts at one. So you just end up leaking both of the objects.

The reason why objects don't retain their delegates is because the delegate is usually the owner of the object. Image if delegates were retained. **The NSWindowController owns the NSWindow, and the NSWindow owns it's delegate, which just happens to be the NSWindowController. Now you've leaked an entire window and its controlle**r, which could take up a huge chunk of memory.

The way you get around this, is you basically say "Ok, the controller is going to live longer than the window which means we shouldn't get any dangling pointers, so just don't retain the controller." If you want to be super safe, you can set the delegate to nil inside of the dealloc of the window controller. That way, even if the window outlives the controller, you can be sure that there won't be a dangling pointer.