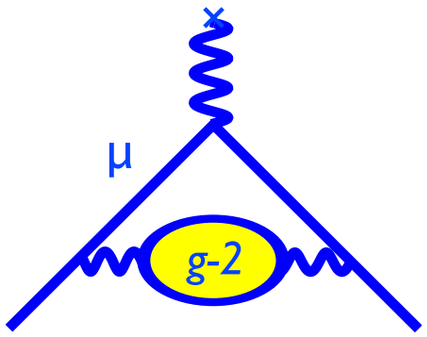


GM2-doc-2459

Developing g-2 Code on your Mac

Adam L. Lyon
FNAL/SCD

January 2015
Updated Feb 2016

UPDATES TO THIS DOCUMENT

Things have changed

Upgrading to El Capitan (Don't). If you upgrade to El Capitan (OSX 10.11), you will not be able build nor run g-2 code on your Mac. Apple has introduced a new security feature called System Integrity Protection (see https://support.apple.com/en-us/HT204899 for more information). This feature ruins our "ups" style of running. You can turn this feature off system wide, but it is not clear if this option will exist in future OS versions. Furthermore, at this point the *art* code has not been updated to El Capitan.

You can either stay on Yosemite (OSX 10.10) or upgrade to El Capitan and use a Linux virtual machine or Docker container on your Mac. Instructions for the latter will appear in a future document.

Using Xcode 7.2 and higher. If you upgrade your Xcode to version 7.2 or higher, you will need to change an internal default setting to allow Xcode to inherit the calling environment (this allows Xcode to see environment variables from your session - necessary for the instructions in this document).

You need to run the following command just once after upgrading your Xcode...

```
defaults write com.apple.dt.Xcode \
    UseSanitizedBuildSystemEnvironment -bool NO
```

Once that command has been run, Xcode will behave properly. Note that for some reason the newer versions of Xcode no longer understand Kerberos, so you will not be able to push or pull with git and Redmine from within Xcode. You can use the command line or a nice free application called SourceTree (https://www.sourcetreeapp.com/) to replace this functionality.

New CVMFS /cvmfs/gm2.opensciencegrid.org. Muon *g-2* has switched to a different CVMFS server. This document has been updated accordingly (replace /cvmfs/oasis.opensciencegrid.org with /cvmfs/gm2.opensciencegrid.org).

OVERVIEW

Setting up your Mac

- *Installing CVMFS*
- *Installing Xcode*

1

Running and Configuring *Xcode* & Instruments

- *What are Xcode and Instruments*
- *How to launch Xcode*
- *Configuring Xcode for your development area*
- *How to launch Instruments*

2

Using Xcode

- *Quick tour*
- *Navigating Source Code*
- *Search and replace*
- *Git Integration*
- *Build*
- *Debug*
- *More*

3

Using Instruments

- *Configuring Instruments*
- *Time Profiler*
- *Memory Allocation and Leak Profiler*

4

WELCOME!

Apple provides a professional set of development tools for free. You should use them!

Since our Muon *g-2* code builds on a Mac, you can use your Mac laptop or desktop for development. Apple provides a professional set of tools, *Xcode* and *Instruments*, for your code development use.

Note: It is important to remember that running code on your Mac is still considered experimental. All long runs and code producing official results should run on the Grid under Linux.

For general information about Muon *g-2* code and development, you should see the manual at GM2-db-1825 or the `gm2swtools` product in the release.

There are several benefits that *Xcode* and *Instruments* give you over coding with text editors like Emacs and vi,

- *Xcode* is a very convenient code editor with advanced navigating and editing capabilities
  - You can click on a `#include` directive and jump to the header file
  - You can click on a class or object name and jump to its definition.
  - *Xcode* understands projects: you can view all of the source code that goes together to make a build
  - Easily search (and replace) text in a file or the entire project
- Code build errors are displayed in the source code at the problematic line
- Debugging with an IDE (not in the terminal)
- Built in git management and version comparisons
- Easily profile your code for speed, memory allocation, and memory leaks

Xcode is a very capable integrated development environment (IDE) and we'll scratch the surface here.

Note that while we are developing on the Mac, we are not using the Apple compiler nor Apple's framework environment. Limitations will result.

RESOURCES FOR LEARNING MORE

The WWDC videos are useful to learn Xcode and Instruments

Xcode comes with extensive documentation. Some go very deep and some are very terse. I have found that several Apple WWDC (World-Wide Developer's Conference) videos give a lot of good information.

To view the videos, you will need to use *Safari* and you will need an Apple Developer's account (it's free). Here is a list of videos that I have found useful.

Many of the videos talk about Mac specific functions that we do not use, but there's enough general information to make them useful.

Xcode is a big complicated program! Have fun trying stuff and learning it.

WWDC 2012 (link):

- *Working Efficiently with Xcode.* Nice explanations of various ways of using the program (single window, single window with tabs, multiple windows)
- *Debugging in Xcode.* Useful demonstration of the *Xcode* debugger
- *Advanced Debugging with LLDB.* Advanced tips for using the Apple debugger (mostly works with our code)
- *Learning Instruments.* A somewhat useful tour of *Instruments*

WWDC 2013 (link):

- *Core Xcode.* Lots of useful tips for using *Xcode*, including a section on *Xcode* fundamentals

WWDC 2014 (link):

- *Improving your App with Instruments.* A good tour of *Instruments*

Tutorials:

A very nice tutorial of Xcode tricks is at
http://www.raywenderlich.com/72021/supercharging-xcode-efficiency

Setting up your Mac

- *Installing CVMFS*
- *Installing Xcode*

1

1.1 HOW TO INSTALL CVMFS

CVMFS makes distributing executables and libraries very easy.

We distribute our executables and libraries via the CERN Virtual File System (CVMFS). It allows you to "subscribe" to published directories. You will get updates automatically as they are pushed out from Fermilab. We use the Open Science Grid Gm2 CVMFS service, supplying CVMFS for several experiments. You may learn more about CVMFS on the web.

If you have CVMFS already. Check if CVMFS is running with,

```
ps -ef | grep cvmfs | grep -v grep
```

If you get no response, then you are not running CVMFS. If you do get a response, then reboot your Mac now to start without CVMFS running.

If you have a `/cvmfs` directory (you've installed CVMFS before), then you should clear your CVMFS cache with,

```
sudo cvmfs_config wipecache
```

and remove all of the mount points with,

```
sudo rm -rf /cvmfs/* # Careful!
```

You may now re-install CVMFS.

Installing CVMFS. Go to http://cernvm.cern.ch/portal/filesystem/downloads to download the Mac OSX client of CVMFS. Install it and follow any instructions that it gives, including installing FUSE. Be sure to install a late version of FUSE (the page that CVMFS may send you to might be old - the correct page is https://osxfuse.github.io/).

Once the installation is complete, You create the configuration files by doing,

```
sudo cvmfs_config setup
```

Install mount point. You must now install the mount point with,

```
mkdir -p /cvmfs/gm2.opensciencegrid.org
```

Download configuration files. Download the configuration files from here (you should get a file called `cvmfs_mac_config_20140114.tgz`). Put it in the right place and then do

```
cd /etc/cvmfs
sudo tar xvzf /<YourDownloadPath>/cvmfs_mac_config_20140114.tgz
sudo cvmfs_config reload
```

Mount CVMFS. To start CVMFS, issue this command now and *whenever you reboot your machine*, (you may want to put this in a script)

```
sudo mount -t cvmfs gm2.opensciencegrid.org /cvmfs/gm2.opensciencegrid.org
```

Stopping and restarting CVMS. You can stop CVMFS with this command,

```
sudo umount -f /cvmfs/gm2.opensciencegrid.org
```

To restart, issue the mount command above.

Problems. You may find that after changing networks or having the computer sleep for a period of time you will get intermittent problems reading files in CVMFS (e.g. i/o errors). If this happens, simply restart CVMFS with (combination of the lines above),

```
sudo umount -f /cvmfs/gm2.opensciencegrid.org
sudo mount -t cvmfs gm2.opensciencegrid.org /cvmfs/gm2.opensciencegrid.org
```

1.2 HOW TO INSTALL XCODE

Xcode is freely available from the App Store.

Get *Xcode* from the Mac App store. It is free. Open the App Store (*Apple Menu > App Store*) and search for *Xcode*. It is a big file (nearly 2.5 GB), so it will take a long time to download and install. You should install it to your machine's main *Applications* folder. *Xcode* was at version 6.1.1 at the time this document was written.

Once *Xcode* has installed, you must check for the command line tools. Do,

```
ls /usr/include
```

You should see nearly 250 files. If you see none or very few, then do from a terminal,

```
xcode-select --install
```

Issuing that command will download and install the command line tools and will populate that and other directories with the correct files.

You are now ready to code with *Xcode*.

The *Instruments* profiling tool is downloaded and installed with *Xcode* automatically.

Running and Configuring *Xcode* & Instruments

- *What are Xcode and Instruments*
- *How to launch Xcode*
- *Configuring Xcode for your development area*
- *How to launch Instruments*

2

2.1 WHAT ARE XCODE AND INSTRUMENTS?

Xcode is an Integrated Development Environment (IDE), a software application providing comprehensive support for software development including source code editing, code building, and debugging. Instruments is an application for profiling software's use of resources.

Apple writes many application programs for their Mac computers as well as iOS devices (iPhone & IPad). They have made some of their development tools available for general use. While our software takes no advantage of the special capabilities and libraries on the Mac and does not run on iOS, we can nevertheless take advantage of the nice software development tools. Some of the features *Xcode* offers were detailed in the Welcome section of this document. You may find other information at the *Xcode* webpage, but note that most of the marketing materials focus on writing Mac/iOS applications.

There are other IDEs out there, many of which are compatible with the Mac as well as Linux. Examples include Eclipse, NetBeans, and IntelliJ Idea. All of these focus on Java development, but have features that work with C++. A new IDE called CLion is meant for C++ work, but is in an Early Access Build program. Despite all of these options, I have focused on *Xcode* as it seems, to me, to be the easiest to start using (only if you have a Mac, of course).

Instruments is an application that can profile your program and show a timeline of resource usages, including CPU time, memory, i/o, etc. It is easy to use and gives a huge amount information that may not be easy to interpret. Some scenarios are given in subsequent sections.

2.2 HOW TO LAUNCH XCODE

You must run Xcode in our software environment by launching it from the command line.

Once *Xcode* is installed, you will have an icon in your machine's Applications directory. But, to run *Xcode* for g-2 code, it must be running within our software environment so that it has access to our environment variables. Therefore, you should ***not** start Xcode by clicking on its icon*. You must instead start it from the command line by following the instructions here.

Prepare the environment. Launch a terminal window (I like iTerm2) and re-establish a session in a development area. For example, see step 1 on the right. If you haven't built the code before or have zapped the build area, you should run `cmake` from the command line by following step 2.

Launch *Xcode*. Assuming you installed *Xcode* in `/Applications`, follow step 3.

1
```
source /cvmfs/gm2.opensciencegrid.org/gm2/prod/g-2/setup
cd /path/to/my/dev/area
source localProducts*/setup
source mrb s    # Set up development environment
```

2
```
mrb b -C  # Runs cmake
```

3
```
/Applications/Xcode.app/Contents/MacOS/Xcode &
```

In your terminal screen, you will see messages from *Xcode*. These are benign, including the ones that tell you to report *Xcode* bugs, and may be ignored.

2.3 CONFIGURING XCODE

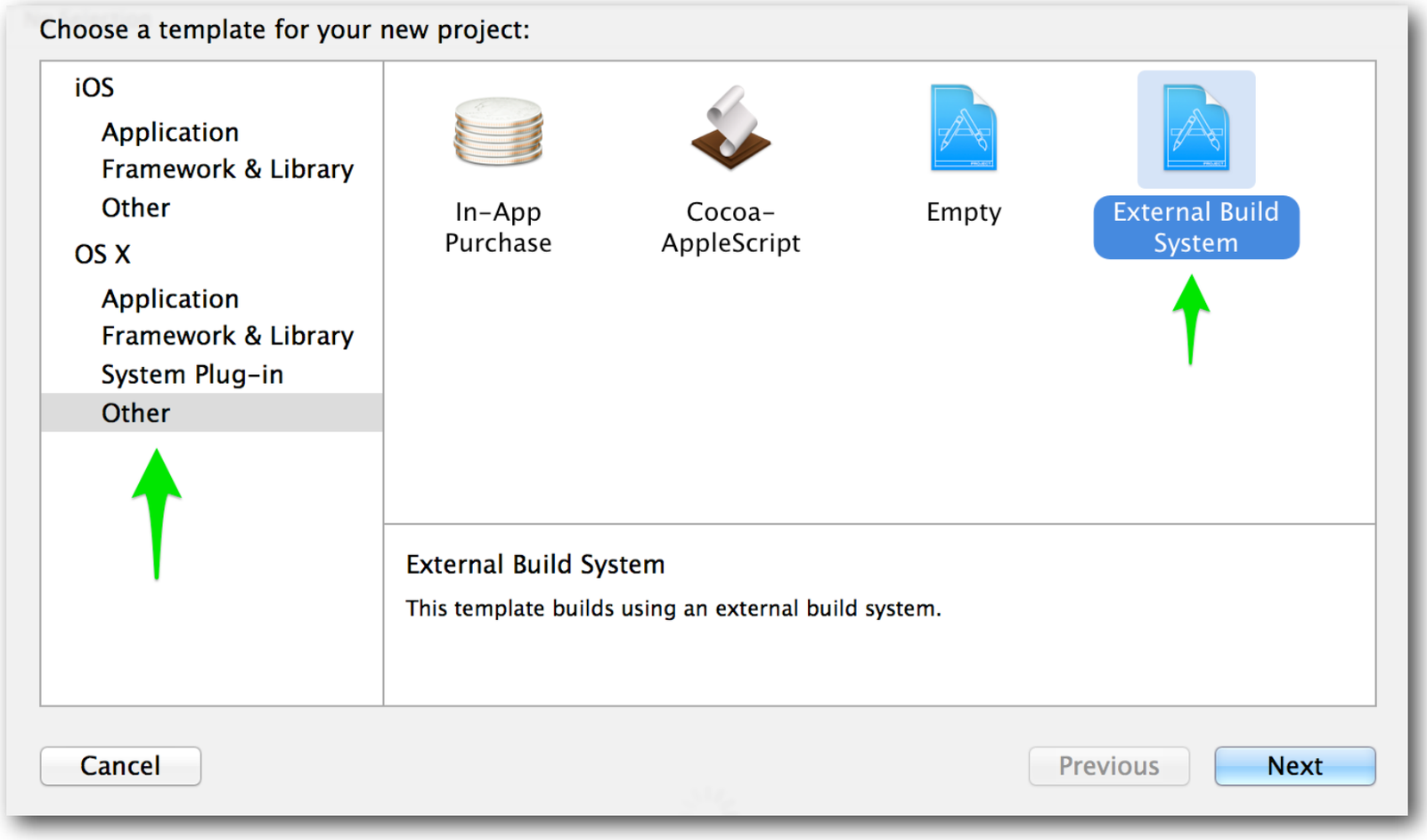
Do these steps once for your development area.

There are several steps that are required to configure *Xcode* for your development area. You only need to follow these steps once for each development area that you are using.

Create the *Xcode* project. Launch *Xcode* (see section 2.2 for how). The welcome screen should appear (see right). If that doesn't appear, then on the menu bar choose *Window > Welcome to Xcode*.

Click on *Create a New Xcode project*.

Choose the template. For the template, choose OS X > Other and then External Build System (as shown). Click Next.

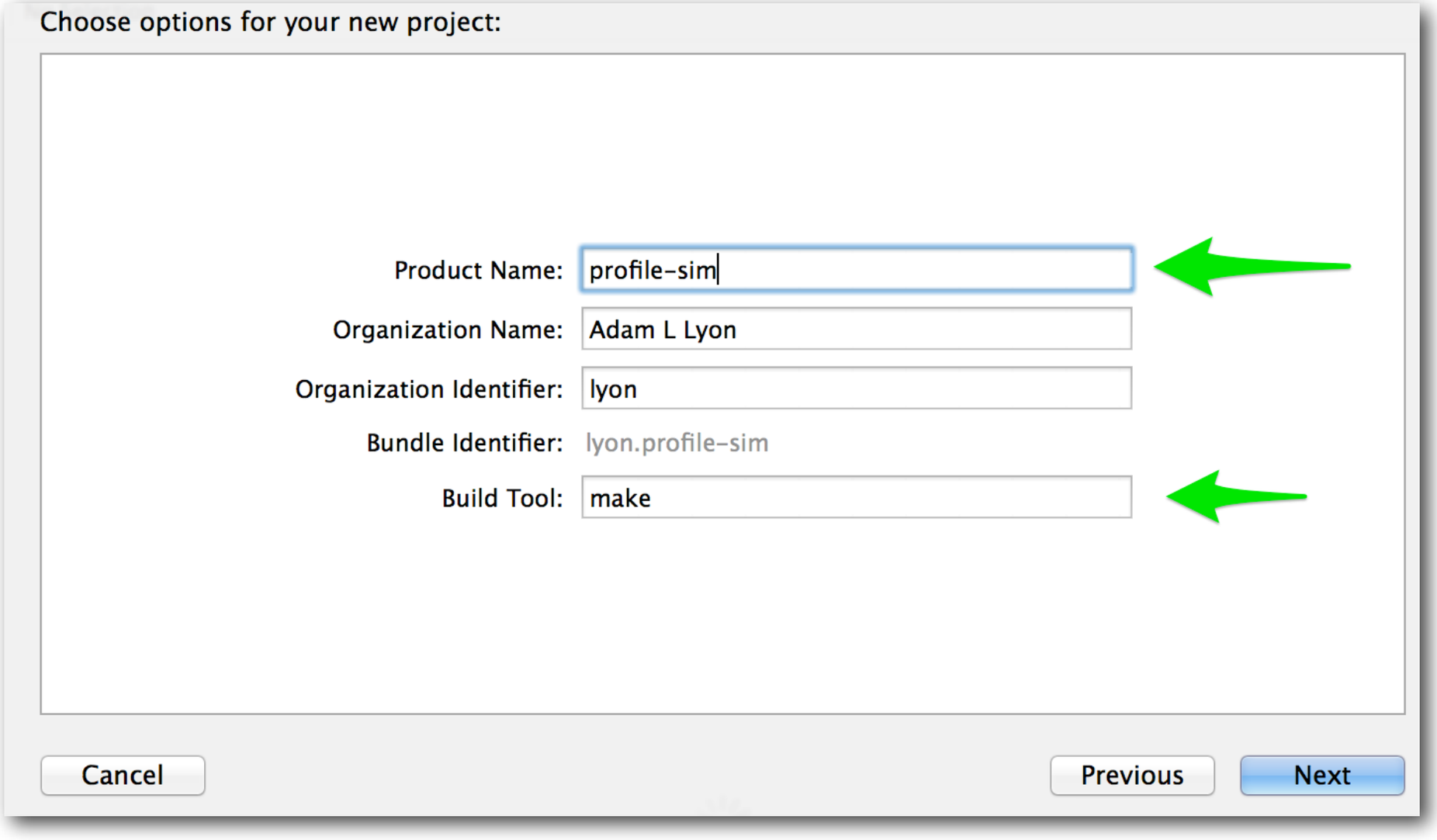


Choose project options.
Now fill in the options sheet. For the product name, it is a good idea to use the same-name as your development area directory.

The organization name and identifier don't matter. Put in anything you like, but your name is probably best.

For the build tool, enter "make" as shown.

Click Next.



Save the project. Now you must save the project files somewhere. These files are specially for *Xcode* and you should *not* save them in your development area nor put them in git.

I have a specific directory on my Mac where I put *all* of my *Xcode* project files called `~/Development/g-2/xcode`. You may want to do the same. *Xcode* will automatically add a directory there named by the product name for your project.

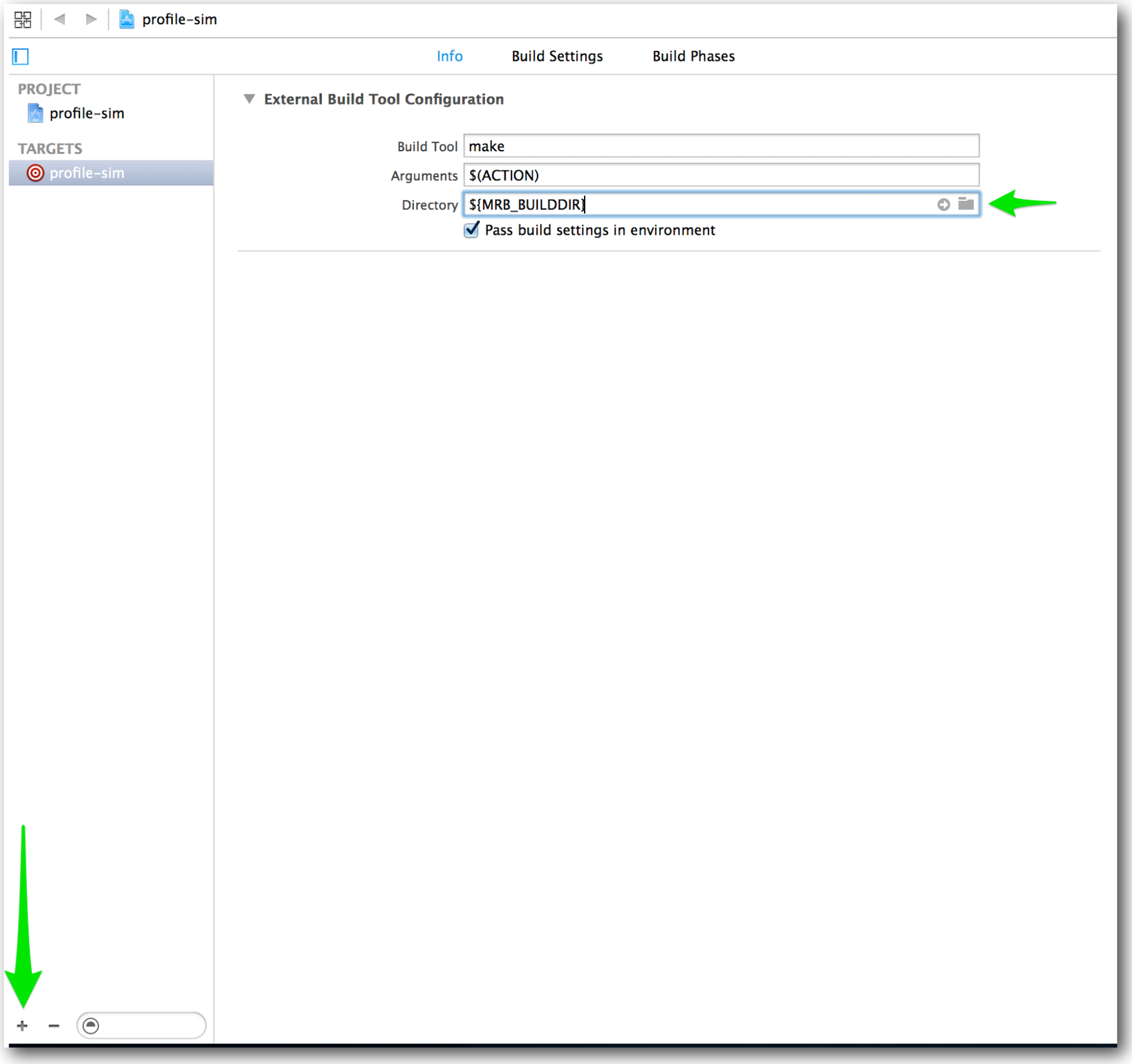
Be sure the “Create Git repository” check box is *unchecked*. You don’t need a new git repository - your sources already have git repositories and *Xcode* will use them.

Click on Create.

No figure necessary

Configure the project. The main project screen will now appear. First, in the External Build Tool Configuration section, for the Directory enter `${MRB_BUILDDIR}` in the box. Because you started *Xcode* from the command line with your development environment set up, *Xcode* will know about that environment variable.

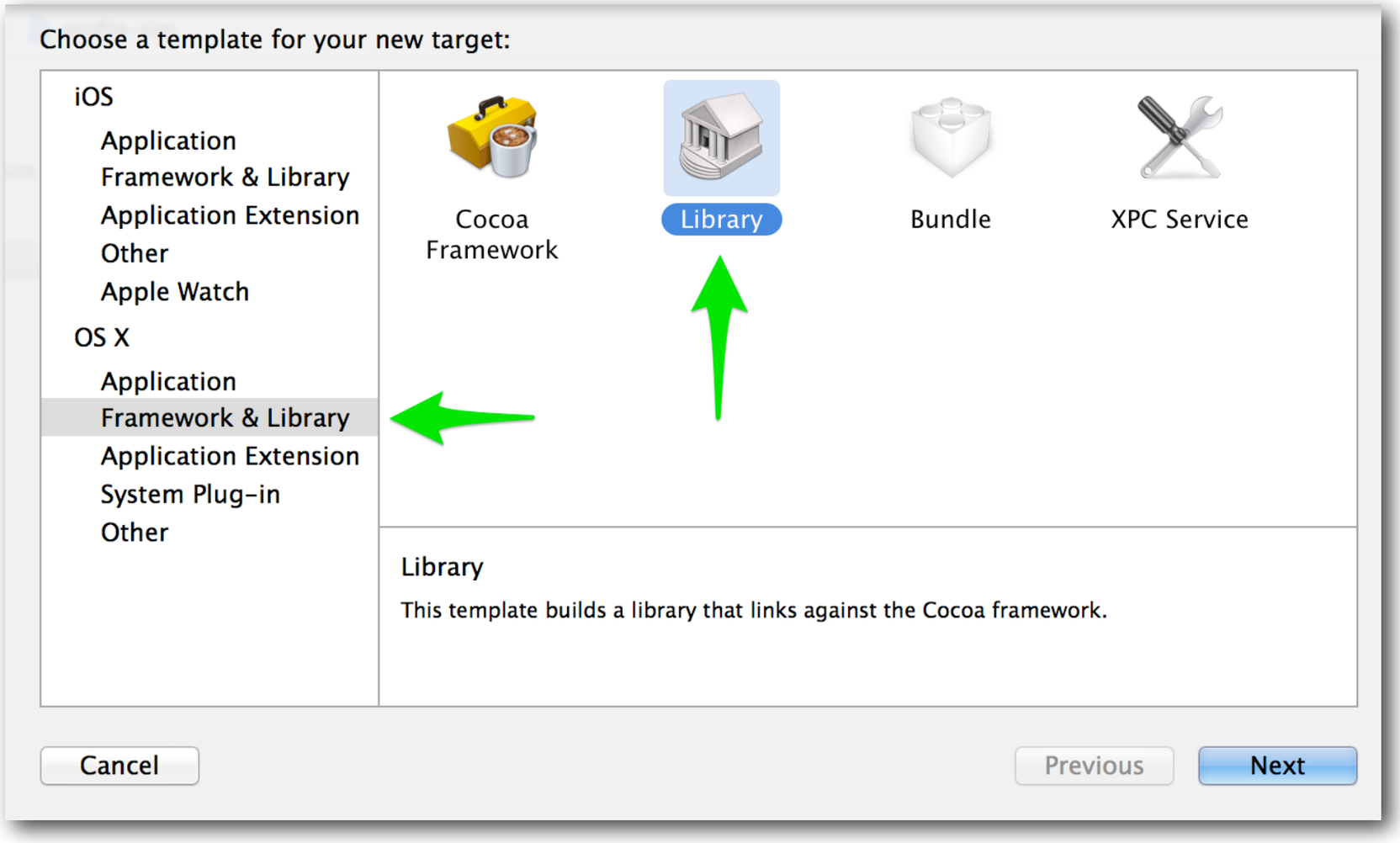
Start the documentation target. Unfortunately, an external build system project will not activate *Xcode*'s nice source code navigation system. To remedy this situation, we will add a documentation target. Start this process by clicking on the + sign in the left corner underneath Project and Targets.



The idea here is to create a fake library target so that *Xcode* will activate its source code navigation features. We won't actually use the library target for anything important.

Choose the target template. Under *OS X* choose *Framework & Library* then choose *Library*.

Click Next.



Choose target options. For the target options, first input a product name. I typically use the same name as the project with "*-docs*" at the end.

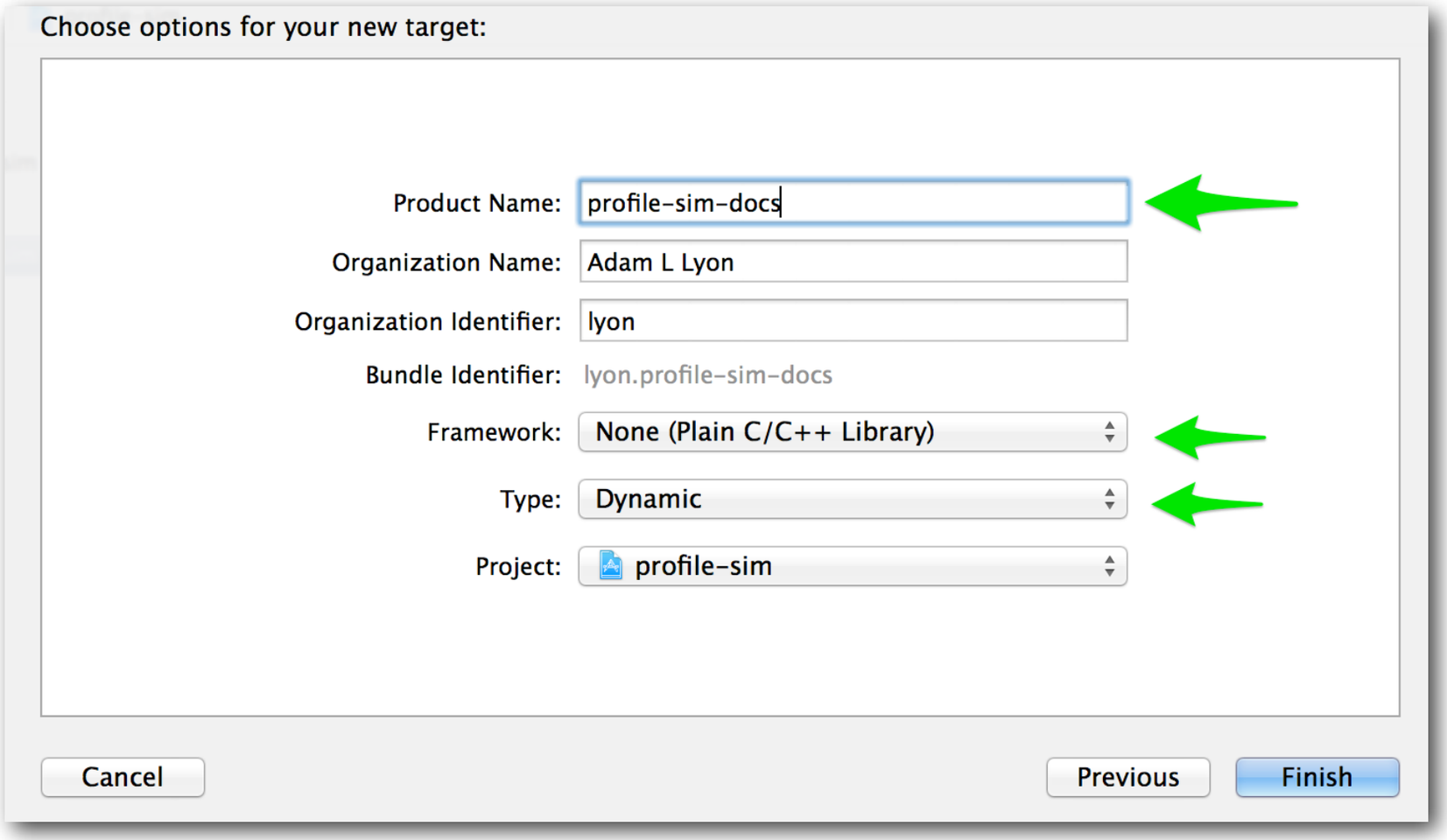
The organization name and identifier typically don't mean anything, so just put in your name.

For *Framework*, choose *None (Plain C/C++ Library)*.

For *Type*, choose *Dynamic*.

The *Project* should be automatically filled in.

Click Finish.

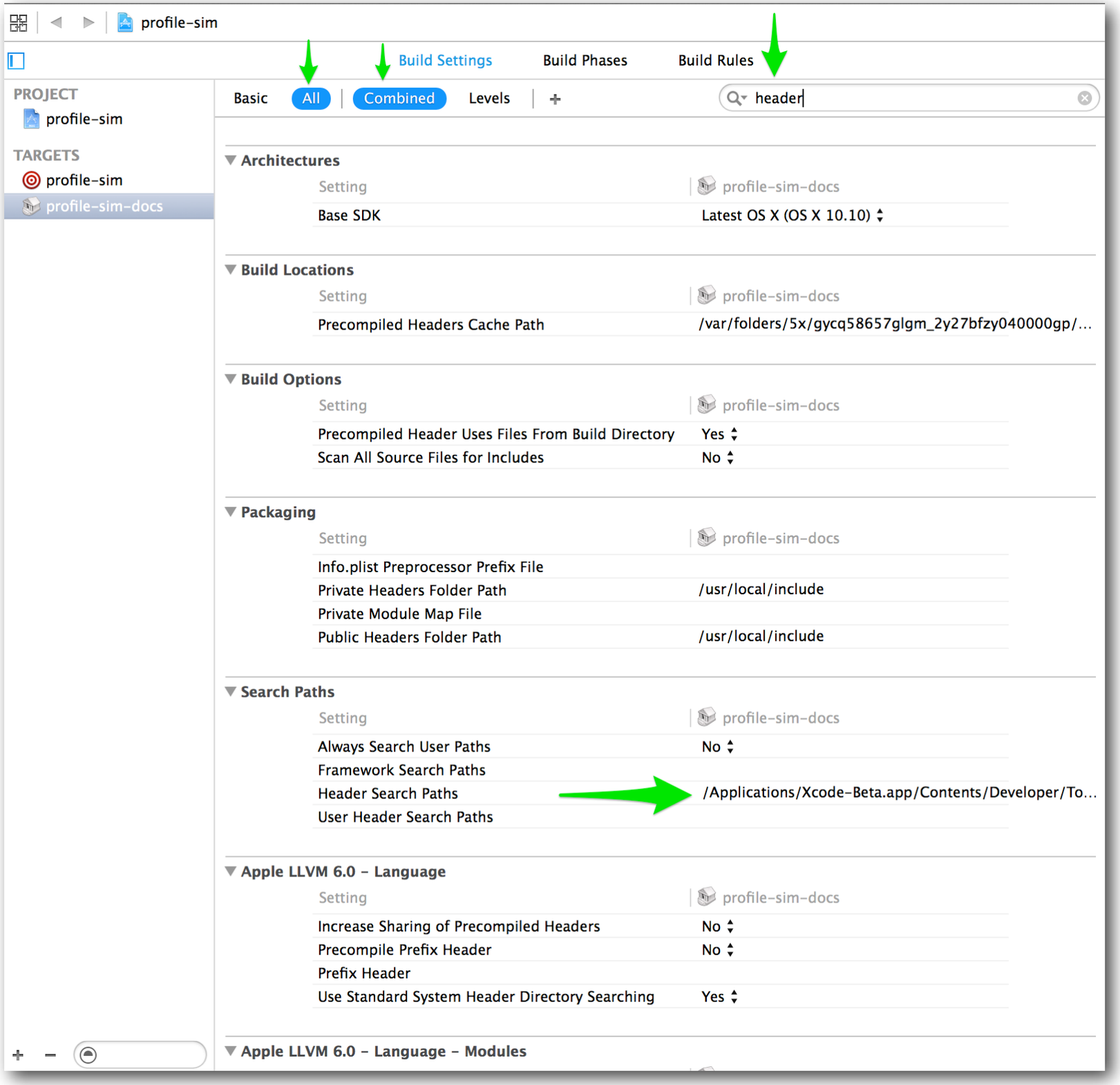


Now, we have to tell *Xcode* where header files for Art, Geant4, and other packages live. I have a little script that tries to make this easy.

Define the header search paths. You should now be back to the main *Xcode* screen with the target information displayed. Be sure that “All” and “Combined” are selected as shown on the right.

Now enter *header* in the search box. You should see a section appear called *Search Paths* and within that an entry called *Header Search Paths*.

Double click on the *value* of the *Header Search Paths*; that is double click on the words `/Application/...`



When you double click on the value of the *Header Search Paths*, a box should appear.

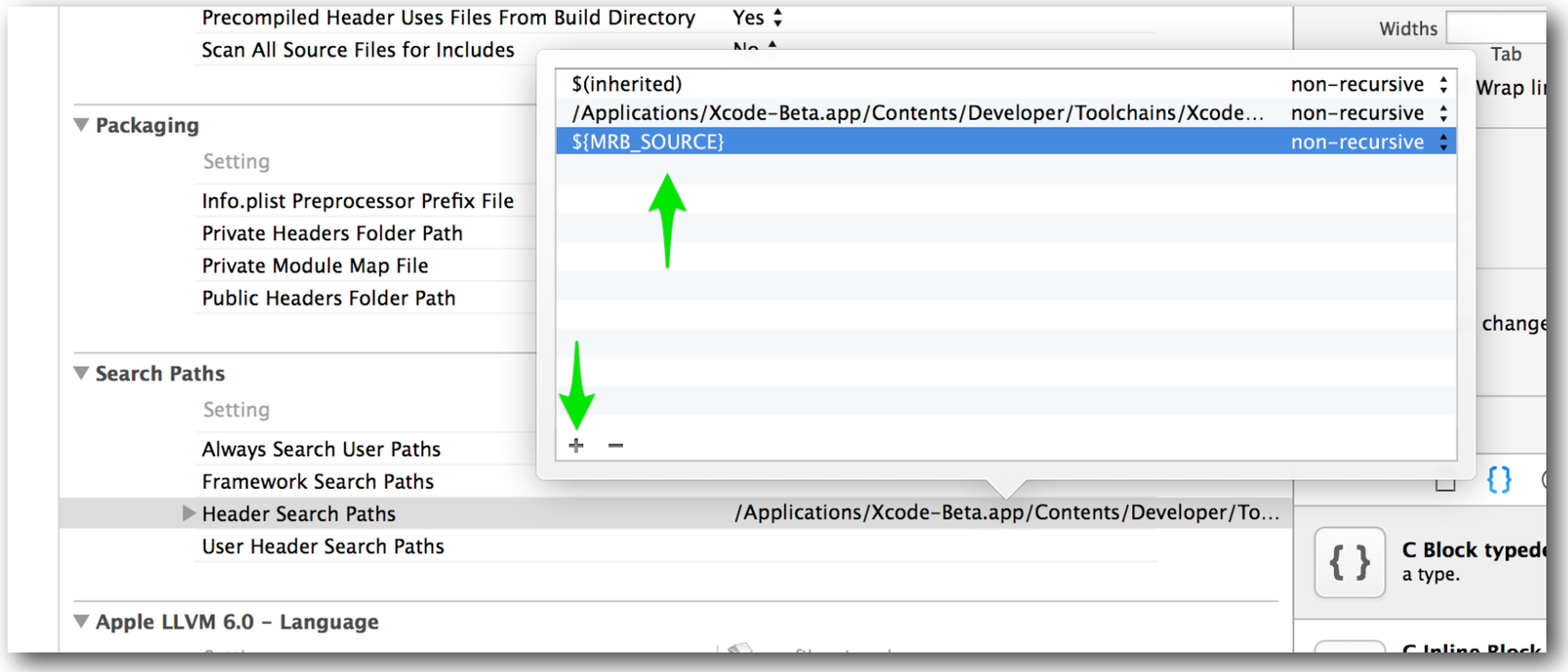
Click on the “+” in the lower left corner.

In the new entry box, enter `${MRB_SOURCE}` and press enter.

Click somewhere outside of the box to close it.

Now, back in your terminal window you used to start *Xcode*, enter the command on the right.

Now return to *Xcode*. You should see “No Editor” displayed on the screen.




```
$MRB_DIR/bin/xcodeIncs.sh  /path/to/xcode/project
```

```
# Example below
$MRB_DIR/bin/xcodeIncs.sh  ~/Development/g-2/xcode/profile-sim
```

Add sources to your project. The final step is to add the source code to your project. Control-click (right click) on the project name in the left hand bar and choose *Add Files to "<project name>".*

Or, you can also choose from the Menu bar, *File > Add Files to "<project name>"*.

First (because it is easy to forget), check the box next to your documentation target (as shown on the right).

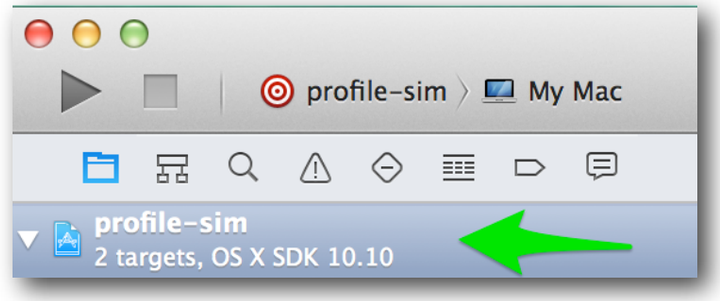
Now navigate to where your `srcs` directory lives and select it.

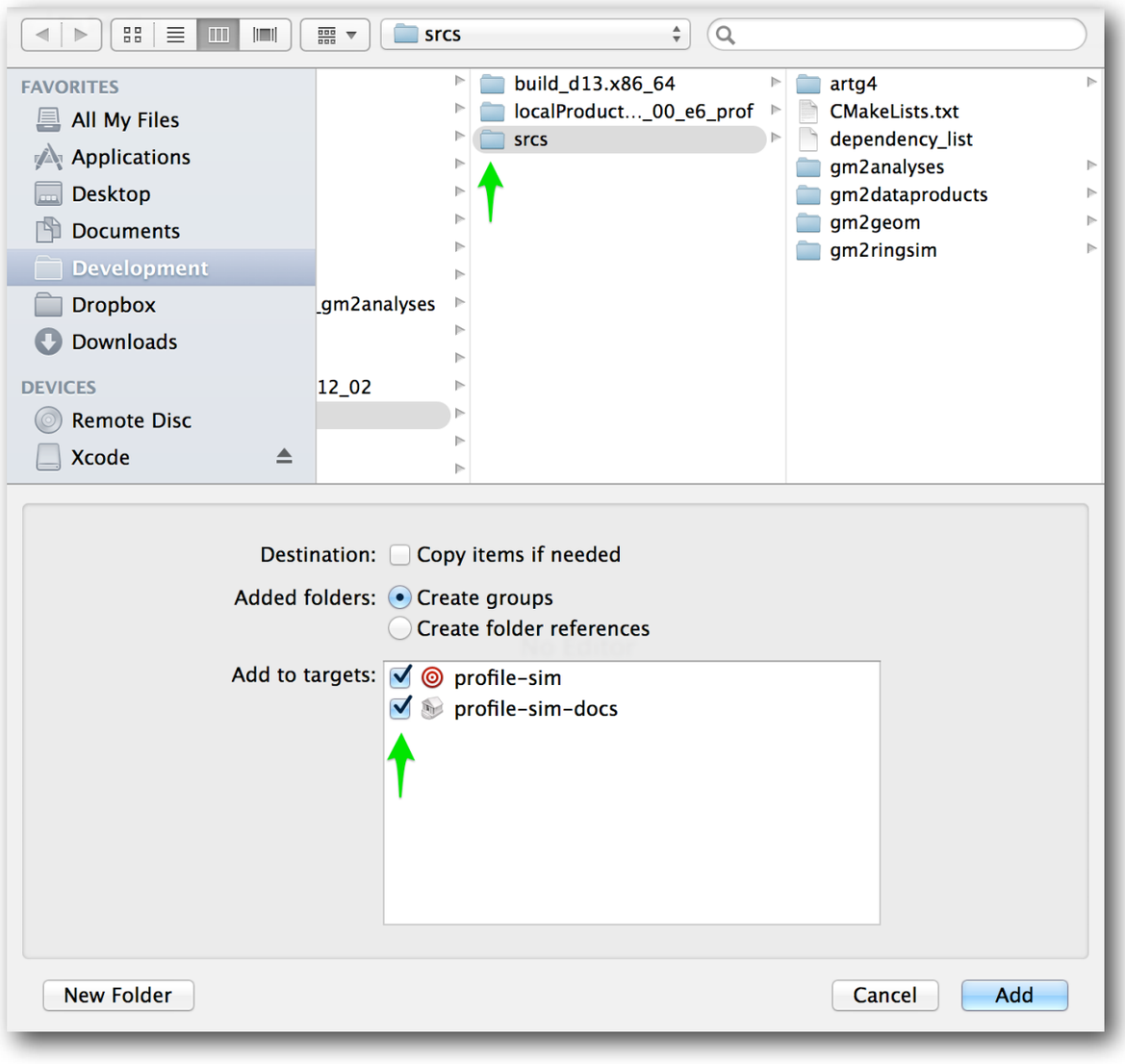
In the Project Navigator pane (on the far left) a *srcs* directory should have been added.

The activity area (center top) may say "Indexing"

Xcode configuration is now complete!

If you quit *Xcode* and start it again, you can get back to your project by selecting it on the right side of the welcome screen.





2.4 HOW TO LAUNCH INSTRUMENTS

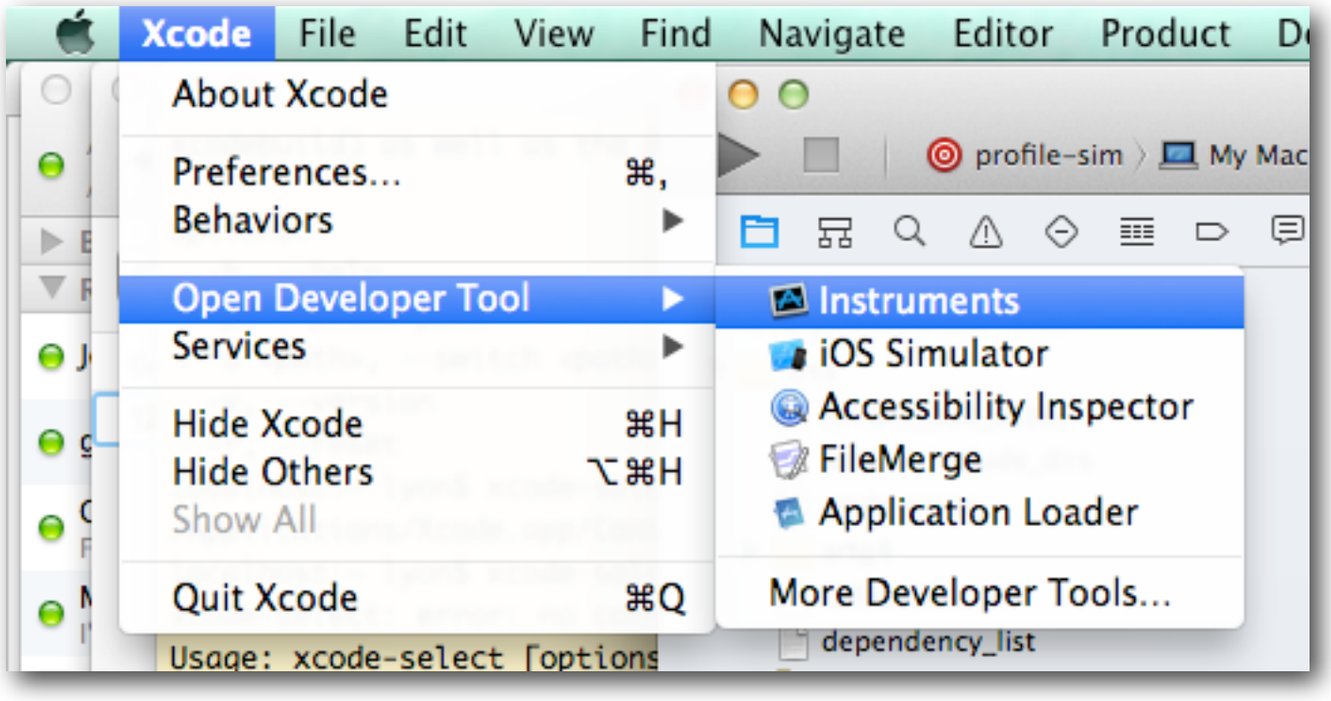
Run Xcode first, then select Instruments

Running *Instruments* is easy once you already have *Xcode* running. Follow the instructions in Section 2.2 for launching *Xcode*.

You may then launch *Instruments* from the menu bar by choosing *Xcode > Open Developer Tool > Instruments*, as shown on the right.

Again, you should not start *Instruments* by clicking on its icon. You need to start it within *Xcode* so that it will be in the same software environment as *Xcode*.

See later sections about how to run *Instruments*.



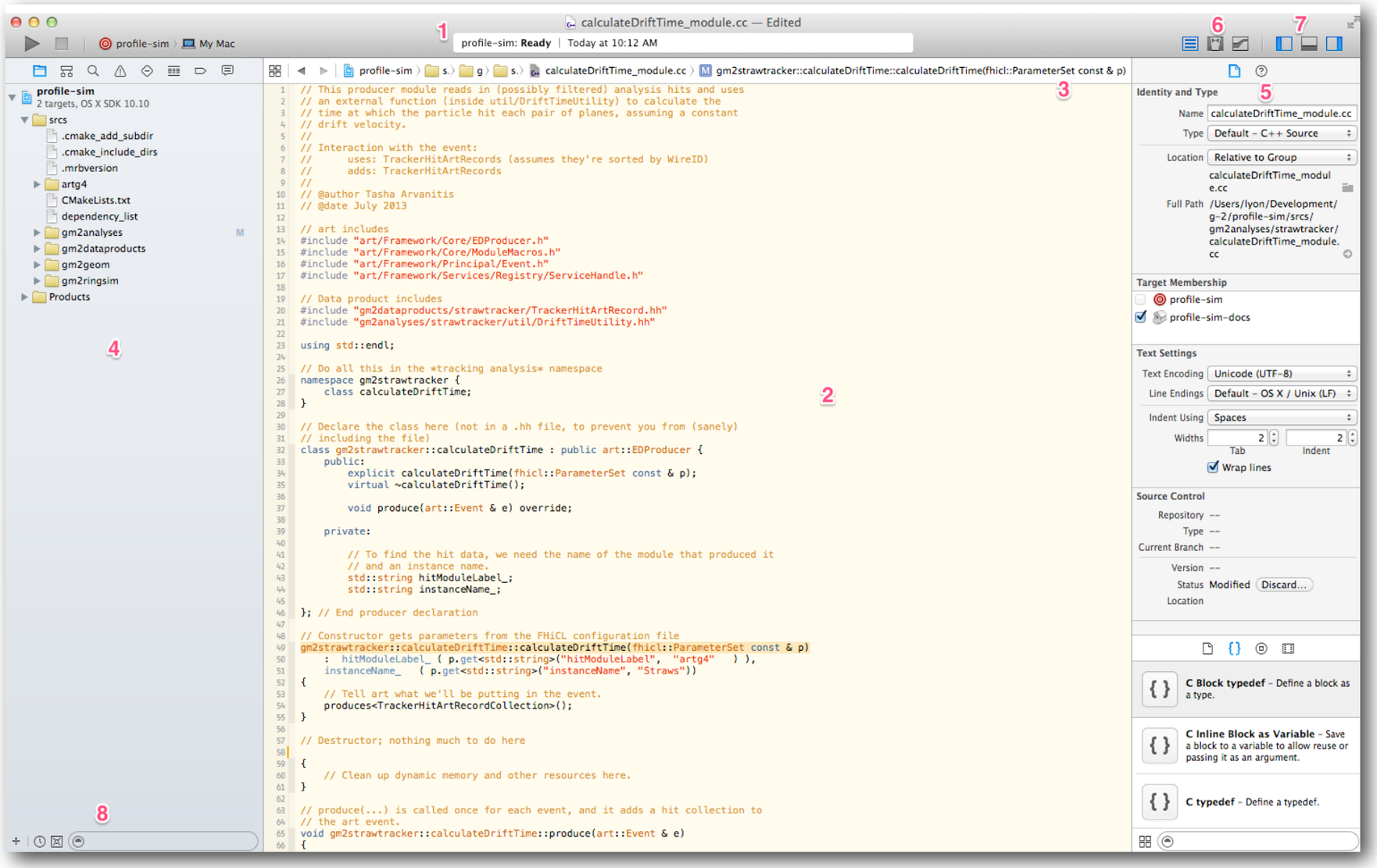
Using Xcode

- *Quick tour*
- *Navigating Source Code*
- *Search and replace*
- *Git Integration*
- *Build*
- *Debug*
- *More*

3

3.1 QUICK TOUR

The picture at the right is a general display from *Xcode* editing source. Refer to the numbers for explanations. Hovering the mouse cursor over something will often give you help on that item.



1) **Activity bar.** This bar shows any activity from *Xcode* (building, indexing, debugging)

2) **Standard Editor.** This is the main editor window for source code.

3) **Jump bar.** Clicking parts of the jump bar allows you to quickly navigate to different parts of the code or other files. You can filter by clicking and then typing. Clicking The left and right triangles allow you to cycle backwards and forwards through viewed code. The four square thing to the left of the triangles is *very* useful. It is a context sensitive list of associated files (e.g. the counterpart file, which is the .h or .cc).

4) **Navigator bar.** Currently showing the *project navigator*. Note the “M” indicates that a file in that directory was modified and has not been committed to git.

5) **Utility bar.** Currently showing the file inspector. I find this bar less useful.

6) **Editor chooser.** Brings up different editors. The version editor (right most) is very useful.

7) **Bar chooser.** Can make the navigator, debug, and utility bars disappear and re-appear.

8) **View restrictor.** Restricts what you see in the navigator view.

3.2 NAVIGATING SOURCE CODE

Xcode makes reading and exploring code very easy

Here are some tips for navigating in *Xcode*.

- *⌘Click (click while holding command key)* on an include file, a class name, or an object name will take you to the source for that include or class/object. *Option-⌘Click* will open an “assistant” editor instead of changing the standard editor. This feature is one of the best in *Xcode*.
- The *assistant editor* is an editor window that is associated with the standard editor and reacts to what you do in the *standard editor.* You can use the jump bar to configure the *assistant editor*. For example, choosing “counterpart” will show the .h or .cc file that corresponds to the file in the *standard editor*. You can have as many assistant editors as you want. You can configure where they initially appear in *View > Assistant Editor*. Assistant editors are very useful once you figure out how to use them (and are an unusual feature not found in other IDEs). There are also nice keyboard short cuts. A handy one is *⌘Return*, which will close all of the assistant editors you have open leaving you with only the *standard editor*.
- To edit multiple files in *tabs*, you use *⌘T* to open a new tab. The new tab will initially be identical to what you just had open. Use *⌘{* and *⌘}* to cycle through tabs.
- Play with the *version editor* to view different git versions of the source files and their differences. This feature is extremely useful.
- *Right (or Control)-Click* on text in an editor brings up a wealth of actions. The *Show Blame for Line* will show you who entered/last changed that line and the git commit.
- Remember to use the *jump bar* to quickly move around the file.

3.3 SEARCH AND REPLACE

Searching and replacing code at the file or project level is easy

One of more important features of an IDE is finding text as well as searching and replacing. *Xcode* is extremely capable in this area

Many actions are available from the *Find* menu. To do file level find/search & replace do ⌘*F*.

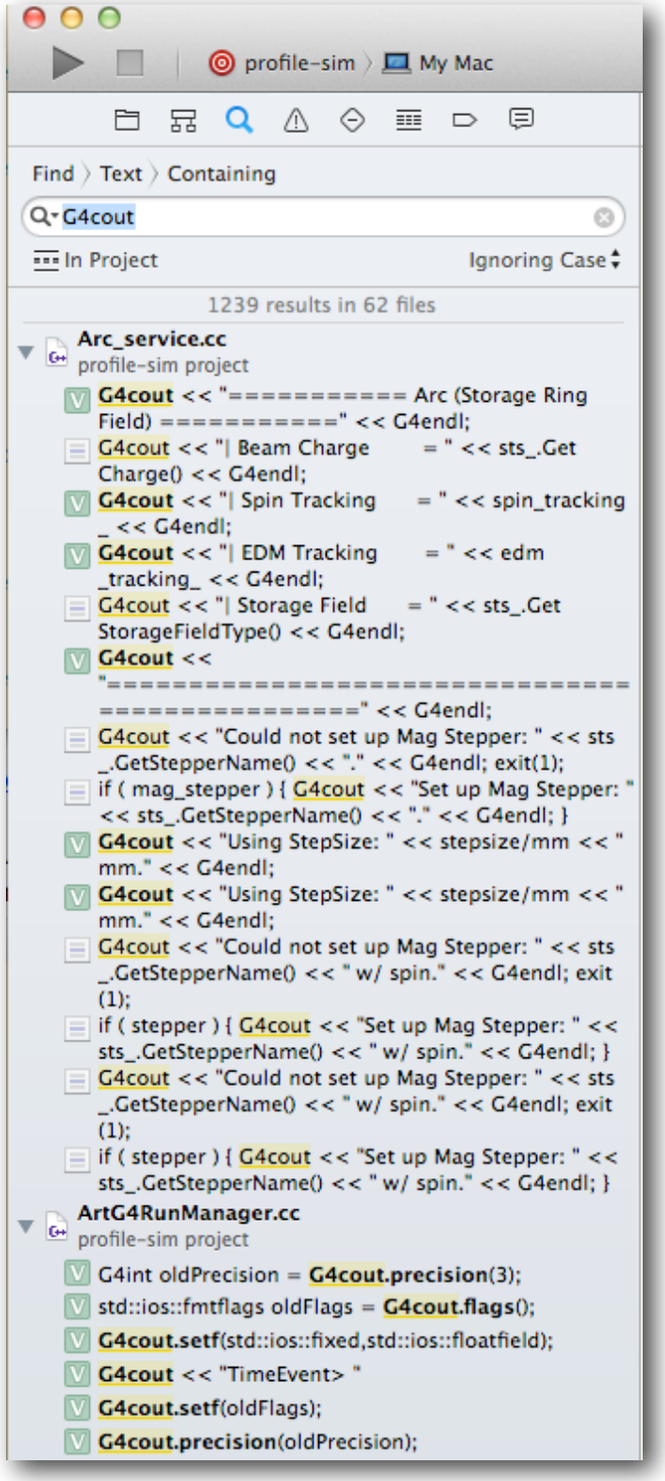
Project level find/search & replace is especially useful. To activate the *Find Navigator* select the search icon on top of the navigator bar (as shown selected on top in the figure on the right).

You may now enter the search string in the text box. Clicking on *Find*, *Text*, *Containing*, and the search icon in the bar brings up a wealth of options (click *Find* to see replace actions). There are also more options below the text bar, including if you want to restrict the search to certain files.

The results are shown below. Clicking on a result will bring it up in the standard editor (option-click will bring up an assistant editor).

You can remove results by clicking on them and pressing *Delete*. Doing so does not change any file, it simply removes the result from the *Find Navigator*.

Many search and replace options and actions are available here.



3.4 GIT INTEGRATION

Xcode has extensive integration with git

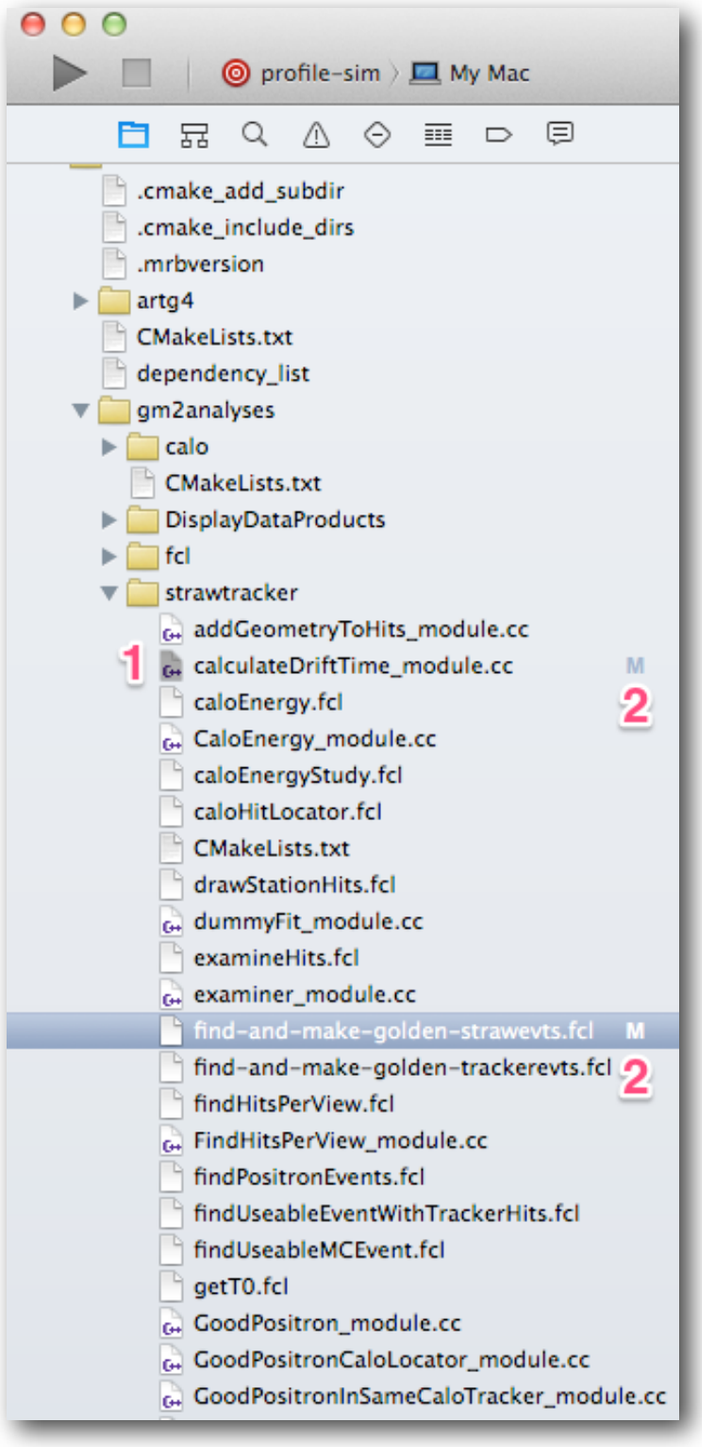
Git indicators. The *Project Navigator bar* gives several clues to the state of files in git.

1) Files that have been modified, but unsaved, have their file icon in grey.

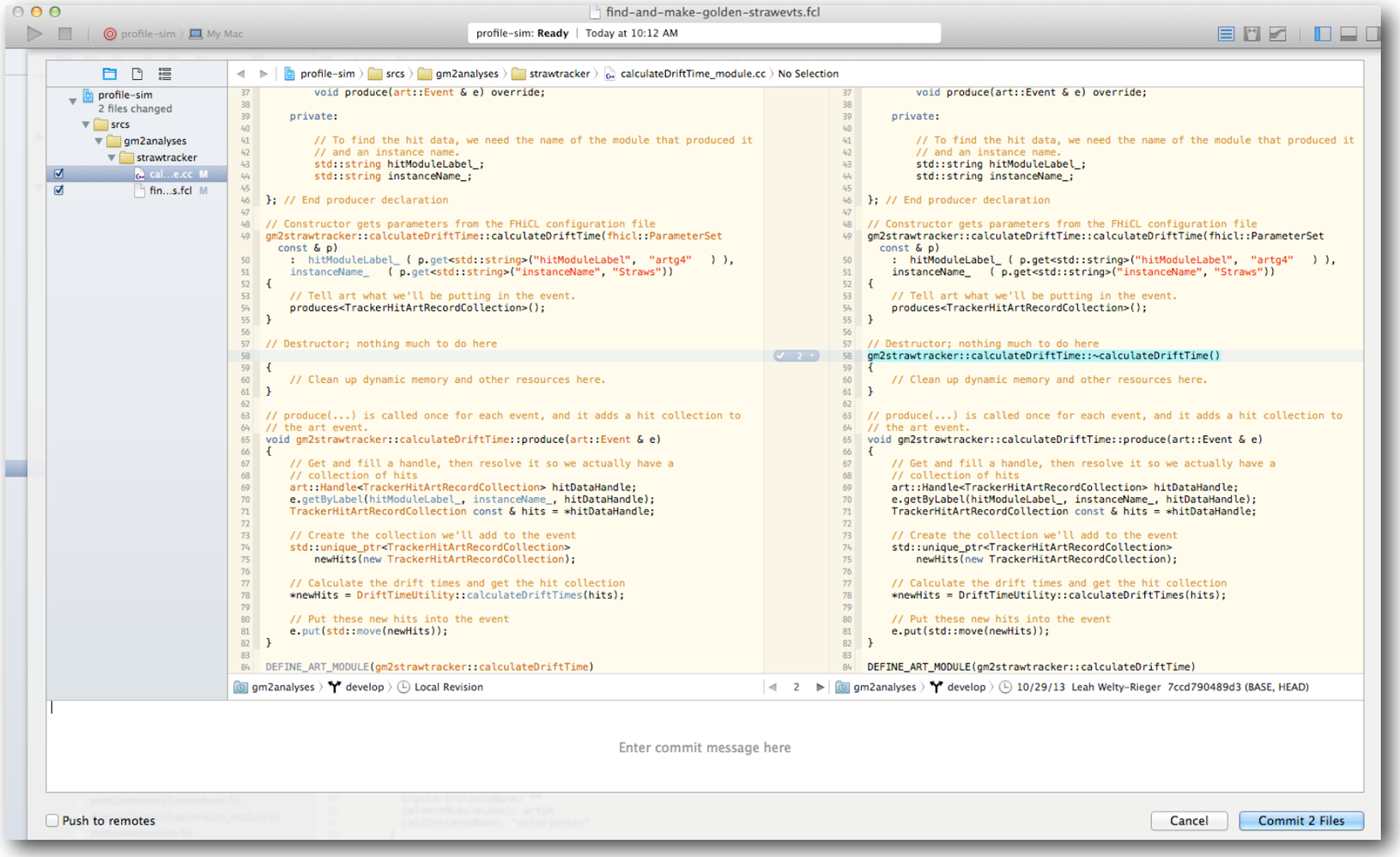
2) Files that have a git status are indicated by a grey letter in the right side of the bar.

   - *M* file was modified
   - *A* file was added
   - *D* file was deleted

Such files have not been committed yet.



Checking in commits. When you have modified files, you may commit them to your local git repository. You can commit individual files by *Right (or Control)-Click* on the file in the *Navigator bar*. You can commit all of the modified files by choosing from the menu *Source Control > Commit...* A sheet similar to the one on the right will appear.



On the commit sheet you will see, on the left, a navigator bar showing the changed files to be committed. Clicking on the checkbox will add/remove that file from the commit. The main part of the sheet shows a *version editor* where you can see the changes you made for a particular file. You may remove particular changes from the commit by clicking on the center-column indicator. Scroll up and down to see all of the changes.

You type your commit message in the block on the bottom of the sheet.

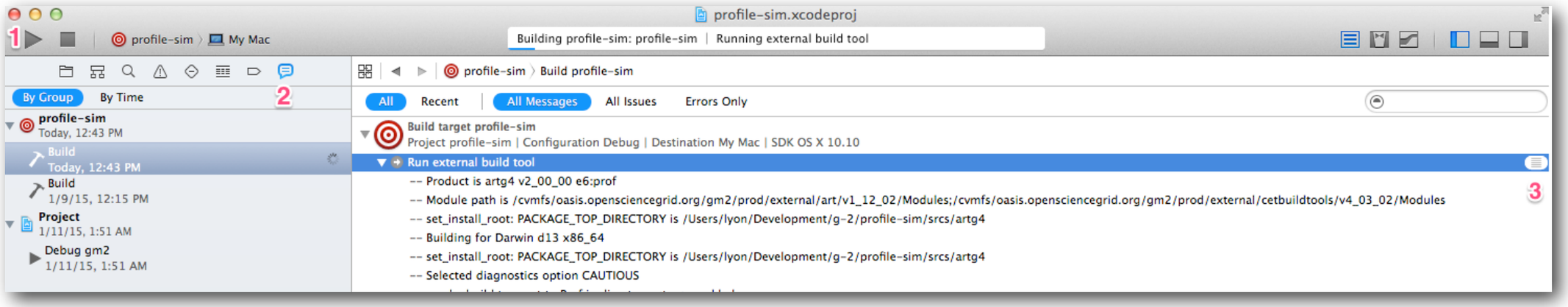
If you want to push to remote repositories (Redmine) at the same time as committing, check the *Push to remote* box in the lower left. Be sure you have an active kerberos ticket to push.

Explore the *Source Control* menu for many more actions and options.

Merging is particularly useful in *Xcode*, because a very helpful merge conflict editor will open if necessary.

3.5 BUILD

Xcode's build error display is very useful



If you have configured *Xcode* correctly, you can build the code by clicking on the big right triangle in the upper left hand corner (1) in the figure above.

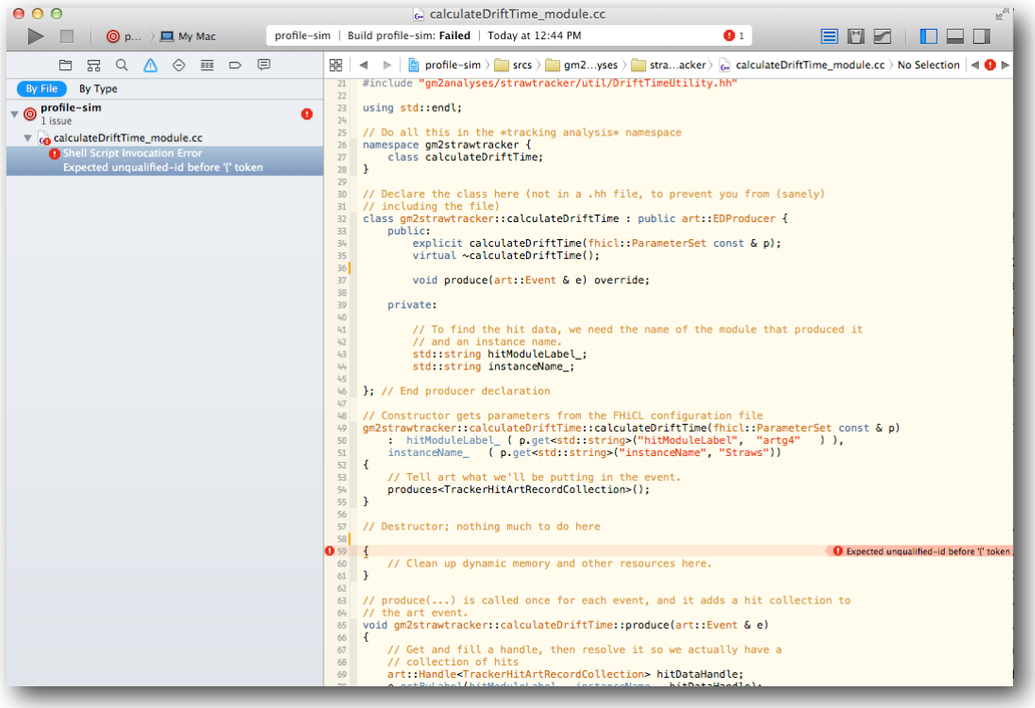
To see the build log, click on the *Log Navigator* (2).

The build log shows a subset of the log output and is not particularly useful. To see the entire log, select the little icon as shown in (3) on the same line with "Run external build tool". Look at the green shaded output.

You will also note that the *Action Bar* indicates that the build is in progress.

Errors will be indicated in many places, including as an icon the *Action Bar*. Clicking on that icon will take you to the *Issue Navigator*. Clicking on an issue will show you the error in the source code (see right). This feature is extremely useful.

If the build is successful, you can run gm2 from the command line (not from within *Xcode*).



3.6 DEBUG

Debugging within Xcode can show you how your code works

Some notes. Debugging our *g-2* code on a Mac has limitations. Apple does not use gdb for debugging. Rather, it uses LLDB, which is tied to the LLVM compiler (we use gcc instead). Many things work, but somethings don't and LLDB can crash if you view certain objects. Simply restart *Xcode* and you'll be running again.

Another limitation is that on the Mac, connections from code to the source are not stored in libraries. Therefore, if you want to debug into source code, you *must build that code yourself*. Typically, this is not a problem as Art and Geant code are complicated and do not lend themselves for easy debugging. If you are debugging and see assembly code instead of C++ source, that is because the connection to the source is not available. All code you build yourself is debuggable.

Lastly, debugging code built in a *prof* release will work, but stepping through code will seemingly be crazy as the current line pointer will jump around. Furthermore, break points that you set may never get hit. This problem is because *prof* builds are optimized. Code you see in the source file may have been altered or removed by the compiler optimizer.

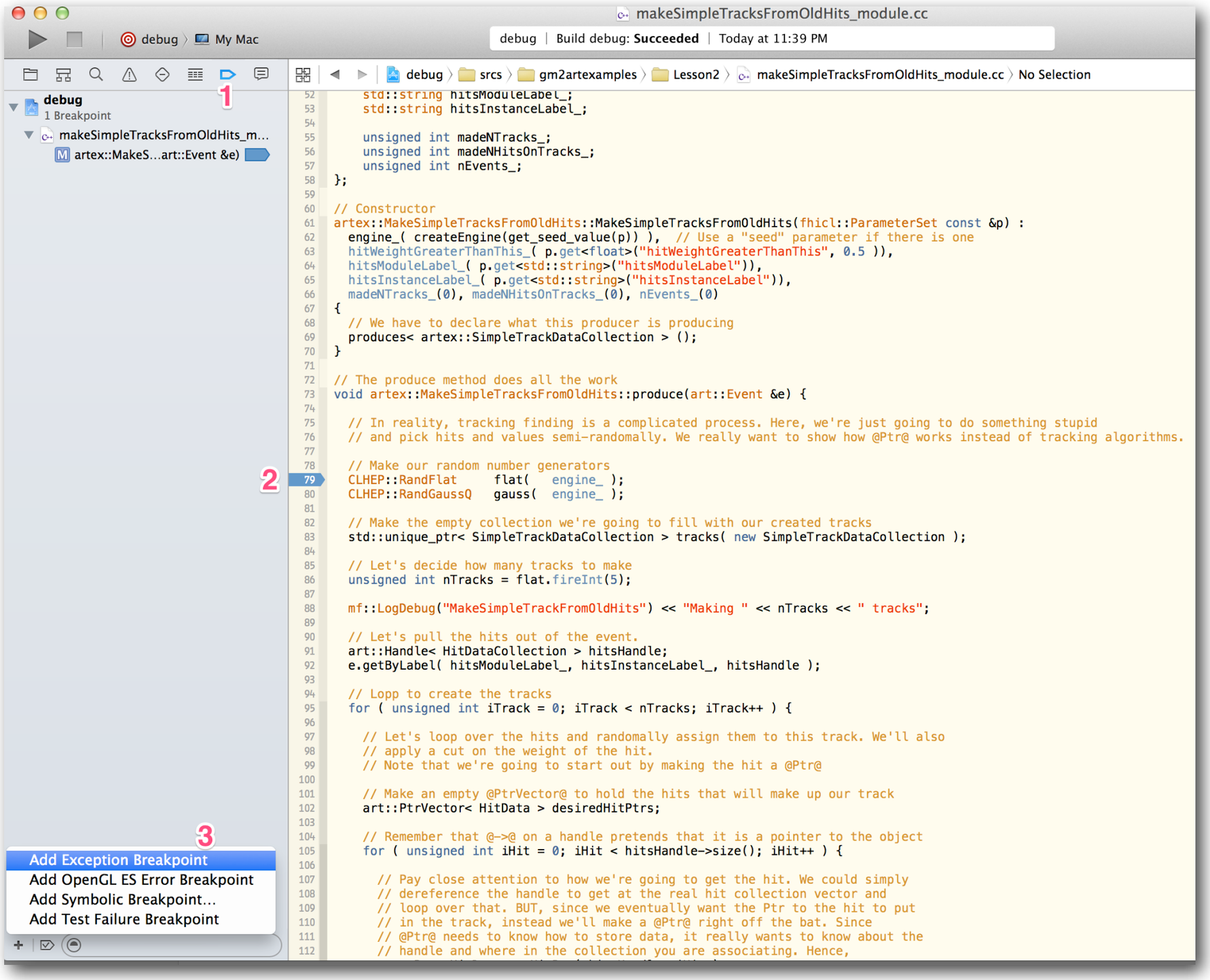
If you are exploring the operation of some code, you should use a *debug* build (no optimization).

To set breakpoints you can simply click on a line number in the source (2).

You can also click on the *Breakpoint Navigator* (1) and see a list of set breakpoints.

You can right (or Control-) click on the break point and edit it.

At the bottom of the navigator you can also set special breakpoints, such as exception breakpoints. Those are especially useful if Art or your code is throwing an exception.



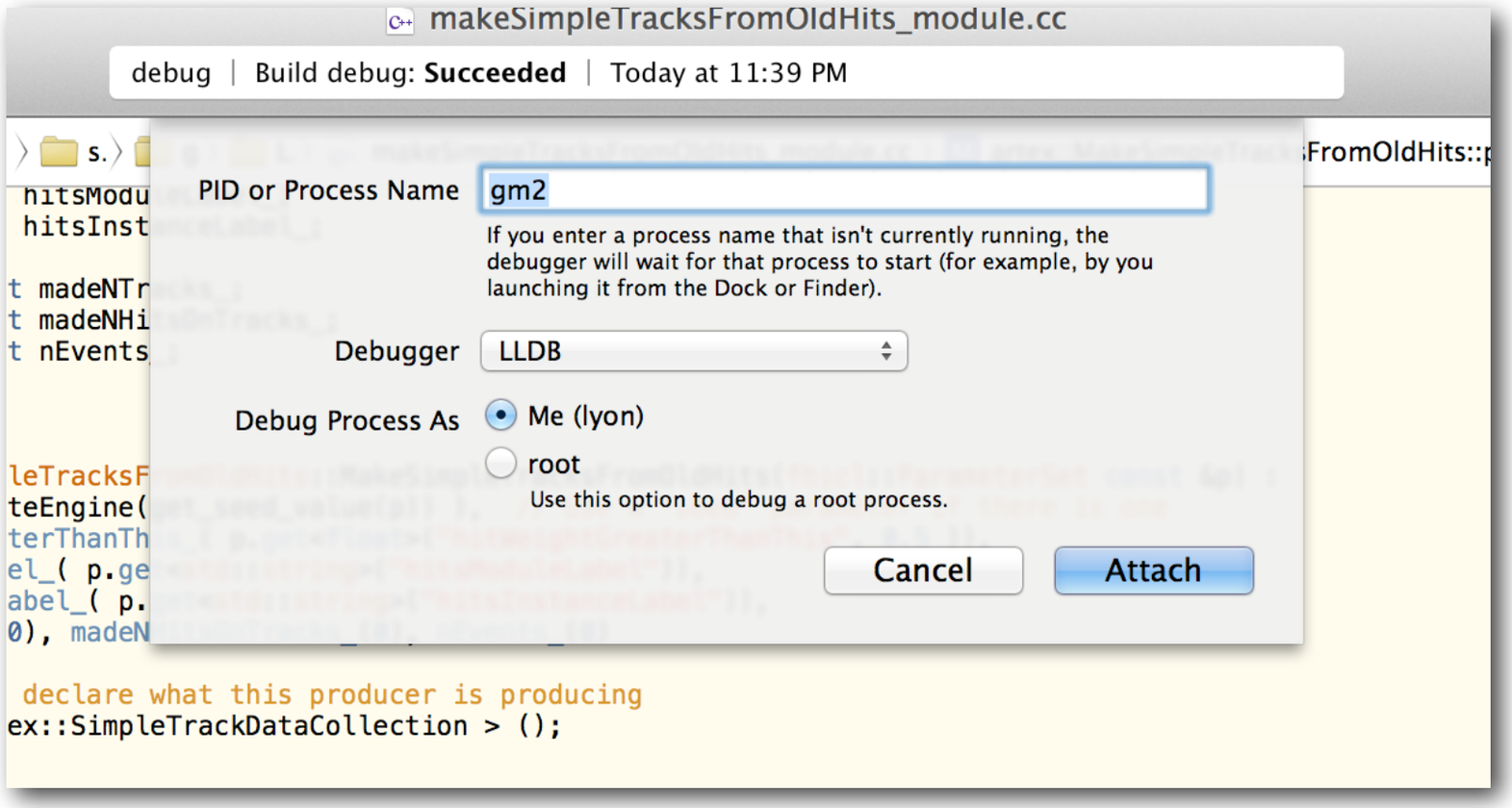
To run the debugger, you must attach to the running process. You can set up Xcode by choosing from the menu *Debug > Attach to Process > By Process Identifier (PID) or Name...*

Enter *gm2* for the process name. Then click on Attach (*Xcode* will wait for the process to start).

The first time you try to attach, *Xcode* may suspend itself. Go to the terminal and do `bg` to resume *Xcode* in the background.

You should see *Waiting for gm2 to Launch* in the activity bar.

Then start *gm2* in the terminal. For example,




```
gm2 -c makeTracksFromOldHits.fcl
```

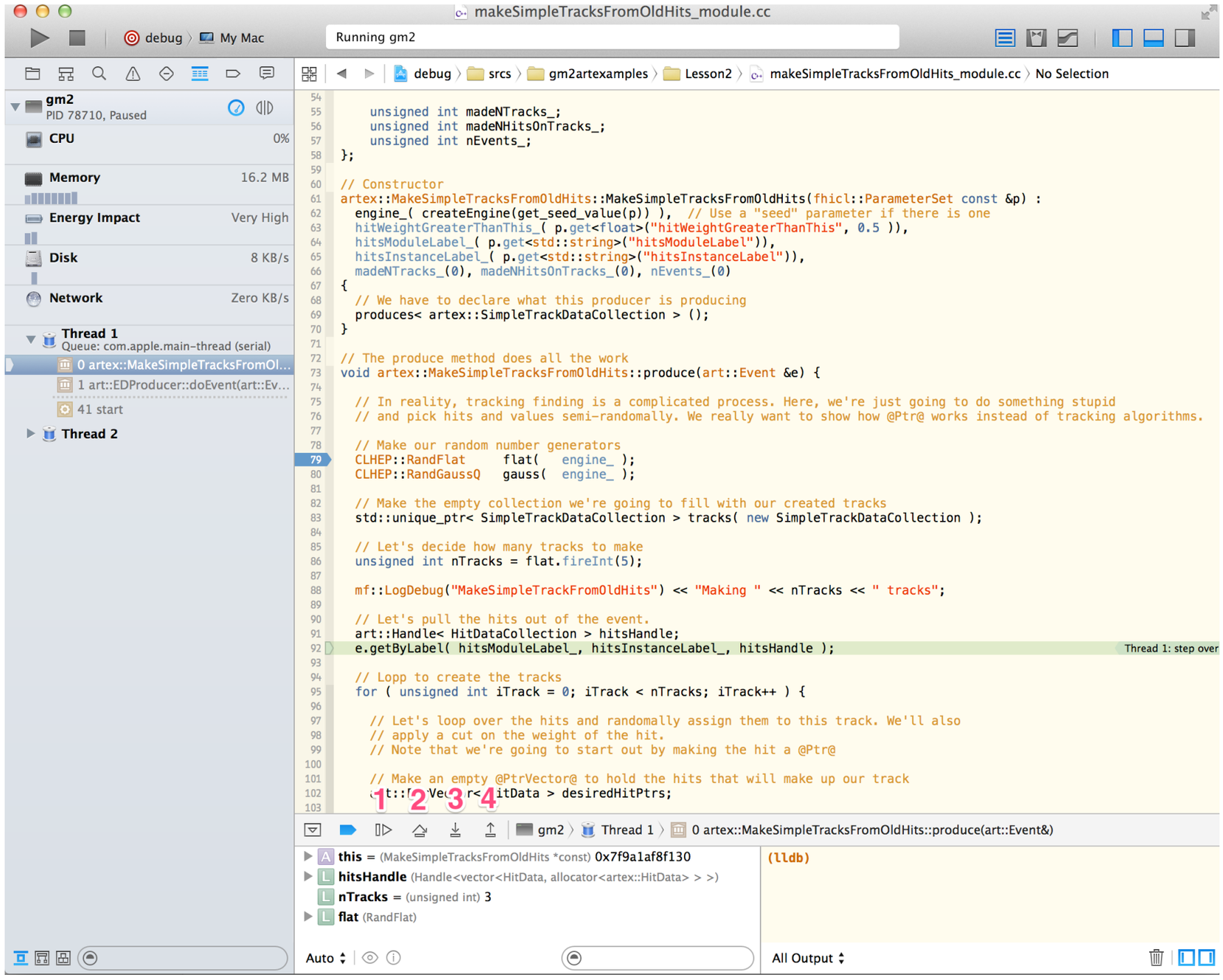
The debugging screen is shown here.

You can advance through the program by continuing (1), stepping over functions (2), stepping into functions (3), and stepping out of the current function (4).

You can view current variables towards the bottom of the window.

You can also hover the mouse over objects and variables in the source code to see values.

If you see assembly code instead of source code, then you are looking at code that you didn't build.



Using Instruments

- *Configuring Instruments*
- *Time Profiler*
- *Memory Allocation and Leak Profiler*

4

4.1 CONFIGURING INSTRUMENTS

Configure Instruments to run "gm2"

See section 2.4 for how to launch *Instruments*.

When it starts up, you will see a screen like the one on the upper right, allowing you to select a profiling template for an executable.

Your screen will have some default executable. Click on it to bring up the chooser, select your Mac laptop as the device, and select *Choose Target...*

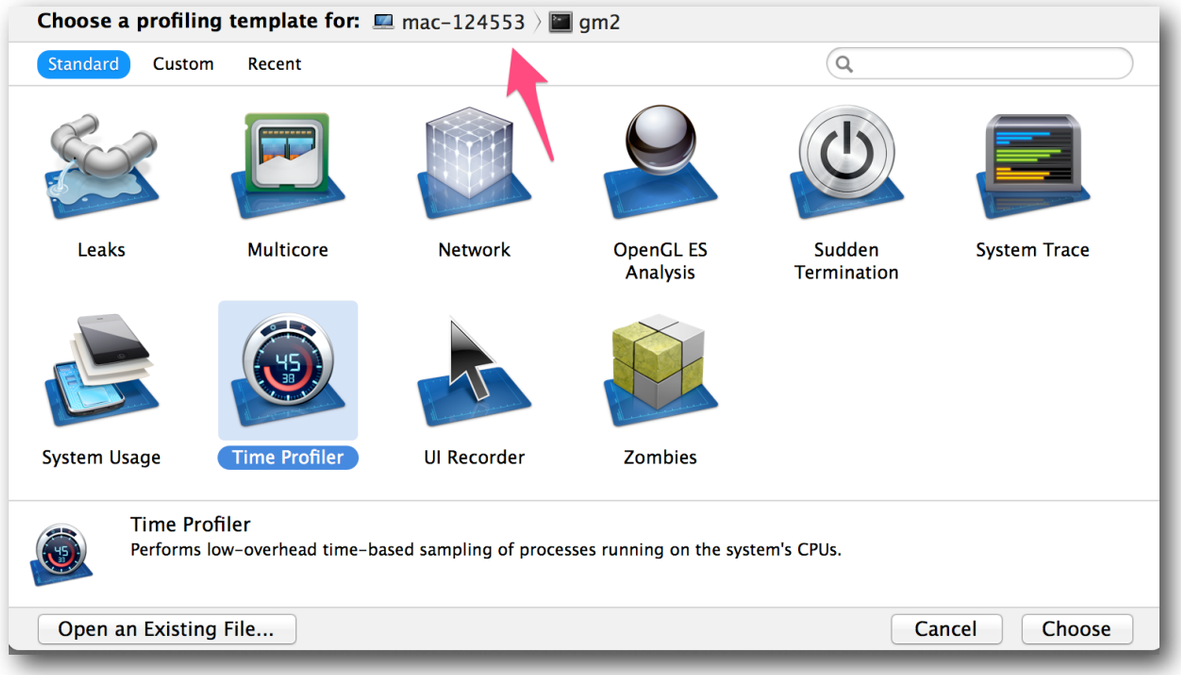
You now need the location of the *gm2* executable. In the terminal where you ran *Xcode*, do

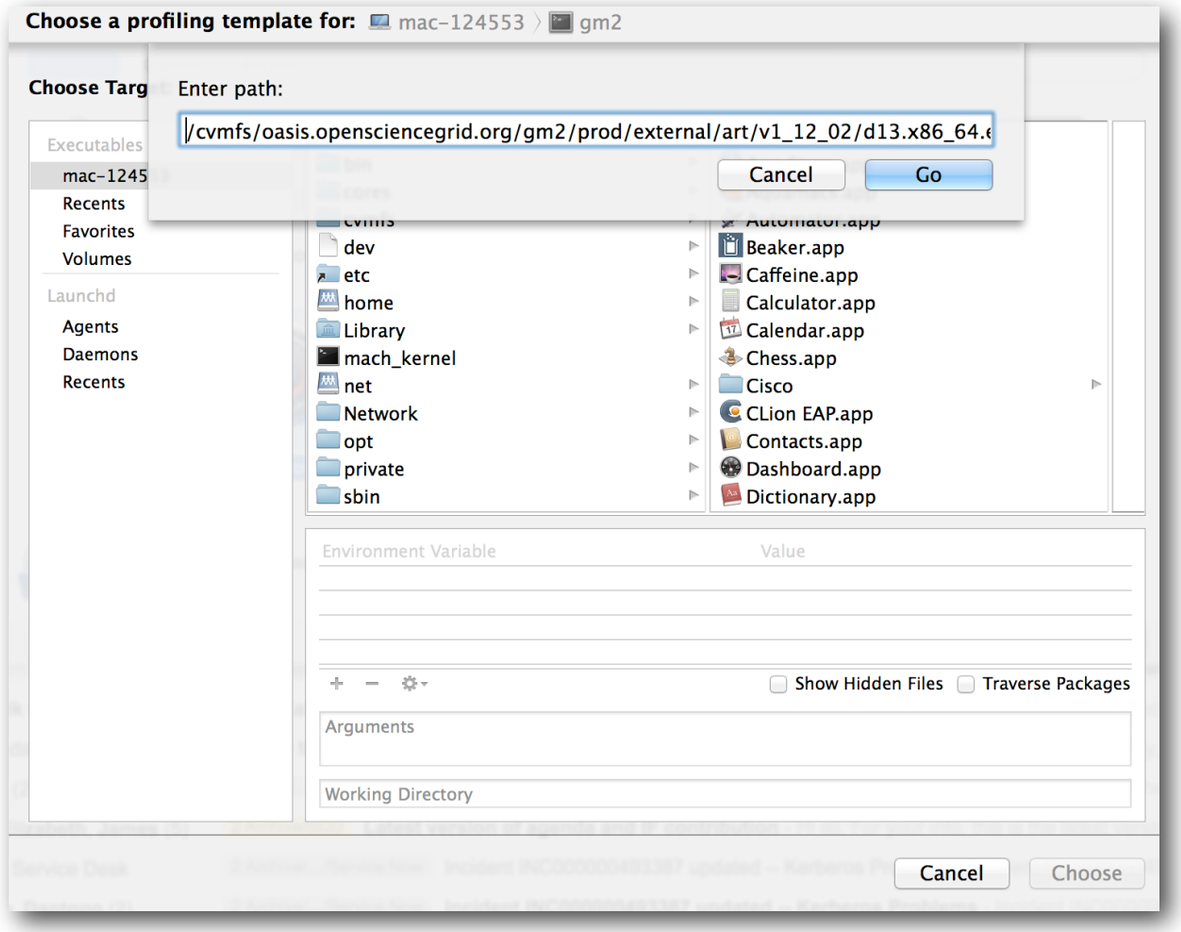
```
which gm2
```

Copy the results to the clipboard with ⌘*C*.Then go back to the instruments window and press *Shift*-⌘*G* to open a path box. Paste the results with ⌘*V* and press Go. (Note this path box works for any file open window in any Mac application).

Next, choose the the *gm2* executable.

See the next page.





Now you need to fill in the *Arguments* (1). Type in the arguments you want to pass to *gm2*. Let's run 2000 events through one of the standard fcl files in *gm2analyses*.

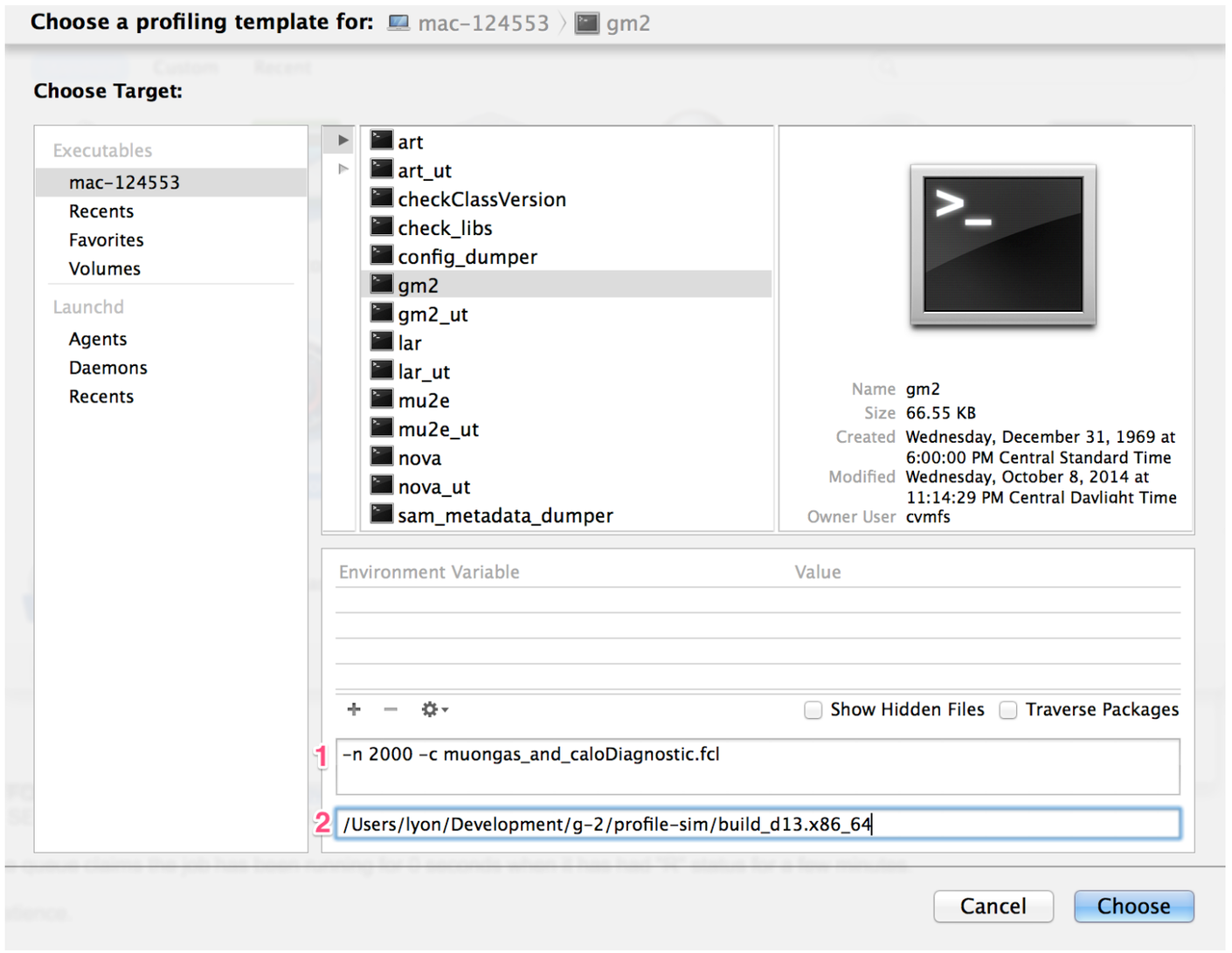
```
-n 2000 -c muongas_and_caloDiagnostic.fcl
```

You also must fill in the *Working Directory* (2). If you want to use your build area, then go to your terminal window and do

```
echo $MRB_BUILDDIR
```

Copy the results to the clipboard with ⌘*C*.Then go back to the instruments window, click in the *Working Directory* box and paste the results with ⌘*V* (unlike in *Xcode*, environment variables do not seem to work here).

Press Choose.



4.2 TIME PROFILER

The time profiler shows you where your program is spending its time.

Choose the *Time Profiler* from the template chooser.

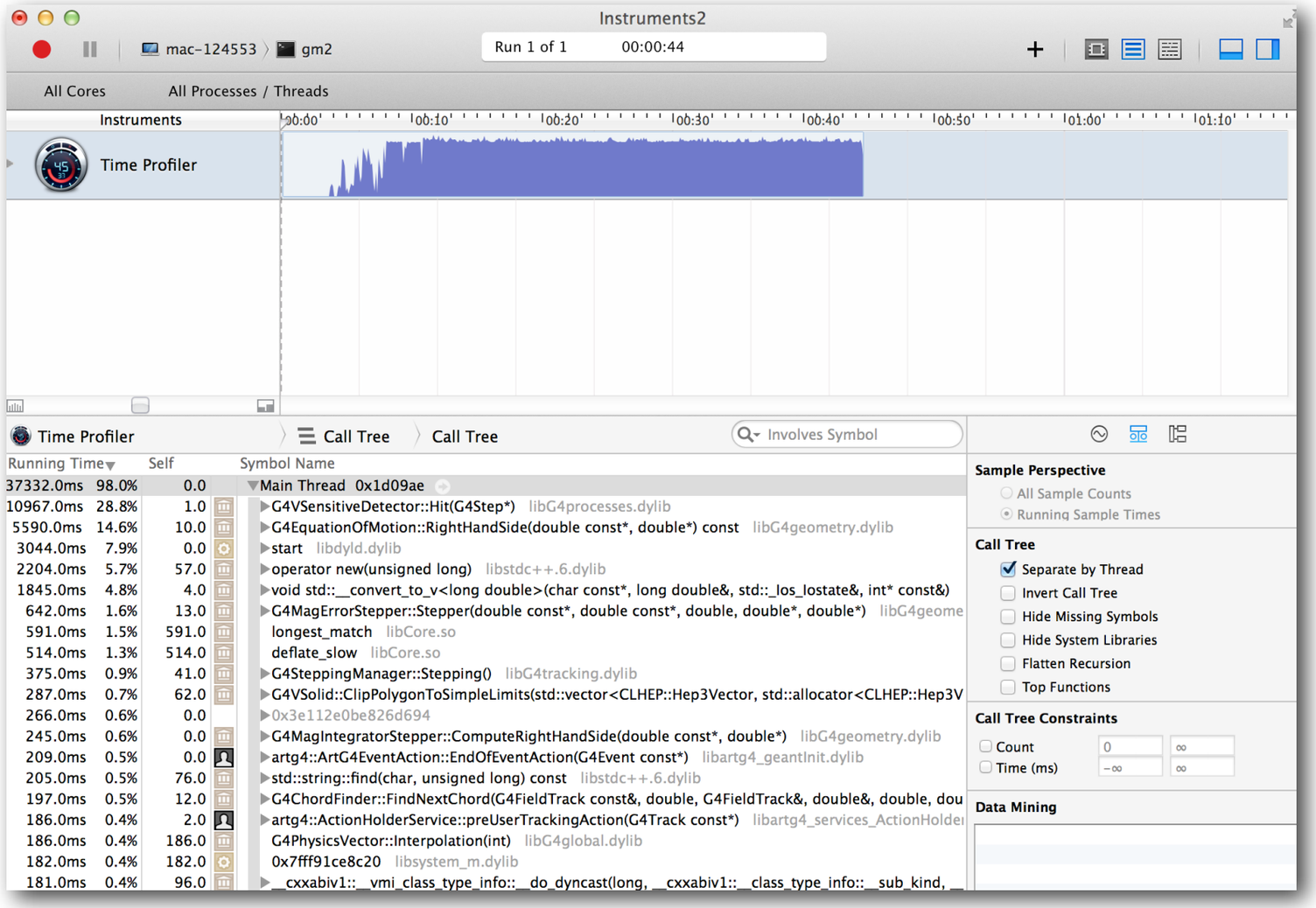
Run the profiler by clicking the red record button. The example given here takes about 40 seconds to run.

The result is shown on the right.

The top part shows the CPU usage over the course of the application.

The bottom shows the detail of the execution.

You can restrict what is seen in the detail by clicking and dragging over the timeline.



The Time Profiler works as follows:

When the executable is run within *Instruments*, the *Stack Trace* is sampled every millisecond.

The Stack Trace is the routine the program is running at a given time along with the traceback (what functions were called to get to that routine).

The samples are collected and displayed in the detail area of *Instruments*.

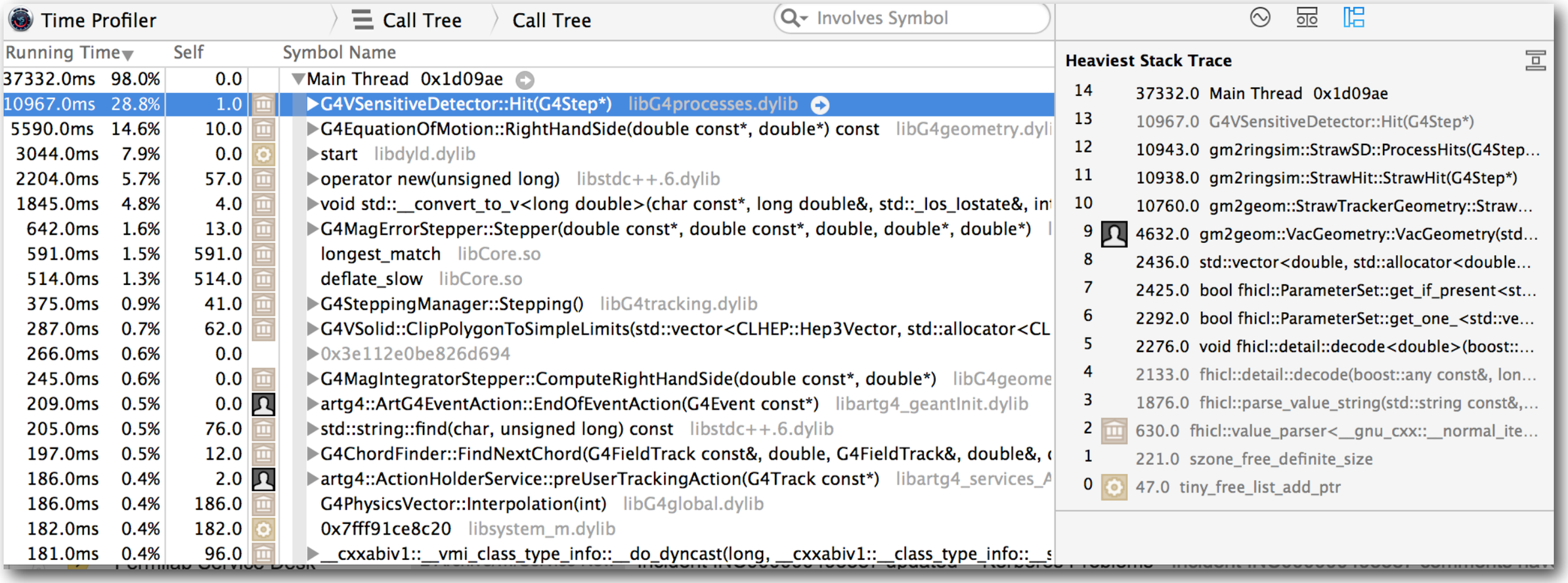
The *Call Tree* shows what functions were active when the samples were taken. Functions that appear often are interpreted to take some fraction of execution time.

For example, if a function is listed in 1000 samples, then one can interpret that the cpu is dealing with the function for 1 second of CPU time (if each sample is 1 ms).

This sampling yields some limitations. If a routine is extremely fast, then it may not show up in the profile. This is ok, since one is worried about routines where the CPU is spending a lot of time.

It is often not possible for *Instruments* to determine the entire stack trace for every sample, so you may seen truncated traces.

Let’s go though the detail in this example to learn some things about the *g-2* simulation.

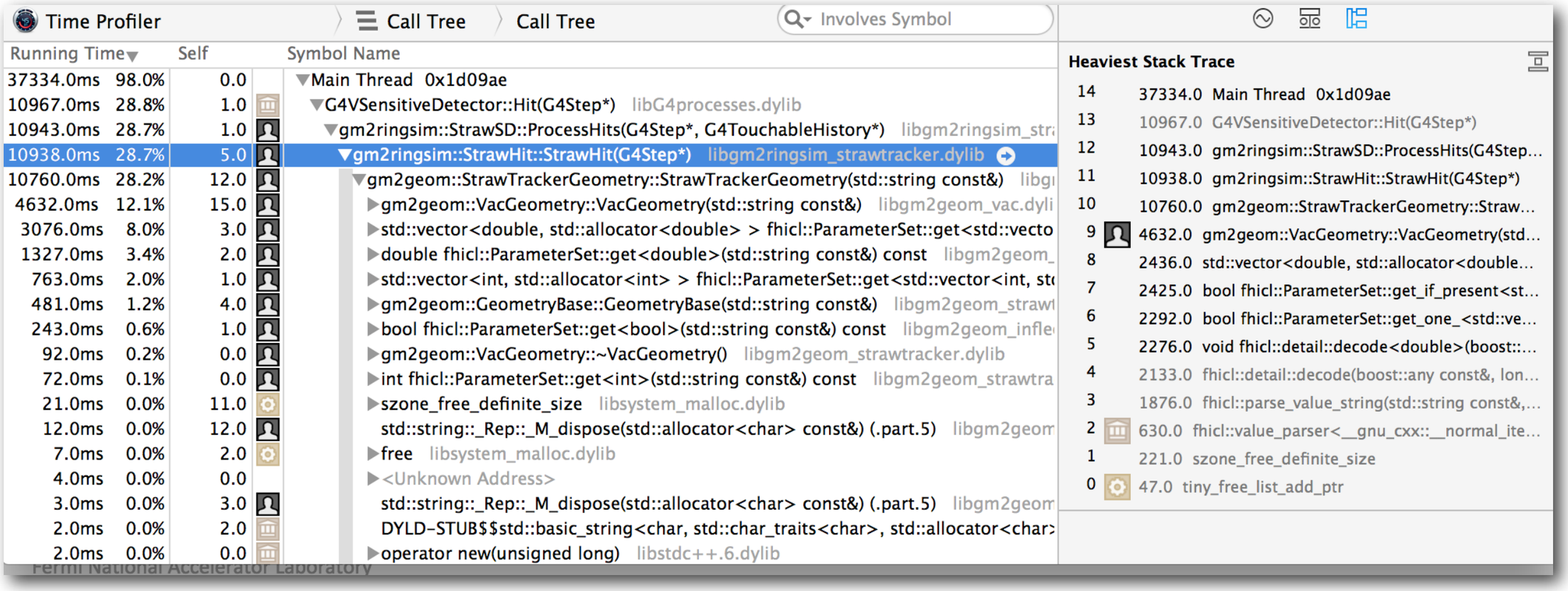


In this display, the *Running Time* shows the total time the CPU is in that routine or a function the listed routine calls.

Self shows how many samples that were taken where the routine listed was at the bottom of the stack (the routine the CPU was actually running).

The symbols are **not** a stack trace. It is a list of routines found in the stack traces. If you select one, on the right side you can see the stack trace that appeared in the most samples. The bottom of the trace is what the CPU was running when the sample was taken. Moving up the trace, you see the routines that were called to lead to that function. This particular trace is truncated.

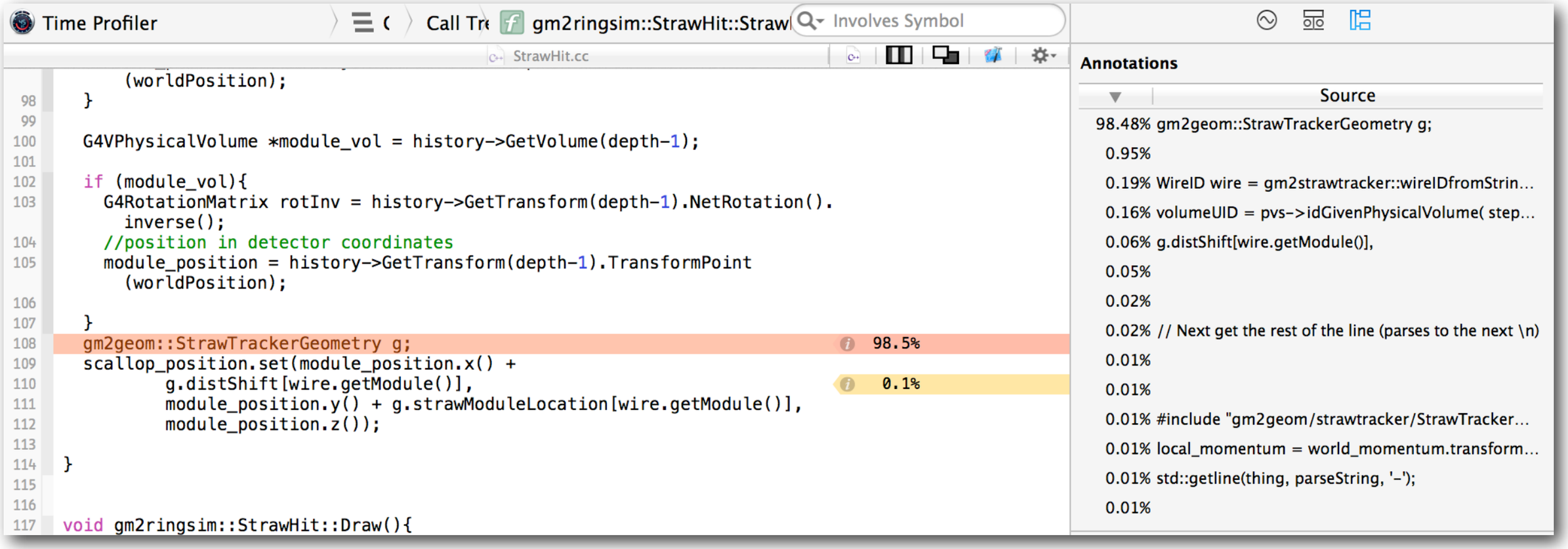
Let's look at the detail window. One thing that stands out is that `G4VSensitiveDetector::Hit` appears in ~30% of the samples. This doesn't mean that the CPU was running that particular routine 30% of the time. Instead it means that in 30% of the samples, the routine at the bottom of the stack trace (the one what was running) was called by this particular routine (with perhaps many routines in between). We will dive into this situation on the next page.



Opening the exposure triangles, you see more routines. These are the routines that are called by the routine at the higher level (outdented). Again, this is not an explicit stack trace. It is a list of routines at the next level of depth in the stack trace with their timing information.

What we see is that 30% of the execution time is in the `StrawHit` constructor. And that is because the constructor calls the `StrawTrackerGeometry` constructor. Let's look at this code to see what is going on. Double clicking on the highlighted line gives the source code.

See next page.



Here we see that for **every** tracker hit, the `StrawTrackerGeometry` is retrieved. And that appears to be an expensive operation, taking up nearly all of the CPU time for this routine.

A good solution would be to cache the geometry somewhere. Doing so could potentially save a large amount of execution time.

4.3 MEMORY ALLOCATION AND LEAK PROFILING

You can watch memory usage over time.

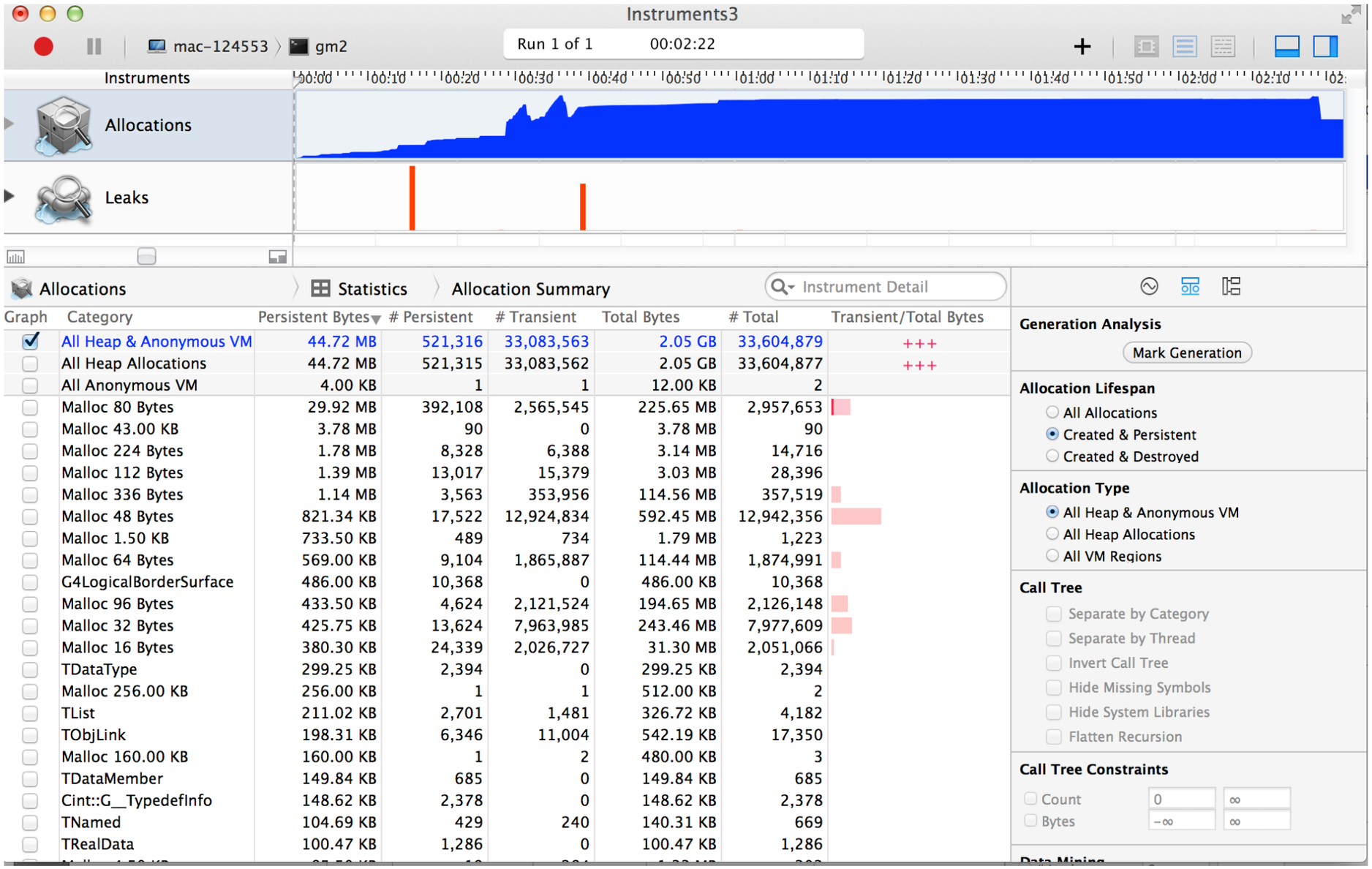
You can start by choosing the *Allocations* and *Leak* profilers from the beginning template screen. Or, if you are already profiling (e.g. with the *Time Profiler*), delete the current instrument (*Instrument > Delete*).

Then press the + sign near the top of the window. Choose the *Allocations* and *Leak* profilers.

The *Allocations profiler* works by instrumenting all calls to allocate and free memory. Doing so adds a significant amount of time to the execution of the program. Following our example, let's change the arguments to do 200 events instead of 2000. You can do so by choosing and then editing the target.

In the *Allocations profiler*, you want to watch for a consistent rise in memory over time. Such a rise could indicate mistakes in memory management.

The *Leak profiler* indicates memory that is no longer accessible. This happens if memory is allocated and referred to by a pointer, but the pointer goes out of scope or is deleted. That allocated memory can no longer be accessed and is a memory leak.



This display shows a list of all memory allocations. For some, *Instruments* can figure out the object type. It can't for others and they are shown as Malloc.

Memory peaks at about 85 MB and then drops at the end to about 45 MB.

Though it is not easy, one can trace when memory is created and destroyed by the program.

Two periods of leaks are seen. Select the *Leak profiler* to learn more about the leaks.