

INTRODUCTION To OBJECTIVE-C

CSCI 4448/5448: OBJECT-ORIENTED ANALYSIS & DESIGN

LECTURE 12 — 02/17/2011

Goals of the Lecture

- Present an introduction to Objective-C 2.0
- Coverage of the language will be INCOMPLETE
 - We'll see the basics... there is a lot more to learn

History (I)

- Brad Cox created Objective-C in the early 1980s
 - It was his attempt to add object-oriented programming concepts to the C programming language
 - NeXT Computer licensed the language in 1988; it was used to develop the NeXTSTEP operating system, programming libraries and applications for NeXT
 - In 1993, NeXT worked with Sun to create OpenStep, an open specification of NeXTSTEP on Sun hardware

History (II)

- In 1997, Apple purchased NeXT and transformed NeXTSTEP into MacOS X which was first released in the summer of 2000
- Objective-C has been one of the primary ways to develop applications for MacOS for the past 11 years
- In 2008, it became the primary way to develop applications for iOS targeting (currently) the iPhone and the iPad and (soon, I'm guessing) the Apple TV

Objective-C is “C plus Objects” (I)

- Objective-C makes a small set of extensions to C which turn it into an object-oriented language
- It is used with two object-oriented frameworks
 - The Foundation framework contains classes for basic concepts such as strings, arrays and other data structures and provides classes to interact with the underlying operating system
 - The AppKit contains classes for developing applications and for creating windows, buttons and other widgets

Objective-C is “C plus Objects” (II)

- Together, Foundation and AppKit are called Cocoa
- On iOS, AppKit is replaced by UIKit
 - Foundation and UIKit are called Cocoa Touch
- In this lecture, we focus on the Objective-C language,
 - we'll see a few examples of the Foundation framework
 - we'll see examples of UIKit in Lecture 13

C Skills? Highly relevant

- Since Objective-C is “C plus objects” any skills you have in the C language directly apply
 - statements, data types, structs, functions, etc.
- What the OO additions do, is reduce your need on
 - structs, malloc, dealloc and the like
 - and enable all of the object-oriented concepts we’ve been discussing
- Objective-C and C code otherwise freely intermix

Development Tools (I)

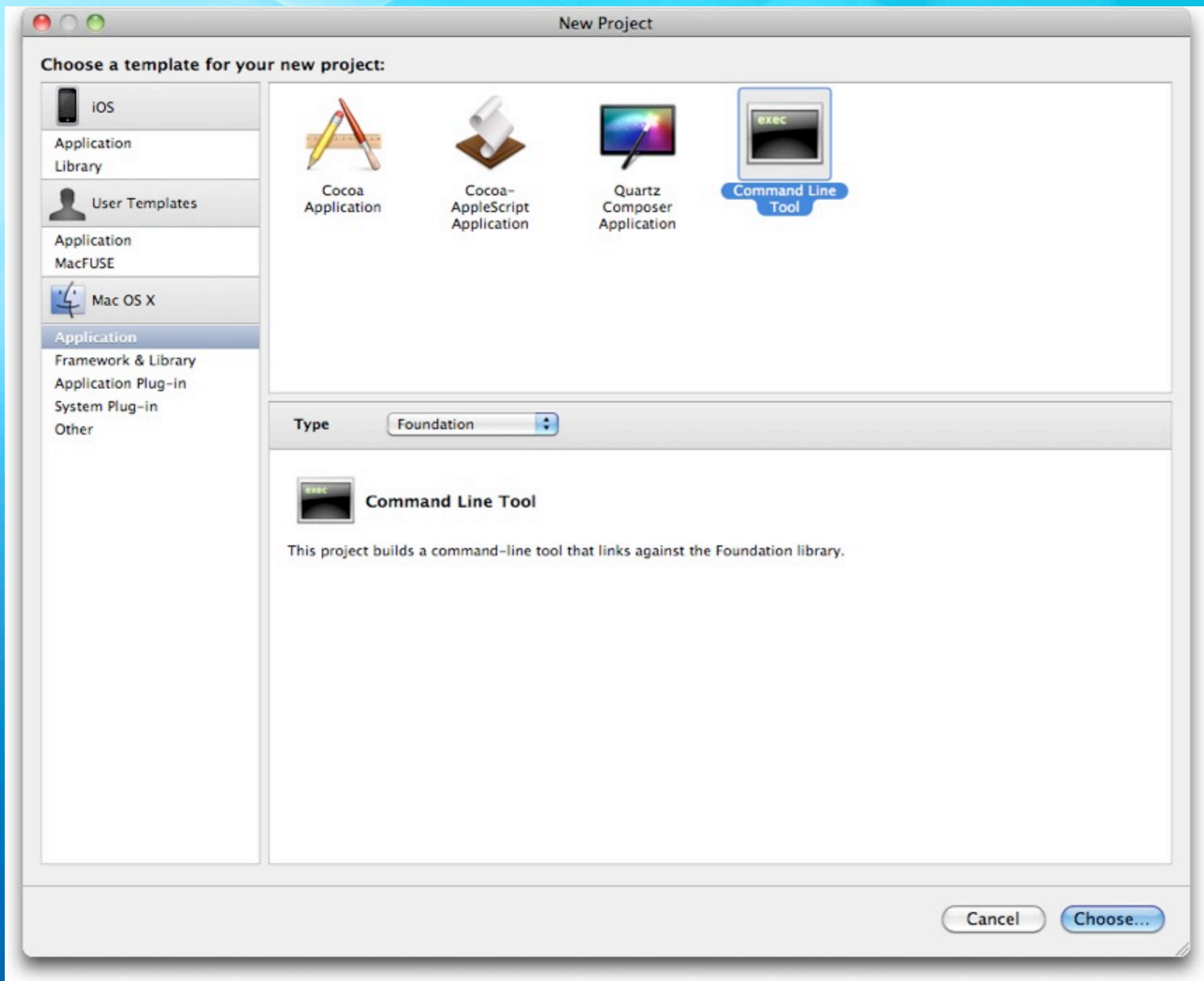
- Apple provides XCode and Interface Builder to develop programs in Objective-C
 - Behind the scenes, XCode makes use of gcc to compile Objective-C programs
- In a future version of XCode, to be released shortly, Interface Builder will go away as a separate application
 - Its functionality will be merged into XCode
 - We'll see examples of Interface Builder next week

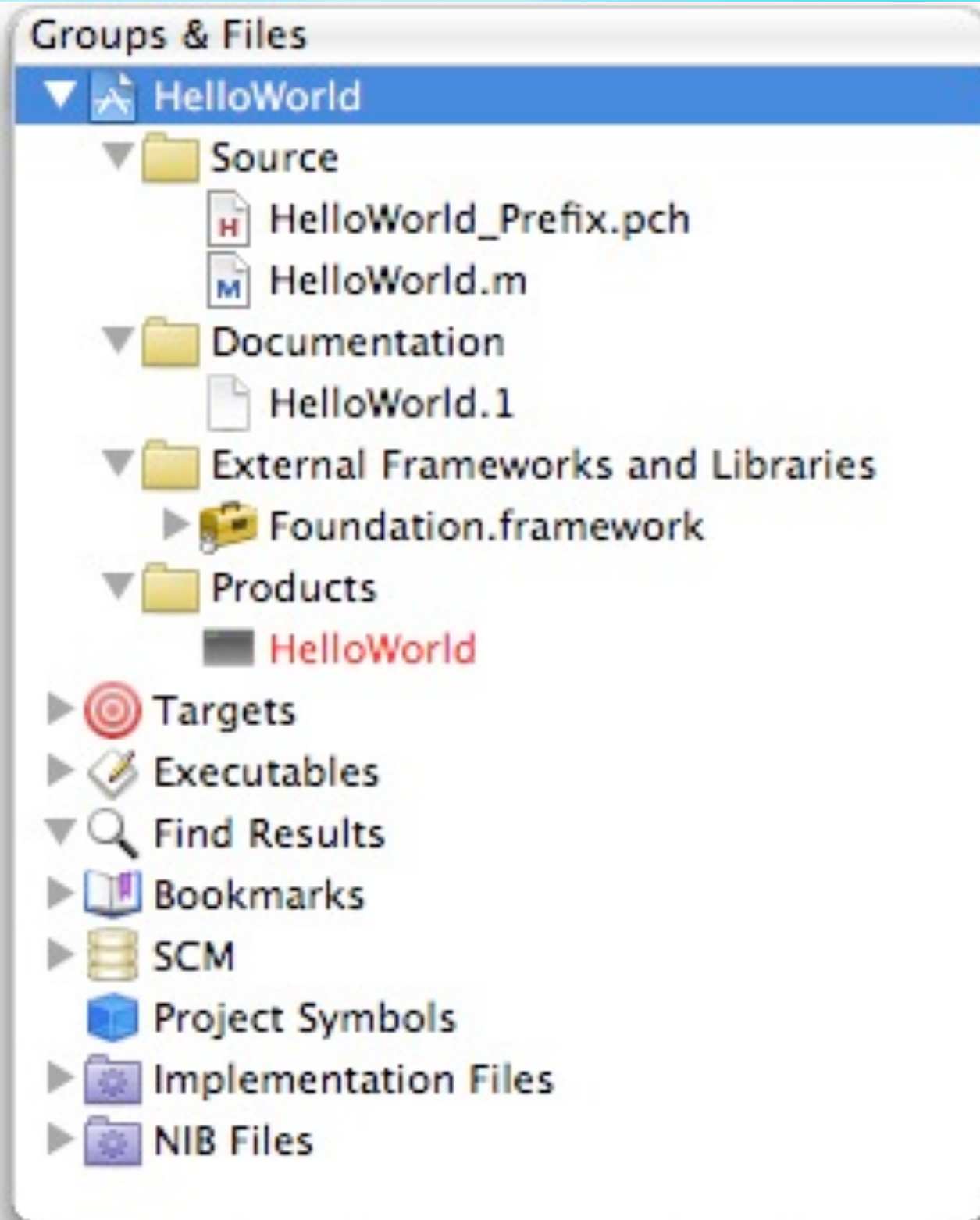
Development Tools (II)

- XCode and Interface Builder comes with Snow Leopard on the Development Tools DVD
- It is also available for download at <[http://
developer.apple.com/](http://developer.apple.com/)>
 - You need to register in Apple's development program but registration is free
- To install: double click the downloaded .dmg file and then double click the installer and follow instructions

Hello World

- As is traditional, let's look at our first objective-c program via the traditional Hello World example
- To create it, we launch XCode and create a New Project
 - In the resulting dialog (see next slide)
 - select Application under the MacOS X
 - select Command Line Tool on the right
 - select Foundation from the pop-up menu
 - click Choose and select a location for the project

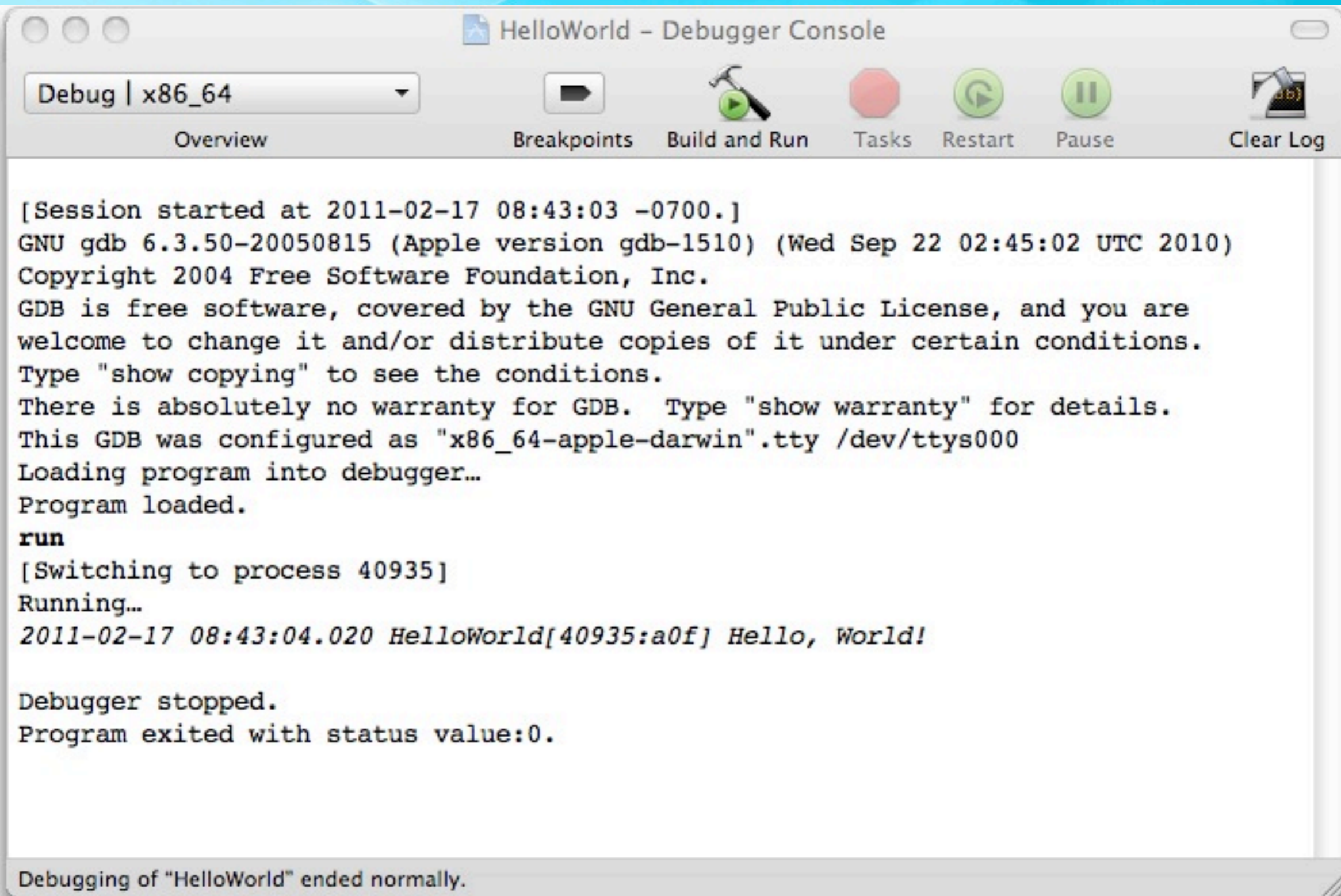




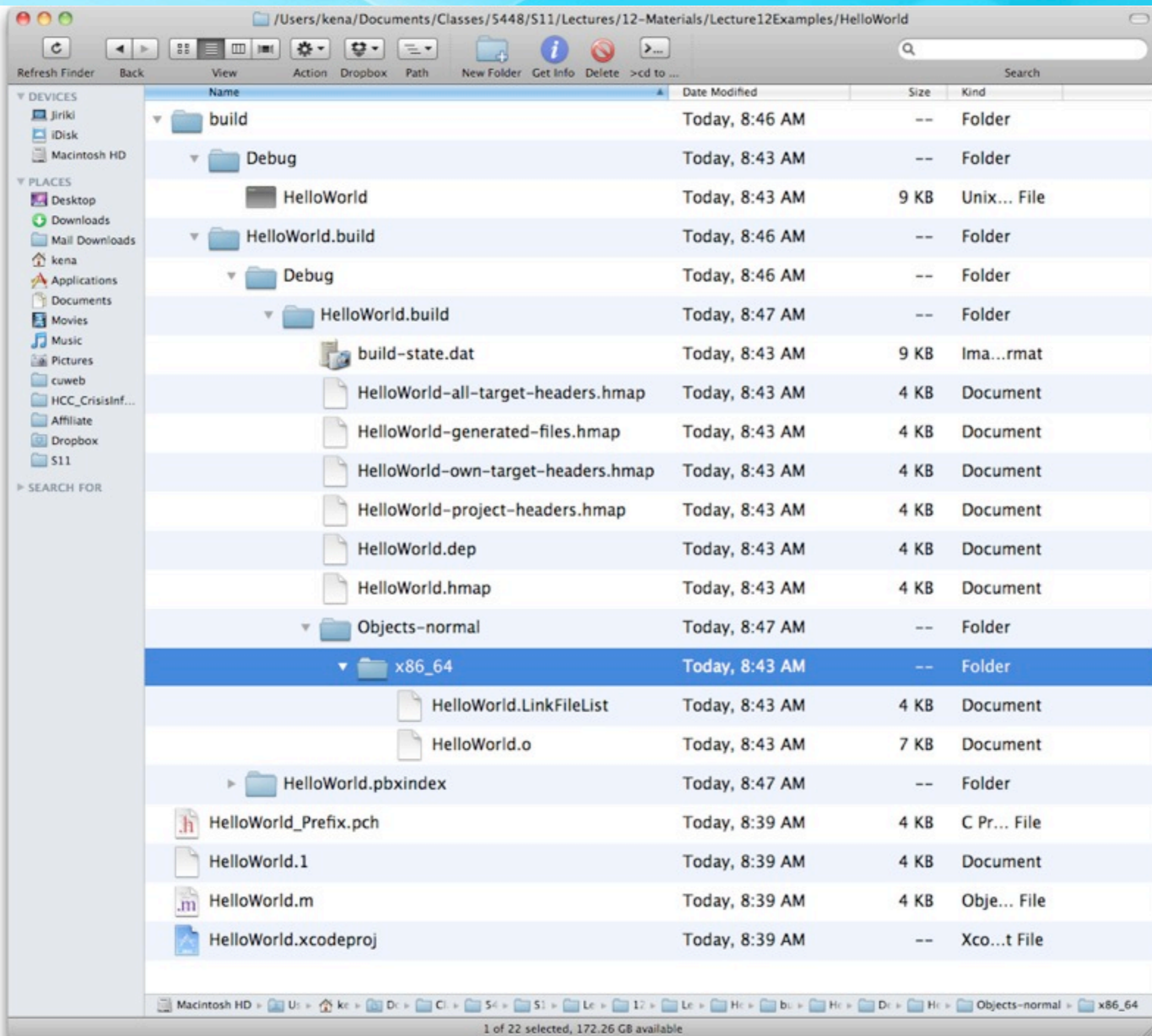
Similar to what we saw with Eclipse, XCode creates a default project for us;

There are folders for this command line program's source code (.m files), documentation, external dependencies and products (the application itself)

Note: the Foundation framework is front and center and HelloWorld is shown in red because it hasn't been created yet

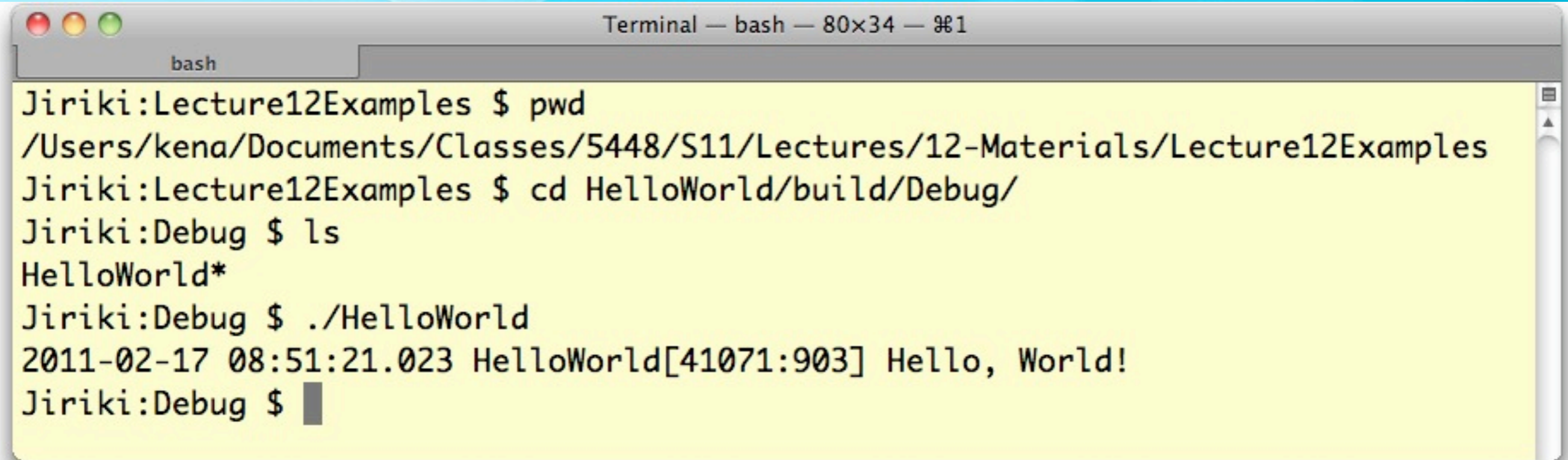


Exciting, isn't it. The template is ready to run; clicking "Build and Run" brings up a console that shows "Hello, World!" being displayed; One interesting thing to note is that the program is being run by gdb



The resulting project structure on disk does not map completely to what is shown in Xcode; The source file, man page, project file, and pre-compiled header file are all in the same directory;

A lot of stuff gets generated when you build

A screenshot of a macOS Terminal window. The title bar at the top reads "Terminal — bash — 80x34 — %1". The terminal has a tab labeled "bash". The command history is as follows:
Jiriki:Lecture12Examples \$ pwd
/Users/kena/Documents/Classes/5448/S11/Lectures/12-Materials/Lecture12Examples
Jiriki:Lecture12Examples \$ cd HelloWorld/build/Debug/
Jiriki:Debug \$ ls
HelloWorld*
Jiriki:Debug \$./HelloWorld
2011-02-17 08:51:21.023 HelloWorld[41071:903] Hello, World!
Jiriki:Debug \$ █

The resulting executable can be executed from the command line, fulfilling the promise that we were creating a command-line tool

As you can see, most of the text on Slide 11 was generated by gdb... our command line tool doesn't do much but say hi to the world.

Note the “2011-02-17 08:51:21.023 HelloWorld[41071:903]” is generated by a function called NSLog() as we'll see next

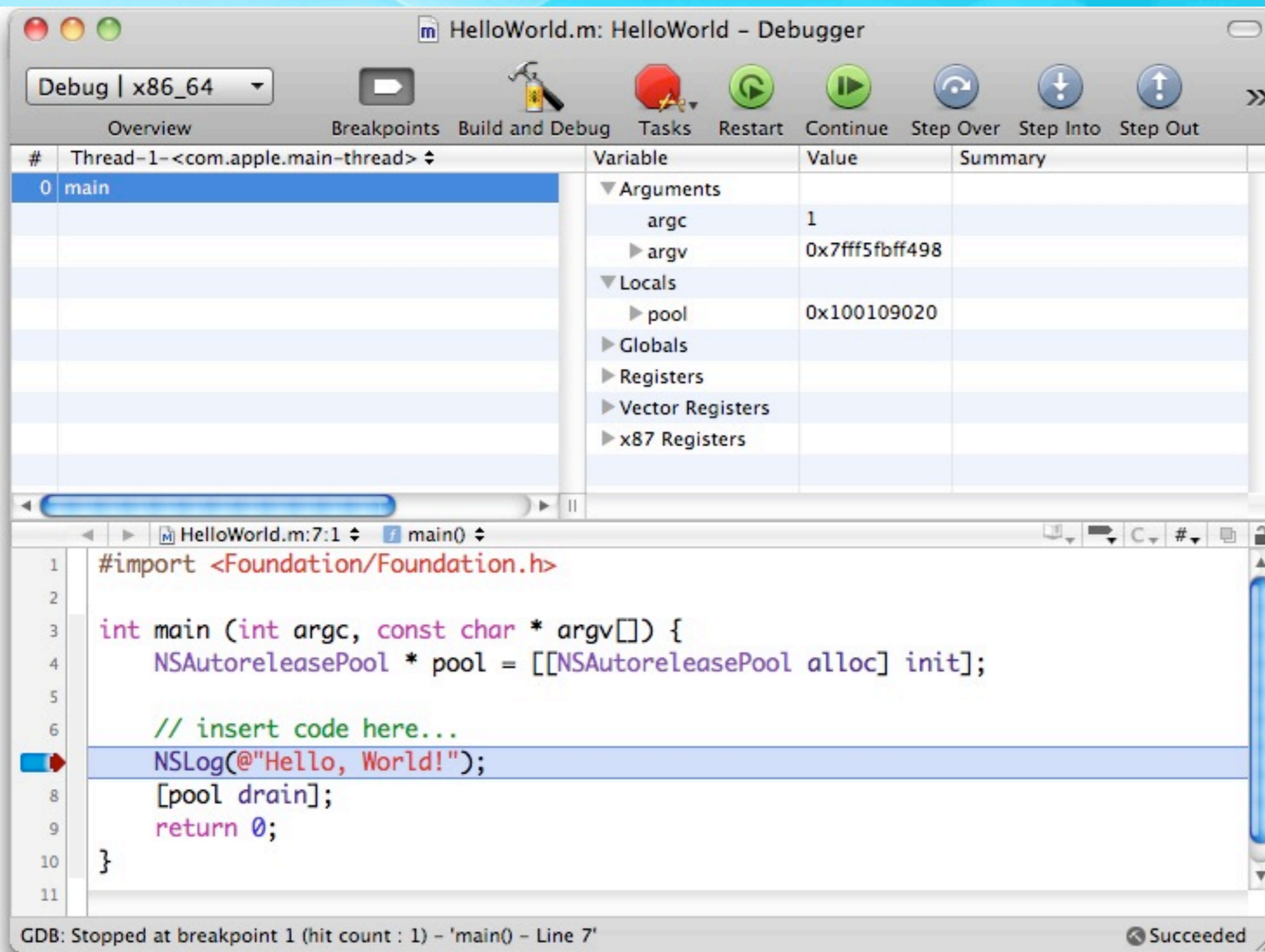
```
1  #import <Foundation/Foundation.h>
2
3  int main (int argc, const char * argv[]) {
4      NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
5
6      // insert code here...
7      NSLog(@"Hello, World!");
8      [pool drain];
9      return 0;
10 }
11
```

Objective-C programs start with a function called main, just like C programs; #import is similar to C's #include except it ensures that header files are only included once and only once

Ignore the “NSAutoreleasePool” stuff for now

Thus our program calls a function, NSLog, and returns 0

The blue arrow indicates that a breakpoint has been set; gdb will stop execution on line 7 the next time we run the program



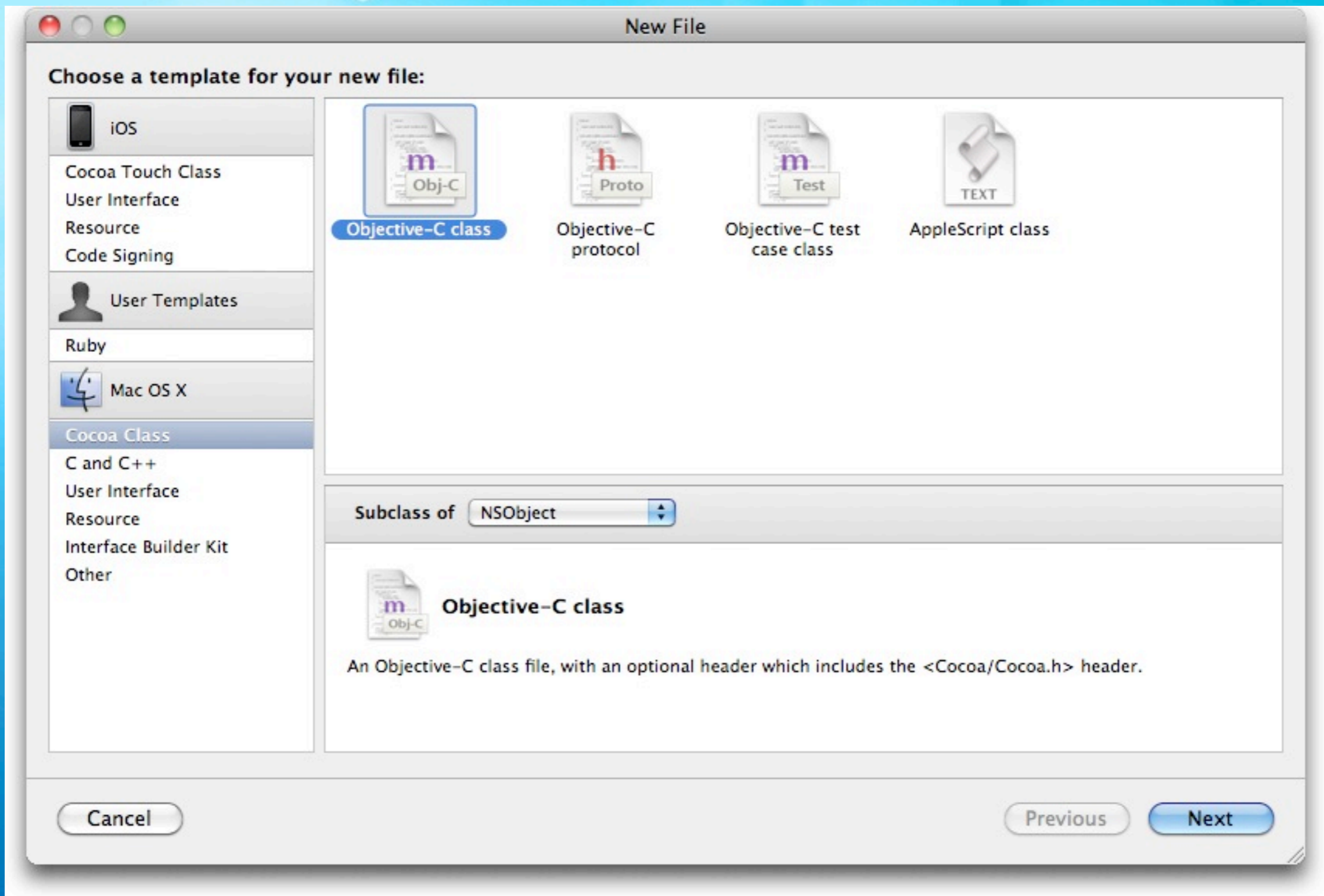
gdb is integrated into XCode; here we see the program stopped at our breakpoint; variables can be viewed in the upper right

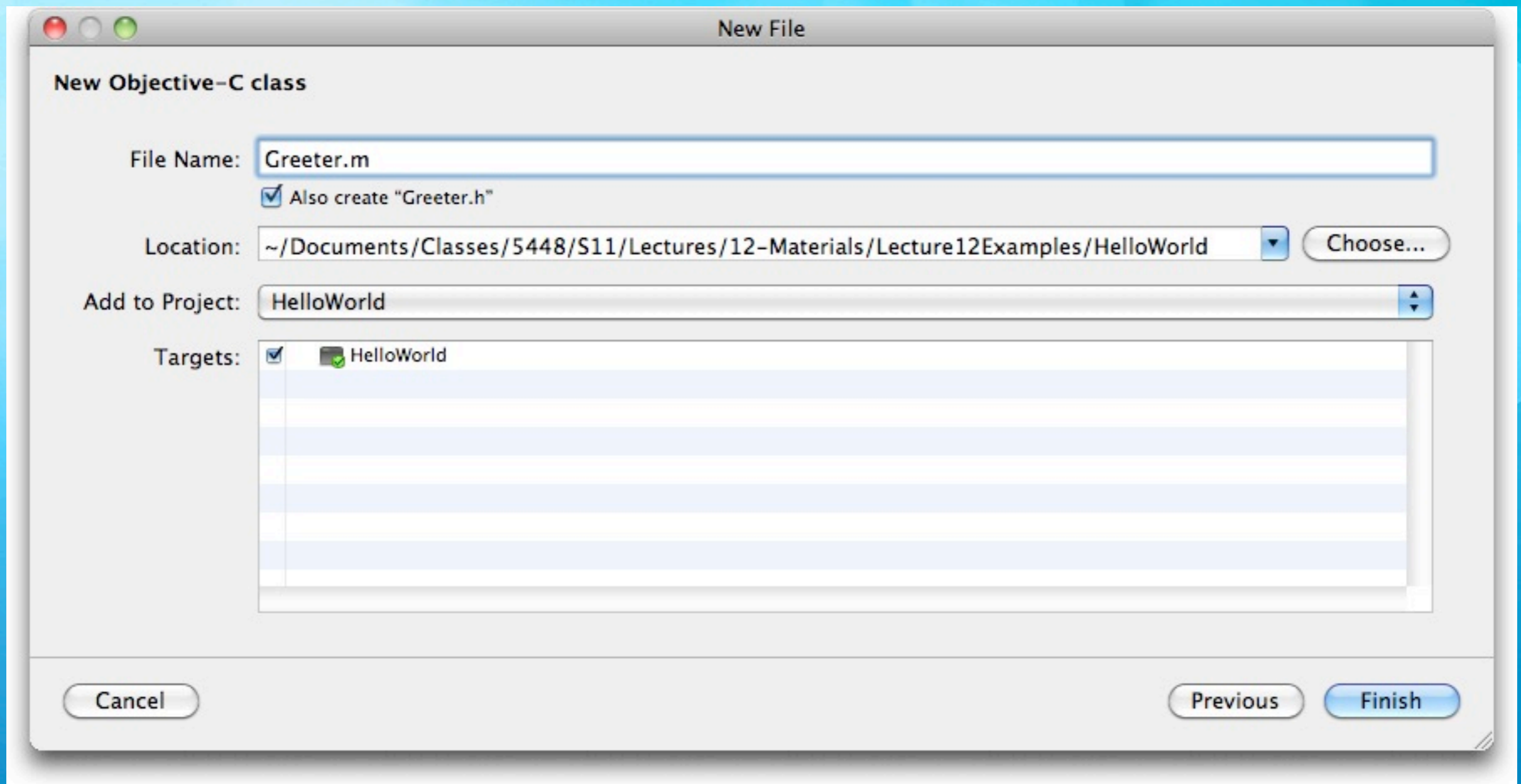
Let's add objects...

- Note: This example comes from “Learning Objective-C 2.0: A Hands-On Guide to Objective-C for Mac and iOS Developers” written by Robert Clair
 - It is an excellent book that I highly recommend
 - His review of the C language is an excellent bonus to the content on Objective-C itself
- We're going to create an Objective-C class called Greeter to make this HelloWorld program a bit more object-oriented

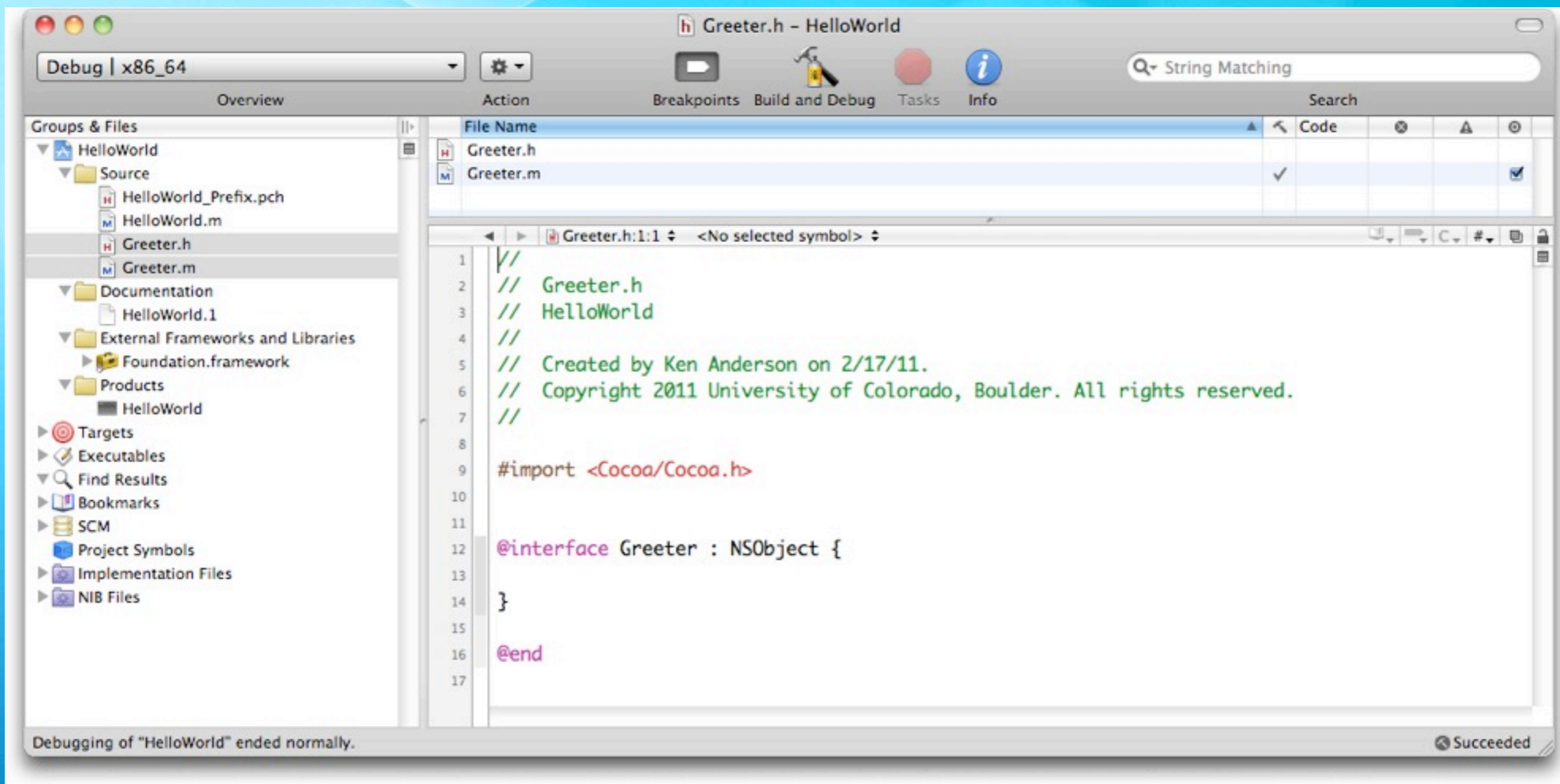
First, we are going to add a class

- Select File ⇒ New File
- In the resulting Dialog (see next two slides)
 - Select Cocoa Class
 - Select Objective-C class
 - Select NSObject from the “Subclass of” menu
 - Click the Next button and title the class Greeter.m and ask that Greeter.h be generated. Click Finish





This dialog names the new class, creates a header file and adds the class to the current project (HelloWorld) and to the current target (HelloWorld); A single project can generate multiple executables, which are known as targets; for this program, we just have one target



After the creation, Greeter.h and Greeter.m have been added to our project and the header file has been displayed automatically; Note the import of Cocoa.h; all this does is import Foundation.h and AppKit.h. We could switch this to just import Foundation.h as we won't be using AppKit

Objective-C classes

- Classes in Objective-C are defined in two files
 - A header file which defines the attributes and method signatures of the class
 - An implementation file (.m) that provides the method bodies

Header Files

- The header file of a class has the following structure

```
<import statements>
```

```
@interface <classname> : <superclass name> {
```

```
    <attribute definitions>
```

```
}
```

```
<method signature definitions>
```

```
@end
```

Objective-C additions to C (I)

- Besides the very useful `#import`, the best way to spot an addition to C by Objective-C is the presence of this symbol



Objective-C additions to C (II)

- In header files, the two key additions from Objective-C are
 - `@interface`
- and
 - `@end`
- `@interface` is used to define a new objective-c class
 - As we saw, you provide the class name and its superclass; Objective-C is a single inheritance language
- `@end` does what it says, ending the `@interface` compiler directive

Greeter's interface (I)

```
#import <Foundation/Foundation.h>
|
@interface Greeter : NSObject {
    NSString *greetingText;
}

- (NSString *) greetingText;
- (void) setGreetingText: (NSString *) newText;
- (void) greet;

@end
```

**We've added
one attribute**

greetingText of
type
NSString *

**“pointer to an
NSString”**

**NS stands for
“NeXTSTEP” !!**

Greeter's interface (II)

```
#import <Foundation/Foundation.h>
|
@interface Greeter : NSObject {
    NSString *greetingText;
}

- (NSString *) greetingText;
- (void) setGreetingText: (NSString *) newText;
- (void) greet;

@end
```

**We've added
three method
signatures**

**one getter, one
setter, and one
method to issue
a greeting**

Objective-C Methods (I)

- It takes a while to get use to Objective-C method signatures
 - `(void) setGreetingText: (NSString *) newText;`
- defines an instance method (-) called `setGreetingText:`
- The colon signifies that the method has one parameter and is PART OF THE METHOD NAME
 - `newText` of type `(NSString *)`
- The names `setGreetingText:` and `setGreetingText` refer to TWO different methods; the former has a parameter

Objective-C Methods (II)

- A method with multiple parameters will have multiple colon characters and the parameter defs are interspersed with the method name
 - `- (void) setStrokeColor: (NSColor *) strokeColor`
 - `andFillColor: (NSColor *) fillColor;`
- The above signature defines a method with two parameters called `setStrokeColor:andFillColor:`

NSString * and NSColor *

- We've now seen examples of types
 - NSString * and NSColor *
- What does this mean?
 - The * in C means “pointer”
 - Thus, this can be read as
 - “pointer to <class>”
 - it simply means an instance has been allocated and we have a handle to the instance

Let's implement the method bodies

- The implementation file of a class looks like this

```
<import statements>
```

```
@implementation <classname>
```

```
<method body definitions>
```

```
@end
```

Greeter's implementation

```
#import "Greeter.h"

@implementation Greeter

- (NSString *) greetingText {
    return greetingText;
}

- (void) setGreetingText: (NSString *) newText {
    [newText retain];
    [greetingText release];
    greetingText = newText;
}

- (void) greet {
    NSLog(@"%@", [self greetingText]);
}

- (void) dealloc {
    [greetingText release];
    [super dealloc];
}

@end
```

We implement the getter, setter, greet method and a Framework method called dealloc that takes care of memory allocation issues

The getter is straightforward;

Let's look at the others in detail

But first, calling methods (I)

- The method invocation syntax of Objective-C is
 - `[object method:arg1 method:arg2 ...] ;`
- Method calls are enclosed by square brackets
 - Inside the brackets, you list the object being called
 - Then the method with any arguments for the methods parameters

But first, calling methods (II)

- Here's a call using Greeter's setter method; @"Howdy!" is a shorthand syntax for creating an NSString instance
 - `[greeter setGreetingText: @"Howdy!"];`
- Here's a call to the same method where we get the greeting from some other Greeter object
 - `[greeterOne setGreetingText:[greeterTwo greetingText]];`
- Above we nested one call inside another; now a call with multiple args
 - `[rectangle setStrokeColor: [NSColor red] andFillColor: [NSColor green]];`

Memory Management (I)

- Memory management of Objective-C objects involves the use of six methods
 - alloc, init, dealloc, retain, release, autorelease
- Objects are created using alloc and init
- We then keep track of who is using an object with retain and release
- We get rid of an object with dealloc (although, we never call dealloc ourselves)

Memory Management (II)

- When an object is created, its retain count is set to 1
 - It is assumed that the creator is referencing the object that was just created
- If another object wants to reference it, it calls retain to increase the reference count by 1
 - When it is done, it calls release to decrease the reference count by 1
- If an object's reference count goes to zero, the runtime system automatically calls dealloc

Memory Management (III)

- I won't talk about autorelease today, we'll see it in action soon
- Objective-C 2.0 added a garbage collector to the language
 - When garbage collection is turned on, retain, release, and autorelease become no-ops, doing nothing
 - However, the garbage collector is not available when running on iOS, so the use of retain and release are still with us; as the hardware of iOS devices gets more powerful, garbage collection will be available everywhere and these memory management techniques will go away

Back to the Code

```
- (void) setGreetingText: (NSString *) newText {  
    [newText retain];  
    [greetingText release];  
    greetingText = newText;  
}
```

- You are now in a position to understand the setter method
 - We retain the new NSString passed in, we release the NSString we previously pointed at, we set our greetingText to point at the new NSString

The greet method

```
- (void) greet {  
    NSLog(@"%@", [self greetingText]);  
}
```

- NSLog is a variable argument function.
 - The number of arguments is determined by the format string that is passed in as the first argument
 - Just like printf in C
 - This format string “%@",” says convert an object into a string
- Note: [self greetingText] is this object invoking the getter function on itself rather than accessing the greetingText string directly

The dealloc method

```
- (void) dealloc {  
    [greetingText release];  
    [super dealloc];  
}
```

- The dealloc method releases the NSString that we are pointing at with our attribute and then invokes the dealloc method of our superclass
 - We've now seen examples of the **self** and **super** keywords

A new main method

- We now need a new version of main to make use of our new Greeter class
 - We'll import its header file
 - We'll instantiate an instance of the class
 - We'll set its greeting text
 - We'll call its greet method
 - We'll release it

```
#import <Foundation/Foundation.h>
#import "Greeter.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Greeter *myGreeter = [[Greeter alloc] init];

    [myGreeter setGreetingText:@"Hello from Objective-C!!"];

    [myGreeter greet];

    [myGreeter release];

    [pool drain];
    return 0;
}
```

The only thing new is the sequence of calls on **alloc** and **init**; alloc is a class method of NSObject; we can invoke it on Greeter since it is inherited; it returns a new instance and we then call init on it. We didn't override init so a default version defined by NSObject will execute instead. Otherwise, we create it, set the greeting, invoke greet, and **release; ignore pool for now** © Kenneth M. Anderson, 2011

Some things not (yet) discussed

- Objective-C has a few additions to C not yet discussed
 - The type `id: id` is defined as a pointer to an object
 - `id iCanPointAtAString = @"Hello";`
 - Note: no need for an asterisk in this case
 - The keyword `nil: nil` is a pointer to no object
 - It is similar to Java's null
 - The type `BOOL: BOOL` is a boolean type with values `YES` and `NO`; used throughout the Cocoa frameworks

Wrapping Up (I)

- Basic introduction to Objective-C
 - main methods
 - class and method definition and implementation
 - method calling syntax
 - creation of objects and memory management
- More to come as we use this knowledge to explore the iOS platform in future lectures

Coming Up Next

- Homework 4 Assigned on Friday
- Lecture 13: Introduction to iOS
- Homework 4 Due on Wednesday
- Lecture 14: Review for Midterm
- Lecture 15: Midterm
- Lecture 16: Review of Midterm