10 Basic Debugging

A fly, Sir, may sting a stately horse and make him wince; but one is but an insect, and the other is a horse still.
Samuel Johnson

Errors, like straws, upon the surface flow;
He who would search for pearls must dive below.
John Dryden

Starting a debugging session

1 **Click the launch button to bring up the Launch panel.**

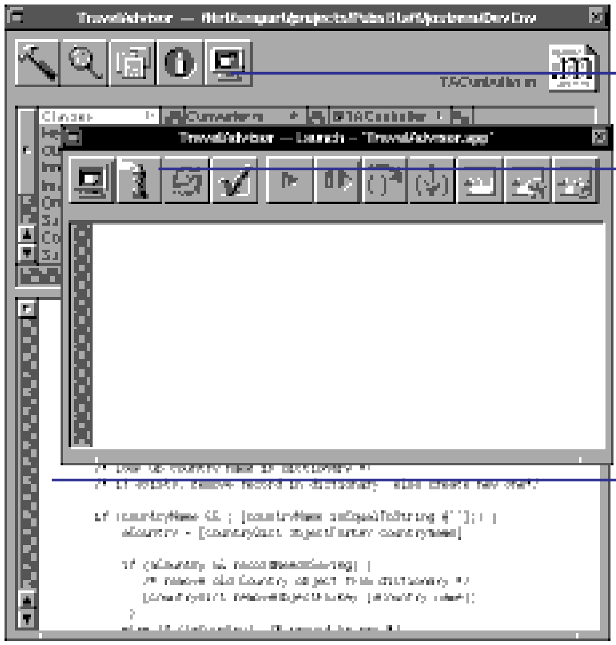
2 **If necessary, click the checkmark button in the Launch panel to set the executable name and environment.**

3 **Click the debug button to start the debugger.**

Or

3 **Click the launch button to run your program without debugging.**

You can use Project Builder's Launch panel to run your program in the debugger or, if you want, outside of the debugger. The Launch panel provides an interface to the Free Software Foundation's **gdb** debugger.


Click this button or choose Launch from the Tools menu.
Click here to start the debugger.
When the debugger is running, this column displays the program counter and marks each breakpoint.

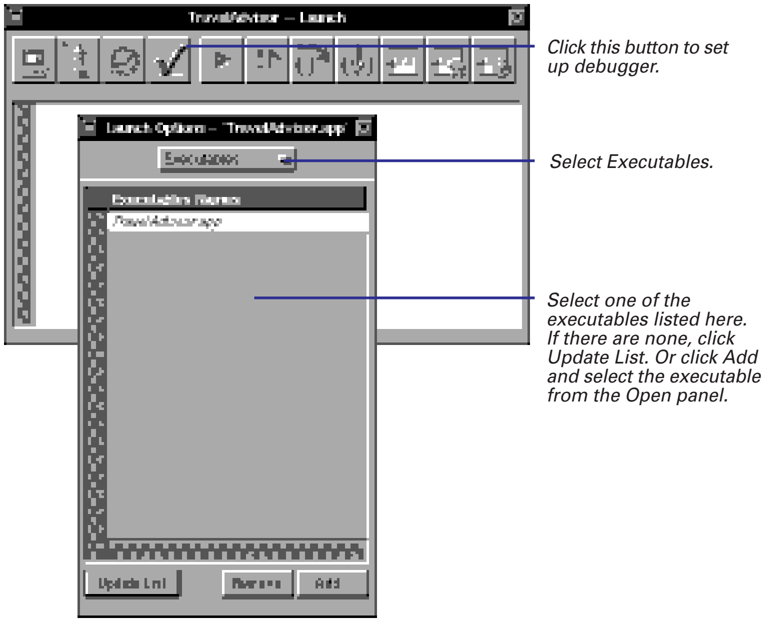
gdb is a command line tool, and it has a rich command-line interface. Project Builder provides a user interface for the most common debugging tasks. You perform other debugging tasks using the **gdb** command line interface.

After **gdb** starts up, you can click the run button to start up your program.

Project Builder automatically includes debugging symbols in your program if you use the default build target. See Chapter 9, "Building" for more information on building a program.

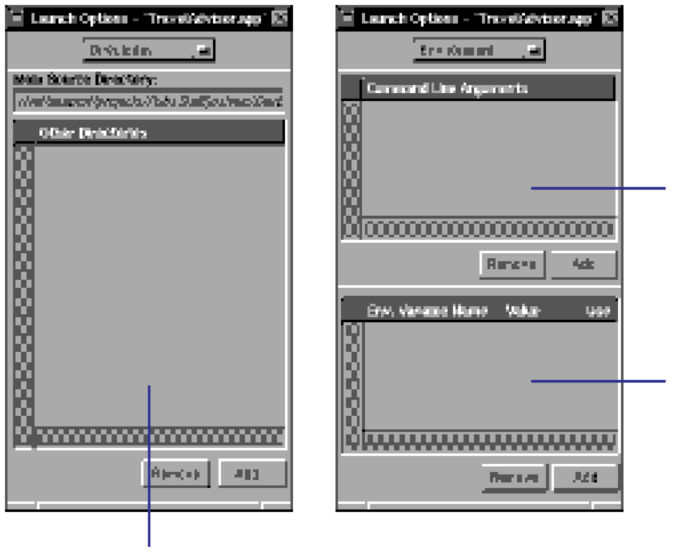
For a complete list of **gdb** commands, see the *OPENSTEP Development Tools Reference*.

If the Launch panel isn't displaying the name of the project executable in its titlebar, you need to set it in the Launch Options panel before you can run or debug the program. Needing to set the executable usually occurs when your project contains subprojects or builds a binary that doesn't run on its own (for example, it's a framework or a loadable bundle).


TravelAdvisor — Launch
Launch Options – 'TravelAdvisor.app'
Executables
Executables Names
TravelAdvisor.app
Update List
Remove
Add
Click this button to set up debugger.
Select Executables.
Select one of the executables listed here. If there are none, click Update List. Or click Add and select the executable from the Open panel.

Setting Up the Program's Environment

Besides allowing you to choose the executable, the Launch Options panel also lets you set environment variables, pass arguments to your program, and specify other directories that contain source code for this project.



Enter any arguments you want passed to the executable here, one per line. Click Add to add a new line.

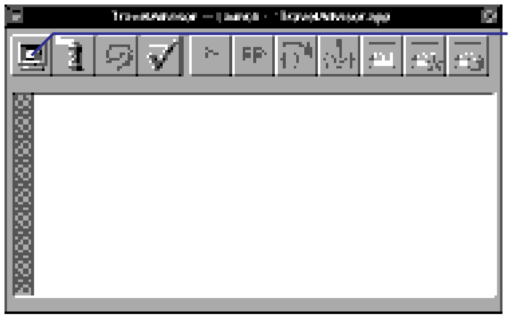
If there are environment variables you want to use, click Add to enter them here.

To have ***gdb*** *look in other directories for source code, click Add to enter them here.*

You can change environment variables and command line arguments while your program is running, but they won't take effect until you restart your program.

Running the Program Outside of the Debugger

If you want to launch the program outside of the debugger, click the Launch button instead of the Debug button. The program is started up and operates independantly, just as if you'd started it outside of Project Builder.


Click here to run the program.

Attaching to an Already Running Process

Sometimes, a problem occurs only when you launch an application outside the debugger. When you launch it inside **gdb**, the problem disappears. If this happens , launch the application using the launch button and use the **gdb** command **attach** to hook up to it:

```
(gdb) attach pid
```

pid is the process ID of the process you want to debug.

attach immediately stops the application. When you use **attach**, you can debug the process just like you normally would: by setting breakpoints, modifying storage, and so on.

When you're finished debugging, use **detach** (which takes no arguments) to detach the debugger from the process. The process resumes executing. You'll kill the process if you try to quit **gdb** or if you try to start the program from the beginning. (**gdb** asks for confirmation before it allows you to do these things.)

If you're having trouble attaching to a process before the errant code is executed, send your program a stop signal as one of the first messages:

```
[NSThread sleepUntilDate:
    [NSDate distantFuture]];
```

This indefinitely suspends execution of the application. Once you attach in **gdb**, click the continue button to go on from there.

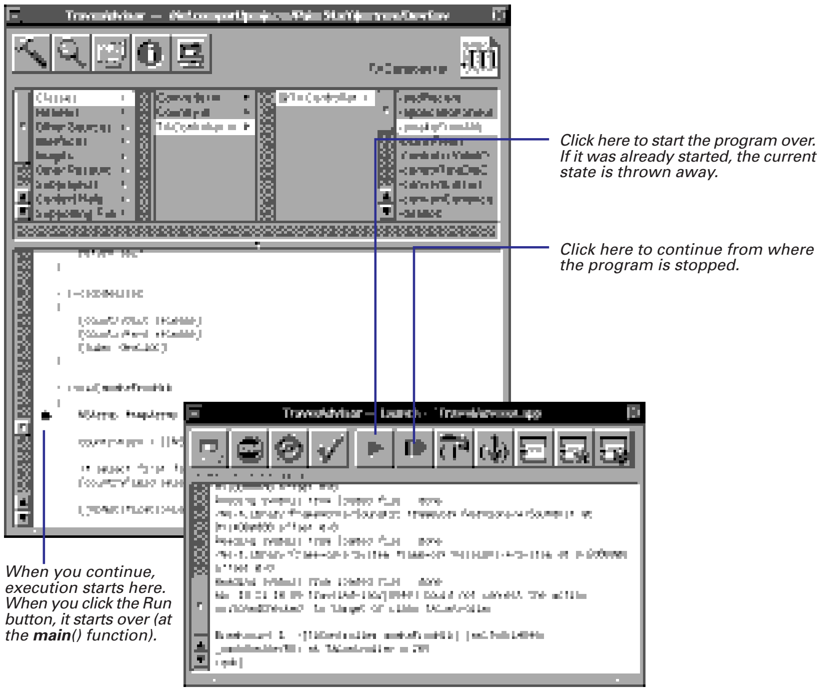
Running the program in the debugger

▶ **Click the continue button to continue from where you left off.**

Or

▶ **Click the run button to start the program over.**

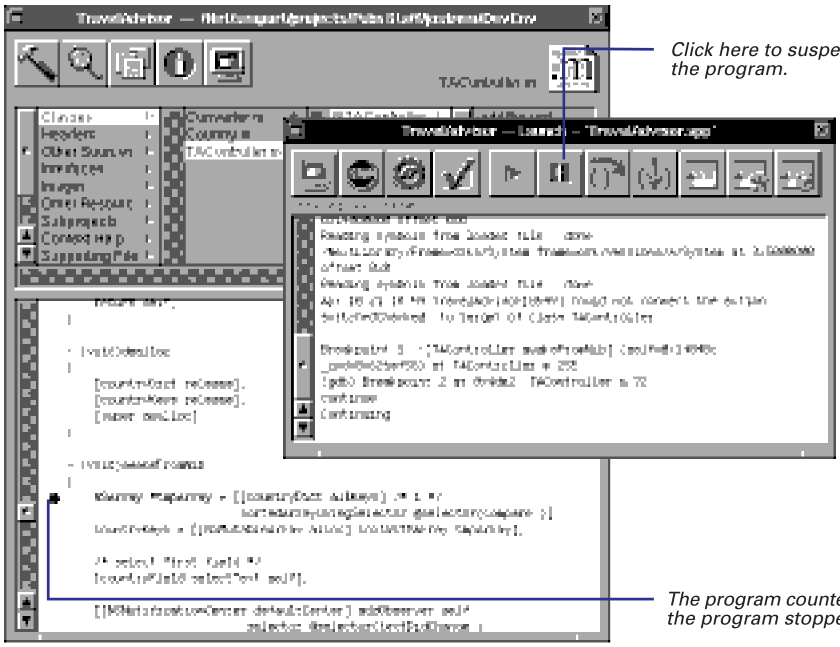
The run button starts you program from the beginning. Typically, you only use the run button to start the program for the first time. If you've interrupted the program or hit a breakpoint and you want to continue executing from that point, use the continue button.


Click here to start the program over. If it was already started, the current state is thrown away.
Click here to continue from where the program is stopped.
When you continue, execution starts here. When you click the Run button, it starts over (at the **main**() function).

Interrupting the program

▶ **Click the suspend button.**

It’s often necessary to suspend the program that you’re debugging so that you can examine its state or enter **gdb** commands. For example, to set a breakpoint or to see the value of a certain variable, you must interrupt the program first.


TravelAdvisor — [illegible]
TravelAdvisor — Launch — TravelAdvisor.app
TAController.m
Classes
Headers
Other Sources
Interfaces
Images
Other Resources
Subprojects
Context Help
Supporting Files
Click here to suspend the program.
The program counter shows where the program stopped executing.

```
[illegible]
Reading symbols from loaded file ... done
/System/Library/Frameworks/System.framework/Versions/A/System at 0x[illegible]
offset 0x0
Reading symbols from loaded file ... done
[illegible]
[illegible]

Breakpoint 1, -[TAController awakeFromNib] [illegible]
[illegible] at TAController.m:[illegible]
(gdb) Breakpoint 2 at [illegible]: TAController.m 72
continue
Continuing
```

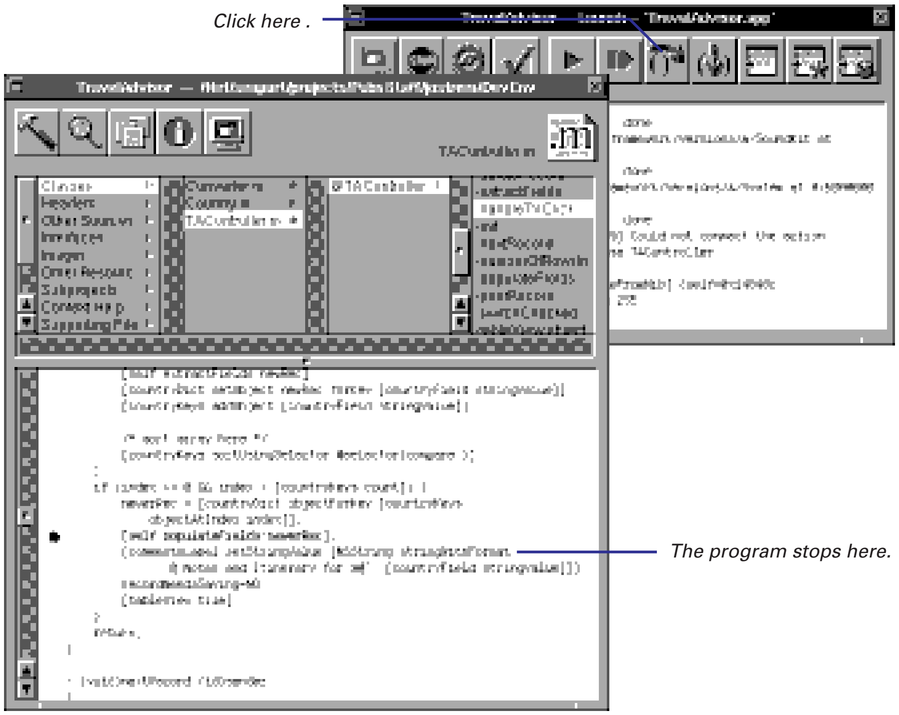
```
        [illegible]
    }

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

    - (void)awakeFromNib
    {
        NSArray *tmpArray = [[countryDict allKeys] /* 1 */
                              sortedArrayUsingSelector:@selector(compare:)];
        countryKeys = [[NSMutableArray alloc] initWithArray:tmpArray];

        /* select first field */
        [countryField selectText:self];

        [[NSNotificationCenter defaultCenter] addObserver:self
                              selector:@selector([illegible]) ...
```

If you’re debugging an application, it will display the spinning cursor when you suspend it. The application won’t accept any input until you continue execution.

Executing a single line of code

▶ **Click the next button.**

If you know that a particular method has an error in it somewhere, you can execute that method a line at a time (called *stepping*) to see exactly where the error occurs.

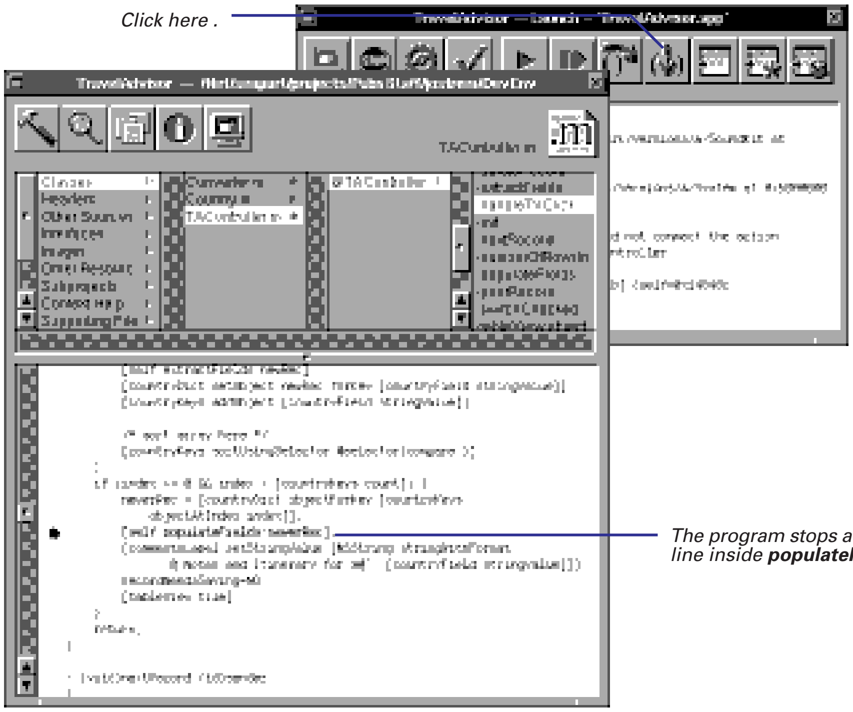


The next button always executes until control returns to the line of code directly below the current line. That is, if the current line contains a function call or invokes an Objective-C method, the entire function or method is executed, and the program doesn't stop until it returns (unless a breakpoint is hit).

Stepping into a method or function

▶ **Click the step button.**

If the program counter is pointing to a line containing a method invocation and you click the next button, that method executes before the program stops. If you want the program to stop inside of that method, click the step button instead.



```
        [self extractFields:newRec];
        [countryDict setObject:newRec forKey:[countryField stringValue]];
        [countryKeys addObject:[countryField stringValue]];

        /* sort array here */
        [countryKeys sortUsingSelector:@selector(compare:)];
    }
    if (index >= 0 && index < [countryKeys count]) {
        newRec = [countryDict objectForKey:[countryKeys
            objectAtIndex:index]];
        [self populateFields:newRec];
        [commentsLabel setStringValue:[NSString stringWithFormat:
              @"Notes and Itinerary for %@", [countryField stringValue]]];
        recordNeedsSaving=NO;
        [tableView tile];
    }
    return;
}

- (void)clearRecord:(id)sender
{
```

The step button executes the program until control reaches a different line in the program. If the current line contains a method invocation, the program stops at the beginning of that method. If the method isn't part of your program (if it's in one of the OpenStep frameworks, for example), the entire method is executed. Execution doesn't stop until the next line of code you own.

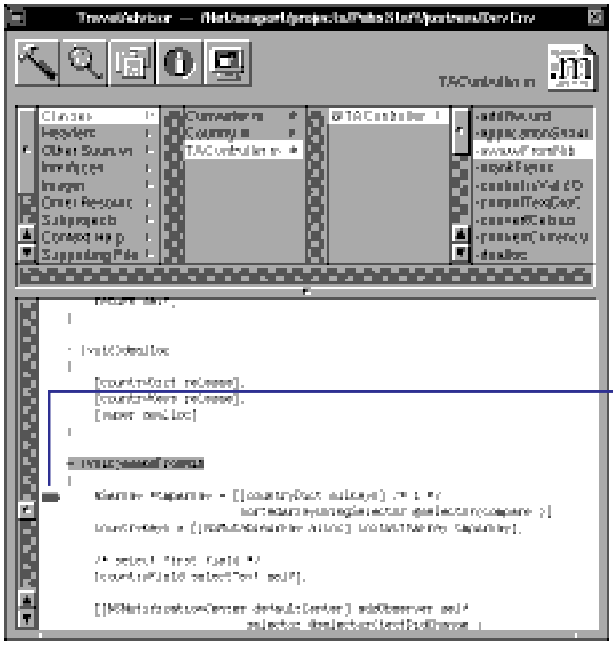
Setting breakpoints

1 **If the program is running, click the suspend button to stop it.**

2 **In the Project Builder main window, double-click in the gray area next to the line where you want the breakpoint.**

3 **Click the continue button to execute up to the breakpoint.**

A breakpoint makes your program stop whenever a certain point in the program is reached. Every time the program encounters the line of code that has the breakpoint, it stops executing.



```
    return self;
}

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

- (void)awakeFromNib
{
    NSArray *tmpArray = [[countryDict allKeys] /* 1 */
                            sortedArrayUsingSelector:@selector(compare:)];
    countryKeys = [[NSMutableArray alloc] initWithArray:tmpArray];

    /* select first field */
    [countryField selectText:self];

    [[NSNotificationCenter defaultCenter] addObserver:self
                            selector:@selector([illegible])
```

Double-click next to the line where you want the breakpoint.

If you're debugging an application, the cursor spins when it hits a breakpoint. When you enter this state, go to the Launch panel and examine the program's state (print values of variables, examine the stack, and so on).

Setting breakpoints on data

Sometimes you want to stop the program whenever the value of a variable changes, no matter which part of your code is doing the changing. To do this, use a watchpoint . To set a watchpoint:

```
(gdb) watch expr
```

where *expr* is any expression or variable.

gdb treats watchpoints and breakpoints the same. Anything you can do to a breakpoint, you can also do to a watchpoint (see "Cool Breakpoint Stuff" in this chapter). The Breakpoints display of the Task Inspector provides information on both breakpoints and watchpoints.

When a watchpoint is set, your program runs much more slowly that if you had set a normal breakpoint, so use watchpoints sparingly. (One alternative is to set a conditional breakpoint, described in "Cool Breakpoint Stuff.") However, watchpoints are sometimes the only way to catch an error when you don't know where the error occurs.

Cool Breakpoint Stuff

Using **gdb** commands, you can add more power to your breakpoints and make debugging a breeze. For complete information on **gdb** breakpoints, see the *OPENSTEP Development Tools Reference* manual. Here are some highlights.

Setting Breakpoints in Dynamically Loaded Code

gdb doesn't know about symbols in dynamically loaded code (such as code inside frameworks or loadable bundles) because it's not laoded until run time. This means you can't set a breakpoint in a framework until after you start the program that uses it. This is pretty frustrating when the framework is what you want to debug. However, you can set a future breakpoint when the framework isn't loaded yet. To do this, use the **future-break** command:

```
(gdb) future-break address
```

(*address* can be a method name, a function name, a file name and line number, and so on.) When you enter this command, **gdb** checks the loaded symbols for a symbol matching *address*. If one is found, it resolves the breakpoint. If not, it holds on to it. Then, whenever a dynamic shared library is loaded, **gdb** checks the breakpoint against the newly loaded symbols until it can resolve the symbol in the breakpoint. (If the symbol can never be resolved, the **future-break** just sits around doing nothing.)

When you quit the program, **gdb** unloads all of the breakpoints set in dynamic shared libraries. These breakpoints are converted into future breakpoints—when the library is loaded again, the breakpoints are resolved again.

Future breakpoints are just like normal breakpoints in every other respect; you can add commands to them, disable them, enable them, and so on. In the Breakpoints display of the Task Inspector, they are listed as "unloaded."

Conditional Breakpoints

If you only want a breakpoint to stop when a certain condition is true, use the **condition** command:

```
(gdb) condition bnum expression
```

expression is any Boolean expression and it's associated with breakpoint number *bnum*. (The Breakpoints view of the Task Inspector tells you the breakpoint number.) From now on, this breakpoint will stop the program only if the value of *expression* is true. To remove a condition from a breakpoint, enter **condition** with no *expression*.

Ignoring Breakpoints

You can disable a breakpoint for a specific length of time with **gdb** command **ignore**:

```
(gdb) ignore bnum count
```

This command ignores the breakpoint the next *count* times it is reached. (0 means the program stops the next time it's reached.) If the breakpoint is a conditional breakpoint, the condition isn't checked unless the ignore count is 0.

Executing Commands at a Breakpoint

You can give any breakpoint a series of commands to execute when the program stops at it. For example, if you want to know what the value of the variable x is whenever breakpoint 5 is hit, enter the following. (You must type **end** when you're through to make the **gdb** prompt return.)

```
(gdb) commands 5
> print x
> end
```

This brings up a handy trick for ignoring breakpoints. Often, you don't know how many times you want to ignore a breakpoint (making the **ignore** command useless), but you know that you want to ignore it until a specific point in a program is reached. For example, say you want to stop at a method named **setCurrent:** but only if the message is sent by the **processParagraph** method. In this case, you can do the following:

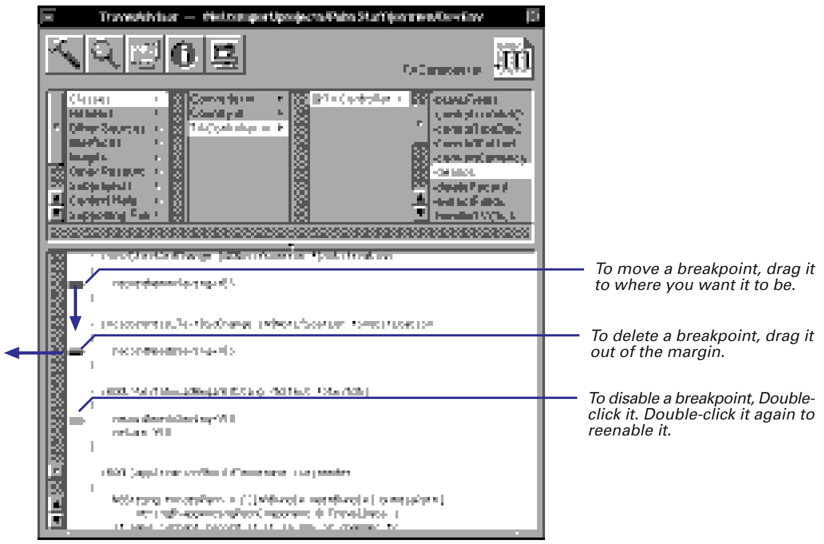
```
(gdb) break setCurrent:
Breakpoint #1 set
(gdb) break processParagraph
Breakpoint #2 set
(gdb) disable 1
(gdb) commands 2
> silent
> enable 1
> continue
> end
(gdb) continue
```

This example sets two breakpoints, one at the beginning of each method. Then, it disables the breakpoint at **setCurrent:**. When the breakpoint at **processParagraph** is reached, it enables the breakpoint at **setCurrent:** and continues executing. (**silent** is just a convenience. It means that **gdb** won't print the usual stopped at breakpoint message.)

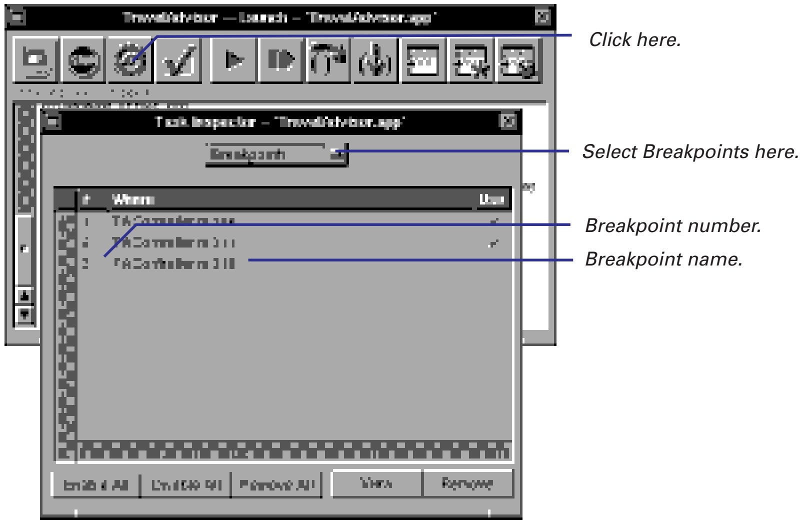
Managing breakpoints

- **To move a breakpoint, drag it to where you want it to be.**
- **To delete a breakpoint, drag it off the Project Builder window.**
- **To disable a breakpoint, double-click it.**
- **To find out information about breakpoints, bring up the Breakpoints display of the Task Inspector.**

When you no longer need to stop at a breakpoint anymore, you can delete the breakpoint; however, you might want to just disable it. When you delete a breakpoint, it is gone forever. When you disable a breakpoint, it still exists and is still displayed in the Project Builder main window, but the program does not stop at the breakpoint. You can enable the disabled breakpoint later.


To move a breakpoint, drag it to where you want it to be.
To delete a breakpoint, drag it out of the margin.
To disable a breakpoint, Double-click it. Double-click it again to reenable it.

Sometimes it's useful to know how many breakpoints you've set and where they are. The Breakpoints view of the Task Inspector provides this information. The first column of this display gives you the breakpoint number, which is used in many **gdb** commands.



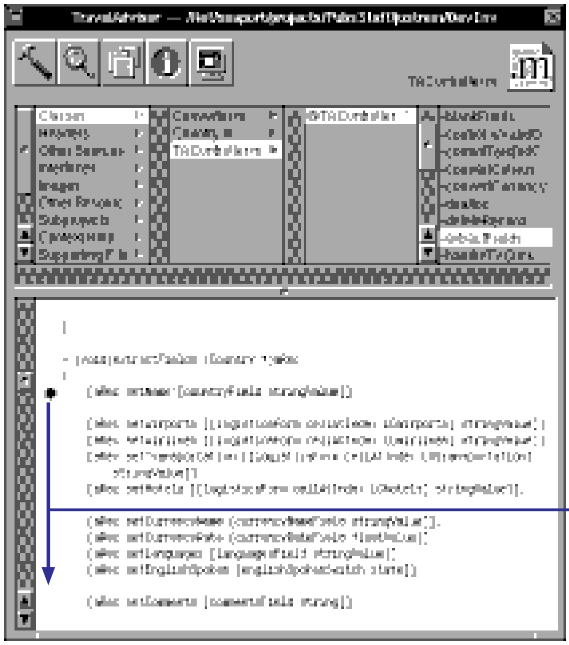
Using the Breakpoints display, you can also enabled and disable each breakpoint by clicking the Use column, or you can enable, disable, and remove all breakpoints. Use the View button to have Project Builder go to the line where the breakpoint is set.

See "Cool Breakpoint Stuff" in this chapter for some useful **gdb** commands involving breakpoints.

Executing several lines of code

▶ **Drag the program counter to the line where you want execution to stop.**

When you're stepping through code, you often hit a place where you'd like to execute several lines of code and stop again. For example, if you encounter a **for** loop that is executed several dozen times, you probably want to jump through the **for** loop and resume stepping after the loop ends. To do this, just drag the program counter to the first line of code past the **for** loop. The entire loop executes, and the program stops when it reaches the line of code you've dragged the counter to. You can then click the next or step button to resume stepping.



```
{
- (void)extractFields:(Country *)aRec
{
    [aRec setName:[countryField stringValue]];

    [aRec setAirports:[[logisticsForm cellAtIndex:LGAirports] stringValue]];
    [aRec setAirlines:[[logisticsForm cellAtIndex:LGAirlines] stringValue]];
    [aRec setTransportation:[[logisticsForm cellAtIndex:LGTransportation]
        stringValue]];
    [aRec setHotels:[[logisticsForm cellAtIndex:LGHotels] stringValue]];

    [aRec setCurrencyName:[currencyNameField stringValue]];
    [aRec setCurrencyRate:[currencyRateField floatValue]];
    [aRec setLanguages:[languageField stringValue]];
    [aRec setEnglishSpoken:[englishSpokenSwitch state]];

    [aRec setComments:[commentsField string]];
```

Drag the program counter past the lines of code yo want to skip over.

Dragging the program counter is more convenient than setting a breakpoint at the end of the loop. You don't have to delete the breakpoint when you're done.

Navigating using the stack

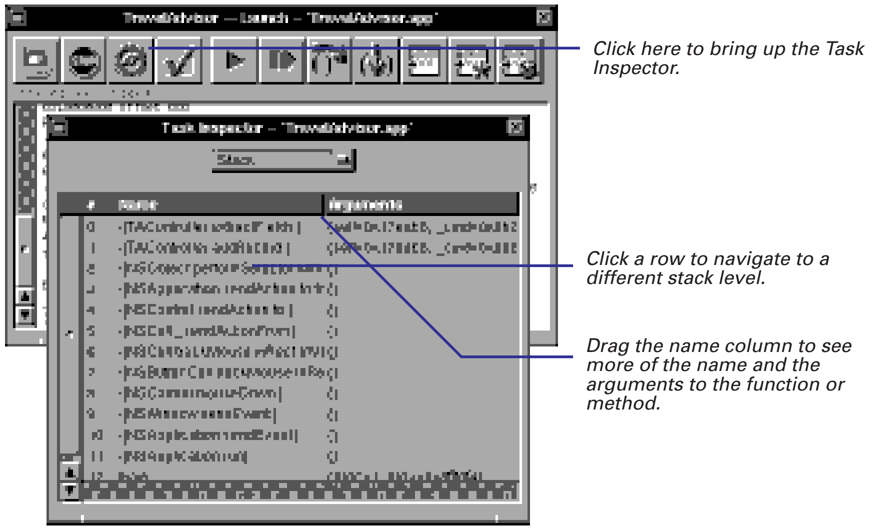
1 **Click the inspector button on the Launch panel to bring up the Task Inspector.**

2 **Choose Stack from the pop-up list.**

3 **Click a stack frame to have the debugger jump to that stack frame.**

Each time your program invokes a C function, a C++ member function, or an Objective-C method, the information about where in the program the call was made is saved in a block of data called a stack frame. The frame also contains the arguments of the call and the local variables of the function that was called.

The Task Inspector displays the stack on the right side of the window. Each row in the stack display represents one stack frame. The current stack frame is numbered 0, the frame that called it is 1, and so on.


TravelAdvisor — Launch – 'TravelAdvisor.app'
Task Inspector – 'TravelAdvisor.app'
Stack
#
Name
Arguments
Click here to bring up the Task Inspector.
Click a row to navigate to a different stack level.
Drag the name column to see more of the name and the arguments to the function or method.

At any given time, one of the stack frames is selected by **gdb**; many **gdb** commands refer implicitly to this selected frame. In particular, whenever you ask **gdb** for the value of a variable in the program, the value is found in the selected frame. You can select any frame by clicking it. You can then examine the values of variables pertaining to that stack frame. As you navigate to a different stack frame, the Project Builder main window shows you the currently executing line of code at that frame.

Tip: You can return from the current level by Shift-clicking the program counter.

Examining the value of a variable or an object

1. **If the program is running, click the Suspend button to suspend it.**
2. **Select the variable in the Project Builder main window.**
3. **In the Launch panel, click the object button if the variable is an object.**

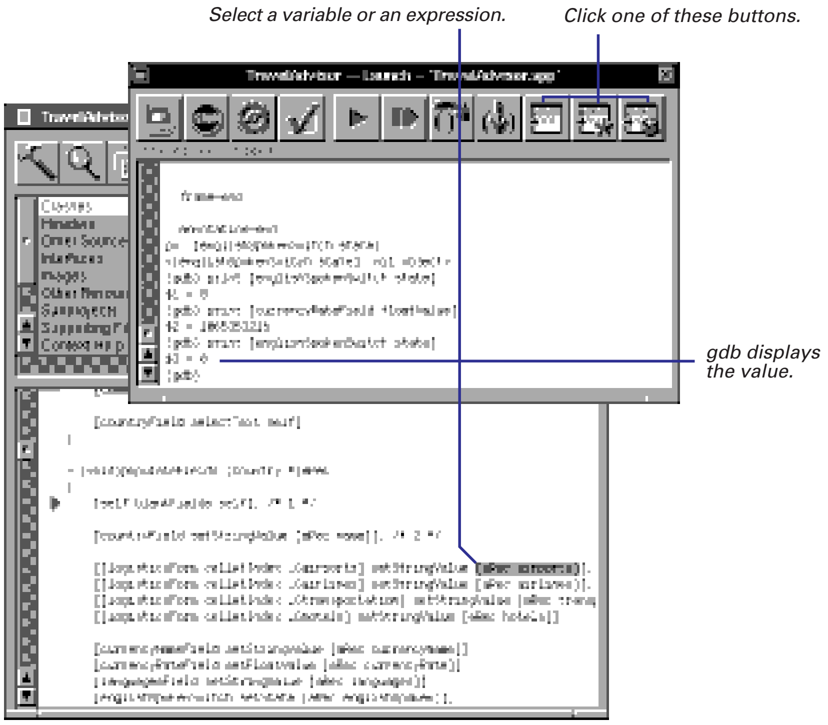
   Or

3. **Click the * button if the variable is a pointer.**

   Or

3. **Click the Print button.**

The three rightmost Debugger buttons print the values of variables or expressions.


Select a variable or an expression.
Click one of these buttons.
gdb displays the value.

The first of the three buttons (the Print button) prints the value of a variable or expression if it's not a pointer or an object. If the variable (or the result of the expression) is a pointer, the Print button prints the address. Usually, you want to know the value at that address, not the address itself. In that case, use the next button over (the one with a dereference symbol), which prints the value pointed to by the selected variable. Similarly, use the button with a cube (the Print-object button) instead of Print to see the information about Objective-C objects.

Getting Useful Information From Print-object

The Print-object button (which invokes the **gdb** command **print-object**) sends the message **description** to the selected object. NSObject defines the **description** method, so all objects respond to it. By default, this method prints the object's class name and hexidecimal address:

```
<NSApplication: 0xbb5e4>
```

However, you can override this method in your classes to provide more useful data. Compared to dumping the contents of the underlying struct, an implementation of **description** can print out just the information that is helpful and use a more readable format. Your **description** method should return an NSString.

Many Foundation classes override **description.** For example, NSArrays, NSDictionaries, and NSStrings print their contents instead of their addresses.

For the Experts: More on Examining Variables

Making Sure Variables Stick Around

When you build the program using the default build target (for example, app for Application projects), an optimized, debuggable executable results. This executable is helpful if a bug surfaces only in the optimized version; however, debugging optimized code sometimes gives surprising results. Control flow may change and variables may disappear without a trace. You ask **gdb** to print such a variable and even though the source clearly shows it is in scope, **gdb** replies:

```
(gdb) print num
No symbol "num" in current context
```

To ensure that a variable be available in the debugger even after optiumization, declare the variable **volatile**.

Value History

gdb maintains a value history for your session. This means that every expression you evaluate using the **print** command (or the Print, Print *, and PO buttons) is assigned a value number in the history, like this:

```
(gdb) print self
$7 = (struct NSApplication *) 0xbb5e4
```

You can refer to this value as $7 and use it in future expressions:

```
(gdb) print (char *) [$7 appName]
$8 = 0xb80cc "FunWithGDB"
```

Once a value is entered into the history, it doesn’t change. The value is stored as $7, not the expression that generated it. This means that $7 doesn’t change to hold the new value of **self** when your program enters a different scope.

Also, at any time, $ refers to the last value in the history and $$ to the next-to-last value.

The **output** command has the same semantics as the **print** command, but doesn’t add the result to the value history. You can use this difference to avoid cluttering the value history with unimportant results. For more sophisticated printing needs, **gdb** provides a **printf** command similar to the C version that provides for formatted output. Like **output**, the results from **printf** are not entered into the value history.

Any name that begins with a $ can be used as the name of a **gdb** convenience variable. These variables are implicitly typed and created at first reference. Use **print** to get the value of a convenience variable and the **set** command to set or change the value. You can set the value to any valid C or Objective-C expression, including methods or functions:

```
(gdb) p $array = [NSArray array]
$24 = 793052
(gdb) p $num = 1230 % 4
$25 = 2
```

All registers have convenience variables associated with them. The **info registers** command dumps the contents of all registers so you can see the names associated with each register. The register convenience variables most often used are **$fp**, which holds the frame pointer, **$sp** for the stack pointer, and **$pc** for the program counter.

Locating Your Variables

To find out how a variable is stored, use this command:

```
(gdb) info address self
Symbol "self" is a variable in register a2.
```

info address tells you if the variable is stored on the stack or in a register. This command is useful to determine if optimizations are causing problems, particularly on RISC machines.

Examining Raw Memory

Use the command **x** (for “examine”) to examine memory without referencing the program's data types.

x is followed by a slash and an output format specification, followed by an expression for an address:

```
x/fmt addr
```

These *fmt* letters specify the size of unit to examine:

b	Examine individual bytes.
h	Examine halfwords (two bytes each).
w	Examine words (four bytes each).
g	Examine giant words (eight bytes).

These *fmt* letters specify how to print the contents:

x	Print as integers in unsigned hexadecimal.
d	Print as integers in signed decimal.
u	Print as integers in unsigned decimal.
o	Print as integers in unsigned octal.
a	Print as an address, both absolute and relative
c	Print as character constants (this implies size b).
f	Print as floating-point. This works only with sizes w and g.
s	Print a null-terminated string of characters.
i	Print a machine instruction in assembler syntax (or nearly).

Once you’ve entered **x** to see the value at an address, hit return to see the value at the next address.

Debugging object allocation and deallocation

- **Use enableFreedObjectCheck: inside gdb.**

  Or
- **Use the oh tool to see where and when objects are allocated and deallocated.**

  Or
- **Use the AnalyzeAllocation tool to see where and when objects are allocated and deallocated.**

Object allocation and deallocation are often trouble spots. Two common problems are using an object after it has been deallocated and releasing an object too many times. Here are some strategies and tools to debug object allocation and deallocation.

A typical autorelease error:

```
objc: FREED(id): message objectForKey: sent to freed object=0xfde44
```

Ignoring Autorelease Errors

You may want to debug the rest of your program first, saving the release problems until later. The **enableRelease:** convenience method defined in Foundation's NSAutoreleasePool class helps you ignore autorelease errors. NSAutoreleasePool defines the application's autorelease pool. When an object is autoreleased, it is added to the autorelease pool. At the top of the event loop, all objects in the pool are sent a **release** message, which decrements the reference count and potentially deallocates the object. NSAutoreleasePool allows you to control that pool.

If you receive messages from the debugger indicating that you are sending messages to deallocated objects, enter this command:

```
(gdb) call [NSAutoreleasePool
enableRelease:NO]
```

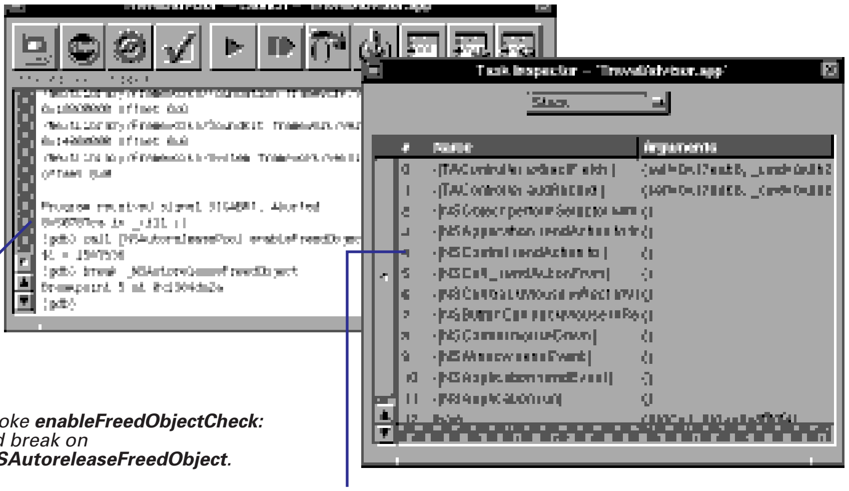
This message disables the deallocation of all objects in your program, ignoring autorelease errors.

Your program must be started when you send this message. It's often useful to break on **main()** and send this message after the first line or two of the program.

Debugging Autorelease Errors in gdb

If you are releasing an object too many times, invoke the NSAutoreleasePool class method **enableFreedObjectCheck:** and set a breakpoint on **_NSAutoreleaseFreedObject**.

enableFreedObjectCheck: causes all **autorelease** and **release** messages to first check to see if the receiving object is already in an autorelease pool. If it is, they won't deallocate the object. When the program hits the breakpoint, look at the stack to see what method was releasing the object.



```
[illegible]
0x[illegible] offset 0x0
/NextLibrary/Frameworks/SoundKit.framework/[illegible]
0x[illegible] offset 0x0
/NextLibrary/Frameworks/System.framework/[illegible]
offset 0x0

Program received signal SIGABRT, Aborted
0x[illegible] in _[illegible] ()
(gdb) call [NSAutoreleasePool enableFreedObjec
$1 = 1547536
(gdb) break _NSAutoreleaseFreedObject
Breakpoint 5 at 0x[illegible]
(gdb)
```

Task Inspector – 'TravelAdvisor.app'

Stack

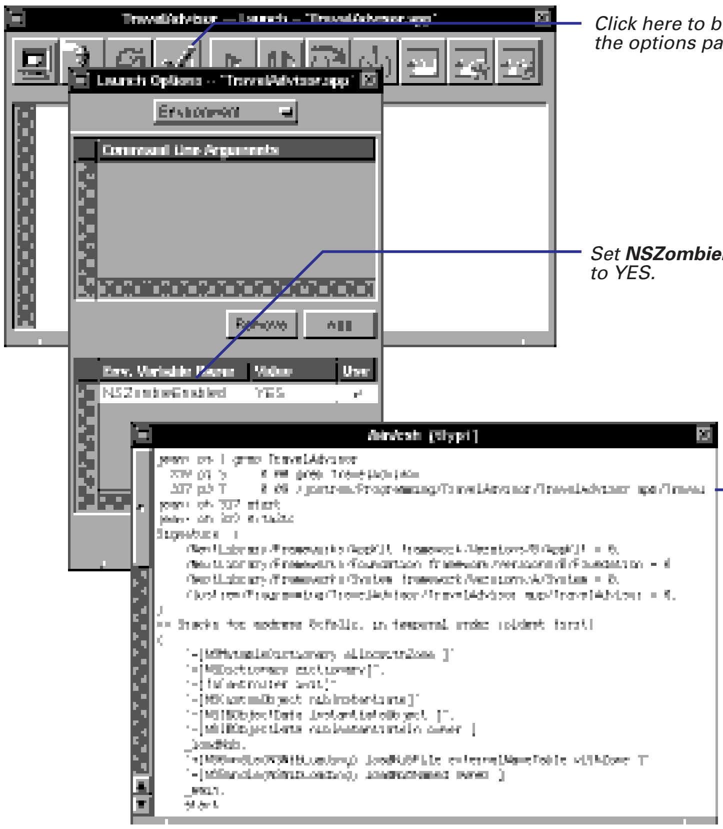
```
#   Name                                  Arguments
0   -[TAController selectFlight:]         (self=0x[illegible], _cmd=0x[illegible]
1   -[TAController addRecord:]            (self=0x[illegible], _cmd=0x[illegible]
2   -[NSObject performSelector:with       ()
3   -[NSApplication sendAction:to:fr      ()
4   -[NSControl sendAction:to:]           ()
5   -[NSCell _sendActionFrom:]            ()
6   -[NSCell trackMouse:inRect:ofVi       ()
7   -[NSButtonCell trackMouse:inRe        ()
8   -[NSControl mouseDown:]               ()
9   -[NSWindow sendEvent:]                ()
10  -[NSApplication sendEvent:]           ()
11  -[NSApplication run]                  ()
12  main                                  [illegible]
```

*Invoke **enableFreedObjectCheck:** and break on **_NSAutoreleaseFreedObject**.*

Jump to the first stack frame that shows code from your program to see which line caused the error.

Using the oh Command

Another way to debug the **autorelease** and **release** errors is to use the **oh** command in conjunction with **gdb**. When you start the **oh** command, it starts recording allocation and deallocation events related to the process you specify. You set **NSZombieEnabled** so that the memory for deallocated objects is not reclaimed. (Released objects are just turned into “zombies.”) The advantage to setting this variable is that you can ensure than an object’s address is unique.


TravelAdvisor — Launch — 'TravelAdvisor.app'
Launch Options -- 'TravelAdvisor.app'
Environment
Command Line Arguments
Remove
Add
Env. Variable Name
Value
Use
/bin/csh (ttyp1)
Click here to bring up the options panel.
*Set **NSZombieEnabled** to YES.*
*In a Terminal window, start **oh** on the process ID of the process you’re debugging.*

Env. Variable Name	Value	Use
NSZombieEnabled	YES	✓

```
jeeves> ps | grep TravelAdvisor
  379 p1 S     0:00 grep TravelAdvisor
  377 p3 T     0:00 /jpatton/Programming/TravelAdvisor/TravelAdvisor.app/Travel
jeeves> oh 377 start
jeeves> oh 377 [illegible]
Signature: (
    /NextLibrary/Frameworks/AppKit.framework/Versions/B/AppKit = 0,
    /NextLibrary/Frameworks/Foundation.framework/Versions/B/Foundation = 0
    /NextLibrary/Frameworks/System.framework/Versions/A/System = 0,
    /jpatton/Programming/TravelAdvisor/TravelAdvisor.app/TravelAdvisor = 0,
)
** Stacks for address 0x[illegible], in temporal order (oldest first)
(
    "-[NSMutableDictionary allocWithZone:]",
    "+[NSDictionary dictionary]",
    "-[[illegible] init]",
    "-[NSCustomObject nibInstantiate]",
    "-[NSIBObjectData instantiateObject:]",
    "-[NSIBObjectData nibInstantiateIn:owner:]",
    _loadNib,
    "+[NSBundle(NSNibLoading) loadNibFile:externalNameTable:withZone:]",
    "-[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
```

When you receive an autorelease error perform the command:

% oh *pid address*

where *address* is address of the object that is being release twice. **oh** will produce a report showing you the stack frame each time that object is allocated, copied, retained, or released, like the one shown on the next page.

```
== Stacks for address 0xfa31c, in temporal order (oldest first):
(
    "+[NSMutableDictionary allocWithZone:]",
    "+[NSDictionary dictionary]",
    "-[TAController init]",
    "-[NSCustomObject nibInstantiate]",
    "-[NSIBObjectData instantiateObject:]",
    "-[NSIBObjectData nibInstantiateIn:owner:]",
    _loadNib,
    "+[NSBundle(NSNibLoading) loadNibFile:...]",
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
(
    "+[NSDictionary dictionary]",
    "-[TAController init]",
    "-[NSCustomObject nibInstantiate]",
    "-[NSIBObjectData instantiateObject:]",
    "-[NSIBObjectData nibInstantiateIn:owner:]",
    _loadNib,
    "+[NSBundle(NSNibLoading) loadNibFile:...]"
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
(
    __NSAPDataReleaseToOffset,
    "-[NSAutoreleasePool release]",
    "+[NSBundle(NSNibLoading) loadNibFile:...]",
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
(
    "-[NSConcreteMutableDictionary release]",
    __NSAPDataReleaseToOffset,
    "-[NSAutoreleasePool release]",
    "+[NSBundle(NSNibLoading) loadNibFile:...]",
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
```

Keeping Memory Allocation Statistics

Another command, **AnalyzeAllocation**, lets you look at memory allocation after your program has finished executing. To use **AnalyzeAllocation**:

1. Set this environment variable:

   ```
   % setenv NSKeepAllocationStatistics YES
   ```

   The **NSKeepAllocationStatistics** variable tells your program to record information about memory allocation in a file named **/tmp/alloc_stats_***name_pid*.

2. Run a specific task in your application. The allocation statistics file becomes very large very quickly, so it is important not to run too much of your program at once with **NSKeepAllocationStatistics** turned on.

3. Turn off the environment variable:

   ```
   % unsetenv NSKeepAllocationStatistics
   ```

4. Perform this command in a Terminal window:

   ```
   % AnalyzeAllocation -v /tmp/alloc_stats_name_pid
   ```

Common Autorelease Mistakes

Once you find the object with the autorelease error, look for the following:

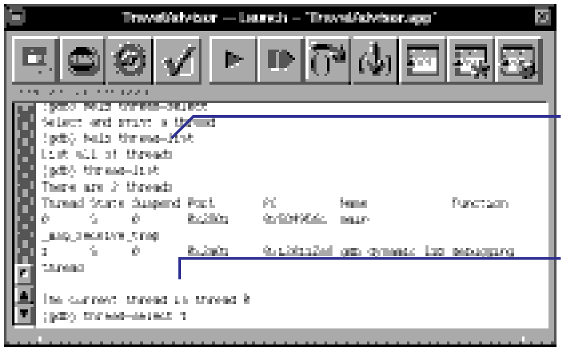
- For every **autorelease** and **release** message in your application, make sure there is a corresponding **alloc**, **copy**, **mutableCopy**, or **retain** message sent to the same object. **autorelease** and **release** decrement an object's reference count. **alloc**, **copy**, **mutableCopy**, and **retain** increment the reference count. The number of increments and decrements for an object must be equal. Another way of thinking about this is: If you don't allocate, copy, or retain an object, you're not responsible for releasing it.
- When an NSArray, NSDictionary, or NSSet (known as the collection objects) is deallocated, the objects stored in the collection are released as well. If you need to access an object you stored in a collection after the collection is released, you must retain that object before you release the collection.
- Superviews retain subviews as you add them to the hierarchy and release subviews as you remove them from the hierarchy. If you swap views in and out of the hierarchy, you should retain the views that are not in the hierarchy.
- When you change a window's content view, the window releases the old content view and retains the new content view.
- Objects do not retain their delegates (to avoid retain cycles).
- **decodeValuesOfObjCTypes:** returns a retained object. **decodeObject** returns an autoreleased object. If you unarchive an instance variable with **decodeObject**, send it the **retain** or **copy** message.

Debugging a multithreaded program

- **Use the gdb command thread-list to obtain information about all of the threads running in the program.**
- **Use the thread-select command to switch to a different thread.**

A single program may have more than one thread of execution. A *thread* is an executable unit that has its own stack and is capable of independent I/O, but shares the address space of the other threads in a *task*.

gdb allows you to observe all threads while your program runs, but whenever **gdb** takes control, one thread in particular is always the focus of debugging. This thread is called the current thread. Debugging commands show program information from the perspective of the current thread. If you want to change to a different thread, use the **thread-select** command (passing it the thread number, which is displayed in the first column of the **thread-list** output).



```
(gdb) help thread-select
Select and print a thread
(gdb) help thread-list
List all of threads
(gdb) thread-list
There are 2 threads
Thread State Suspend Port      PC          Name            Function
0      5     0       [illegible]    [illegible]  main
_msg_receive_trap
1      5     0       [illegible]    [illegible]  gdb dynamic lib debugging
thread

The current thread is thread 0
(gdb) thread-select 1
```

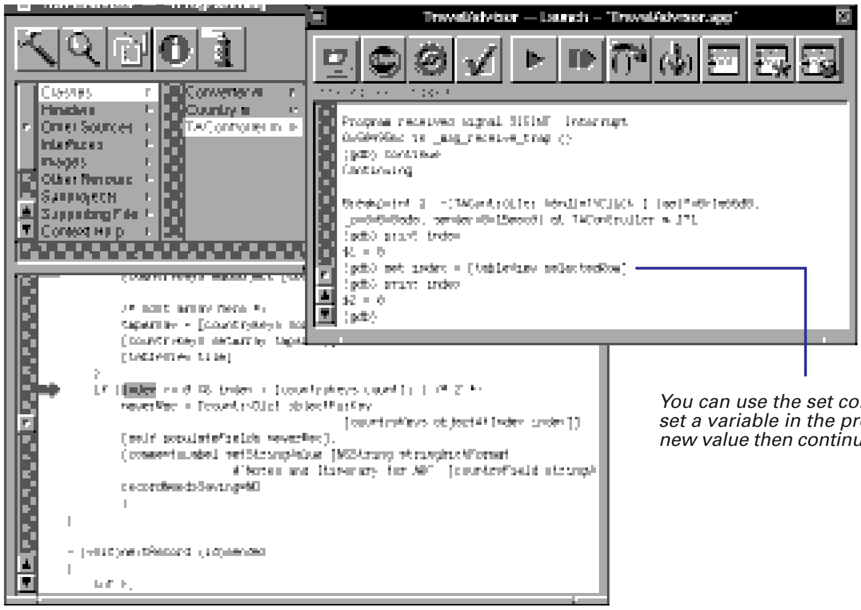
*The **thread-list** command shows information about all of the threads in the program.*

*Use the **thread-select** command to jump to a different thread.*

Changing program execution while debugging

▶ **Use gdb commands to simulate a solution to a bug before building.**

Once you find out what's wrong with your program, you might want to test that the solution you've come up with will work before you change the source code and rebuild. For example, what if you set a variable to a different value? Will that solve the problem?



```
Program received signal SIGINT, Interrupt.
[illegible] in _msg_receive_trap ()
(gdb) continue
Continuing.

Breakpoint 2, -[TAController [illegible]
[illegible] of TAController.m [illegible]
(gdb) print index
$1 = 0
(gdb) set index = [tableView selectedRow]
(gdb) print index
$2 = 0
(gdb)
```

You can use the set command to set a variable in the program to a new value then continue executing.

Command	Description
call *function*	Executes the *function*. You can also use this for Objective-C messages.
jump *linenum*	Resume execution at line number *linenum*. Execution may stop immediately if there's a breakpoint there. The jump command doesn't change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If *linenum* is in a different function from the one currently executing, the results may be wild if the two functions expect different patterns of arguments or of local variables. For this reason, the jump command requests confirmation if the specified line isn't in the function currently executing.
jump **address*	Resume execution at the instruction at address *address*.
set *var* = *exp*	Perform an assignment .

Changing code while debugging

1 **Use the gdb command kill to quit your program in the debugger.**

2 **Make changes to file in Project Builder.**

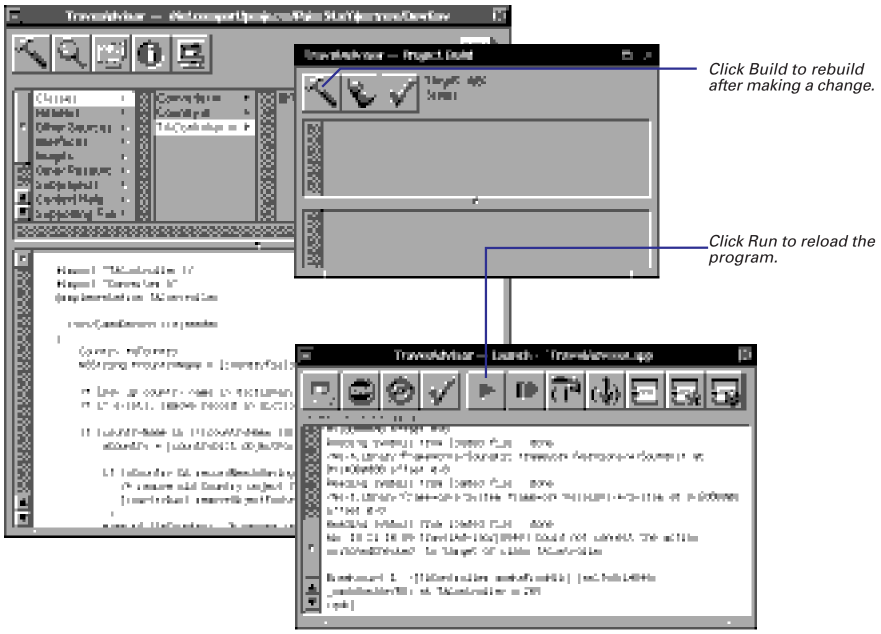
3 **Click the build button to bring up the Project Build panel.**

4 **Click the build button to build the program.**

5 **Go back to the Launch panel.**

6 **Click the debugger's run button.**

After you've found a bug, you need to fix your code and rebuild the program. You don't need to quit the debugger. Just edit the file in Project Builder like you normally would, save it, and rebuild. When the build finished, stop and restart the program in **gdb**. When you click the run button, **gdb** checks for a more recent version of the executable and loads it if necessary. By not quitting **gdb**, you can preserve all of your breakpoints.

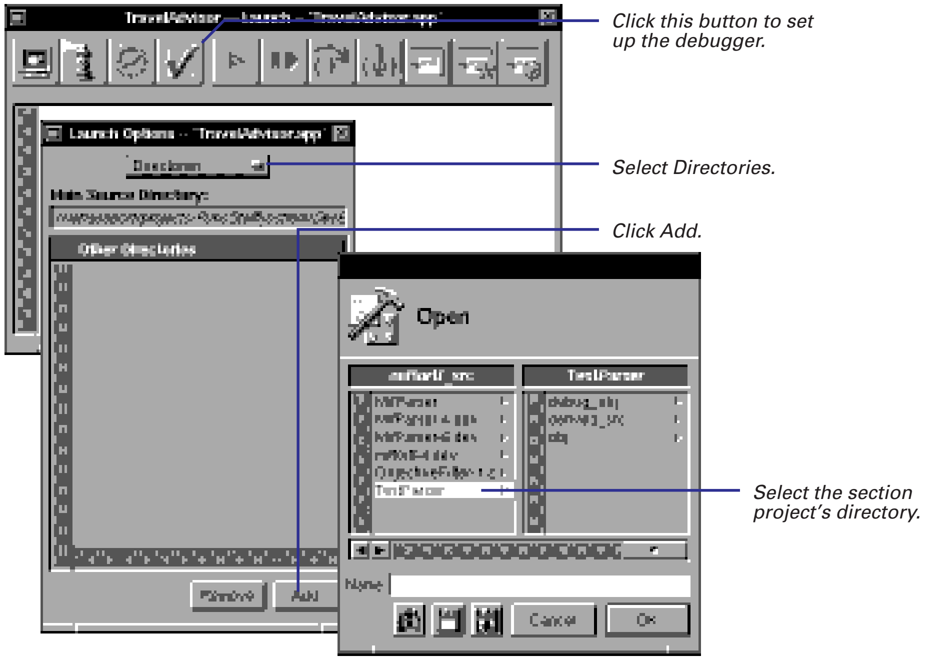

Click Build to rebuild after making a change.
Click Run to reload the program.

Debugging multiple projects

1. **Open the first project.**
2. **Click the checkmark button in the Launch panel.**
3. **In the Directories display of Launch Options panel, click the Add button.**
4. **Select the second project's directory in the Open panel.**
5. **Start the debugger.**

A debugger session applies to only one project file at a time. If you have more than one project open, the Launch panel displays the debugger session for the currently selected project. If you set a breakpoint in one project, it doesn't affect the other project's debugger session.

If you want to debug two projects that relate to each other (for example, and application and a framework), you need to make the debugging symbols of one project visible to the other project. You do this in the Launch Options panel.


Click this button to set up the debugger.
Select Directories.
Click Add.
Select the section project's directory.
Launch Options – 'TravelAdvisor.app'
Main Source Directory:
Other Directories
Remove
Add
Open
Name
Cancel
OK

When you're debugging multiple projects and the debugger stops in code that's part of the unopened project, it displays the appropriate source code file under Non Project Files. If you want to go to other files in the second project and set breakpoints there, open them in the current project window.

Debugging frameworks

1 **Create a tool that tests your framework.**

2 **In the framework project, bring up the Launch Options panel.**

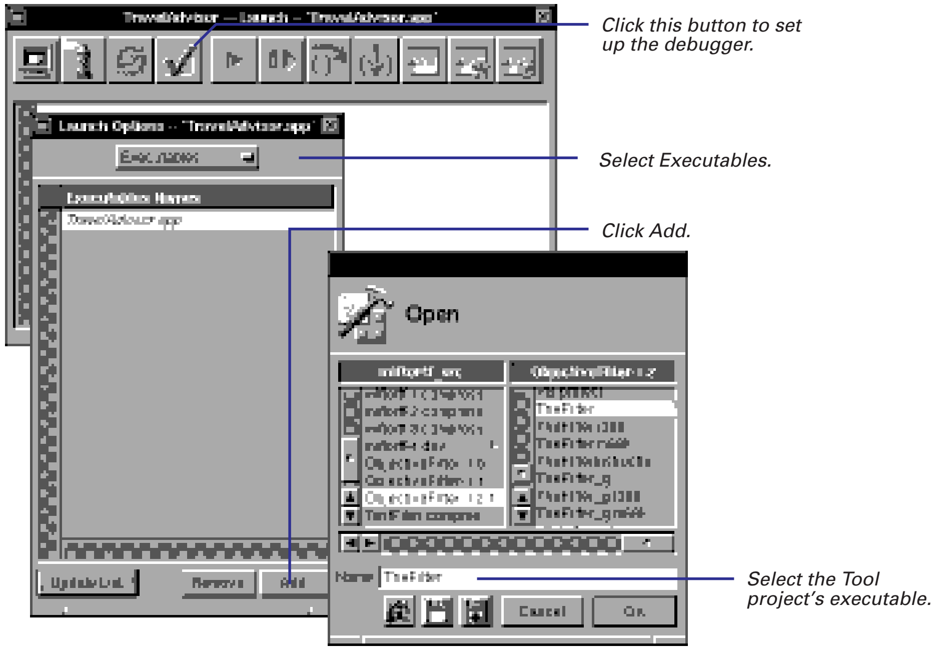
3 **Select Executables.**

4 **Click the Add button.**

5 **Select the tool's executable in the Open panel that appears.**

6 **Start the debugger.**

To debug a framework or library project, you usually create a tool project that uses all of the framework's features. However, you don't really want to debug the tool's code; you want to debug the framework's code. To debug the framework, you can select a tool's executable as the framework's debugger target. When you click the debugger's run button, the tool's executable is what is run. However, you can set breakpoints in the framework's code and step through it, just as if you were debugging an application.


Click this button to set up the debugger.
Select Executables.
Click Add.
Select the Tool project's executable.
Launch Options
Executables
Open
Name
Update List
Remove
Add
Cancel
OK