Debugging in R

1. Basic functions with debugging facility:

trace()
A call to trace allows you to insert debugging code (e.g., a call to browser or recover) at chosen places in any function. A call to untrace cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the function. Trace code can be any **R** expression. Tracing can be temporarily turned on or off globally by calling tracingState.

browser()
Interrupt the execution of an expression and allow the inspection of the environment where browser was called from. A call to browser can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the **R** interpreter.

debug(), undebug()
Set or unset the debugging flag on a function

traceback()
By default traceback() prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. It can also be used to print arbitrary lists of deparsed calls.

2. Debug package:

Debug package is an alternative to trace and browser, offering:

- a visible code window with line-numbered code and highlighted execution point;
- the ability to set (conditional) breakpoints in advance, at any line number;
- the opportunity to keep going after errors;
- multiple debugging windows open at once (when one debuggee calls another, or itself);
- full debugging of on.exit code;
- the ability to move the execution point around without executing intervening statements;
- direct interpretation of typed-in statements, as if they were in the function itself.

Debugging function f is a two-stage process. First, call mtrace(f) to store debugging information on f, and to overwrite f with a debug-ready version that will call the debugger itself. Second, do whatever you normally do at the command prompt to invoke f.
When f is invoked, a window will appear at the bottom of the screen, showing the code of f with a highlight on the first numbered line. (There is also an asterisk in the far left hand column of the same row, showing that there's a breakpoint.) The command prompt in **R** will change to "D(...)> ", showing that you are "inside" a function that is being debugged.

mtrace(f)

First, it stores debugging information about f in tracees$f. The list tracees has one element for each function currently mtraced; it can be shown by search(). This environment lasts until the end of the R session.

Second, mtrace overwrites f with a modified version that calls the debugger. This change is not apparent, however, if you save an mtraced function and then load it into a new R session, you'll see an error when you invoke the function. Before saving, type mtrace(f,F) to untrace f, or mtrace.off() to untrace all tracees.

After a function is read back into R, it will need to be mtraced again. All previous breakpoint will be cleared, and a new one will be set at line 1.

go()

To progress through the code of f, you can keep pressing <ENTER>, but you can also type go(n) or go() (without a line number) to put the debugger into "go mode". Calling go(n) sets a temporary breakpoint at line n, which will be triggered the first time execution reaches line n but not subsequently; go() will keep executing code statements without manual intervention until either:

- the function completes normally, or
- an error occurs in the function, or
- a breakpoint is triggered

In the first case, control is returned to the normal **R** command prompt, just as if the debugger had not been used. In the other cases, the D(...)> prompt will return and the line that caused the error (or that has the breakpoint) will be highlighted in the code window. You are then back in step mode. If there was an error, you can type statement(s) that will cause the error not to happen when the highlighted line executes again, or you can move the highlighted execution point to another line number by calling skip. Execution carries on quite normally after errors, just as if the offending statement had been wrapped in a try call. If your function eventually exits normally (i.e. not via qqq(), as described next), it will be as if the error never happened (though the error message(s) will be displayed when the **R** command prompt returns).

skip()

skip(n) moves the execution point (highlighted in the code window) to line n, without executing any intervening statements. You can skip forwards and backwards. You can skip in and out of loops and conditionals, except that you can't skip into a for loop (the execution point will move to the start of the loop instead). Note that skipping backwards does not undo any statements already executed. skip is useful for circumventing errors, and for ensuring that exit code gets run before calling qqq().

bp()

Breakpoints, including conditional breakpoints, are set and cleared by bp(). bp(n) will set a breakpoint at line n, as shown by the red asterisk that appears to the left of line n. The breakpoints can be cleared by bp(n,F).

Conditional breakpoints can be set by e.g. bp(n, i<=j). The second argument is an (unquoted) expression which will be evaluated in the function frame whenever line n is reached; if the

expression evaluates to anything except FALSE, the breakpoint will trigger. Such breakpoints are mainly useful inside loops, or when a function is to be called repeatedly.
Breakpoints are usually set inside the debugger, after the code window has appeared. However, they can also be set in advance. Unless you are clearing/setting at line 1, you will need to check the line list, which can be seen in tracees$f$line.list.

qqq()
When in step mode, you can finish debugging and return to the normal **R** command prompt by typing qqq(). <ESC> also seems to work, but qqq() is probably safer.

3. Debugging strategies

If there is an error,
Call traceback() to see where the error occurred; often, this is not in your own functions. Then call mtrace either on your function, or on the highest-numbered function from traceback. Next, type go() when the code window appears, and wait for the crash. The debugger will highlight the error line, and your can inspect variables. If it’s still not obvious why the crash occurred, mtrace whichever function is invoked by the error line, and repeat the similar procedures.
If there’s no error but f is giving strange results,
Mtrace and invoke f, and step through the code with <ENTER>, watching intermediate results and printing any variables you are curious about. If there is an unproblematic block of code at the start of f, then it’s faster to use go(n) (or bp(n) followed by go()) to get to line number n after the block. Once you understand roughly where the problem is, you can either qqq() or patch things up, by setting variables directly from the keyboard and/or using skip, just to check that the next few statements after the bug are OK.

Reference:
1. How to use the debug package, R documentation.
2. Mark Bravington, Debugging Without (Too Many) Tears, The Newsletter of the R Project Volume 3/3, December 2003.