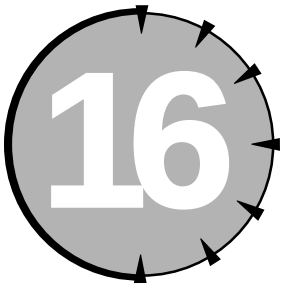
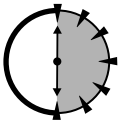
SESSION



Debugging II

Session Checklist

- ✔ Stepping through a program
- ✔ Setting breakpoints
- ✔ Viewing and modifying variables
- ✔ Debugging a program using the debugger



30 Min. To Go

Session 10 presented a technique for debugging programs based on writing key data to `cout` standard output. We used this so-called write-statement technique to debug the admittedly very simple ErrorProgram example.

For small programs the write statement technique works reasonably well. Problems with this approach don't really become obvious until the size of the program grows beyond the simple programs you've seen so far.

In larger programs, the programmer often doesn't generally know where to begin adding output statements. The constant cycle of add write statements, execute the program, add write statements, and on and on becomes tedious. Further, in order to change an output statement, the programmer must rebuild the entire program. For a large program, this rebuild time can itself be significant.

A second, more sophisticated technique is based on a separate utility known as a debugger. This approach avoids many of the disadvantages of the write-statement

approach. This session introduces you to the use of the debugger by fixing a small program.

Much of this part of the book is dedicated to studying the programming capabilities made possible by pointer variables. However, pointer capabilities come at a price: pointer errors are easy to make and extremely hard to find. The write-statement approach is not up to the task of finding and removing pointer problems. Only a good debugger can ferret out such errors.

Which Debugger?

Unlike the C++ language, which is standardized across manufacturers, each debugger has its own command set. Fortunately, most debuggers offer the same basic commands. The commands we need are available in both the Microsoft Visual C++ and the GNU C++ rhide environments. Both environments offer the basic command set via drop-down menus. In addition, both debuggers offer quick access to common debugger commands via the function keys. Table 16-1 lists these commands in both environments.

For the remainder of this session, I refer to the debug commands by name. Use Table 16-1 to find the corresponding keystroke to use.

Table 16-1
Debugger Commands for Microsoft Visual C++ and GNU rhide

Command	Visual C++	GNU C++ (rhide)
Build	Shift+F8	F9
Step in	F11	F7
Step over	F10	F8
View variable	see text	Ctl+F4
Set breakpoint	F9	Ctl+F8
Add watch	see text	Ctl+F7
Go	F5	Ctl+F9
View User Screen	Click on Program Window	Alt+F5
Program reset	Shift+F5	Ctl+F2

To avoid confusion over the slight differences that exist between the two debuggers, I describe the debug process I used with rhide first. I then debug the program using more or less the same steps with the Visual C++ debugger.

The Test Program

I wrote the following "buggy" program. Writing a buggy program is particularly easy for me because my programs rarely work the first time.

This file is contained on the accompanying CD-ROM in the file Concatenate (Error).cpp.

```
// Concatenate - concatenate two strings
//               with a " - " in the middle
//               (this version crashes)
#include <stdio.h>
#include <iostream.h>

void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    cout << "This program concatenates two strings\n";
    cout << "(This version crashes.)\n\n";

    // read first string...
    char szString1[256];
    cout << "Enter string #1:";
    cin.getline(szString1, 128);

    // ...now the second string...
    char szString2[128];
    cout << "Enter string #2:";
    cin.getline(szString2, 128);

    // ...concatenate a " - " onto the first...
    concatString(szString1, " - ");
```

```
    // ...now add the second string...
    concatString(szString1, szString2);

    // ...and display the result
    cout << "\n" << szString1 << "\n";

    return 0;
}

// concatString - concatenate the string szSource
//                to the end of szTarget
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex;
    int nSourceIndex;

    // find the end of the first string
    while(szTarget[++nTargetIndex])
    {
    }

    // tack the second to the end of the first
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }
}
```

The program builds without problem. I next execute the program. When it asks for string #1, I enter this is a string. For string #2, I enter THIS IS A STRING. Rather than generate the proper output, however, the program terminates with an exit code of 0xff. I click OK. In an attempt to offer some solace, the debugger opens the Message Window underneath the edit window shown in Figure 16-1.

The first line of the message window indicates that rhide thinks that the error occurred on or about line 46 of the module `Concatenate(error1)`. In addition, the function that crashed was called from line 29 of the same module. This would seem to indicate that the initial `while` loop within `concatString()` is faulty.

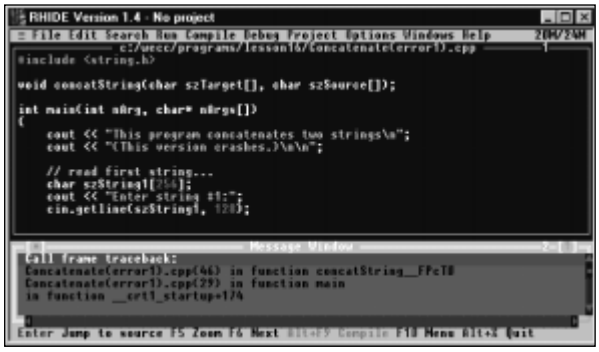
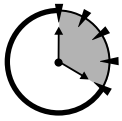


Figure 16-1
The rhide debugger gives some hint as to the source of the error when a program crashes.

Because I don't see any problem with the statement, I decide to bring the rhide debugger to my aide.

Actually, I see the problem based on the information that rhide has already provided, but work with me here.



Single-Stepping Through a Program

I press Step Over to begin debugging the program. rhide opens an MS-DOS window as if it were about to execute the program; however, before the program can execute, the debugger closes the program window and displays the program edit window with the first executable line of the program highlighted.

An *executable statement* is a statement other than a declaration or a comment. An executable statement is one that generates machine code when compiled.

The debugger has actually executed the program up through the first line of the `main()` function and then snatched control of the program. The debugger is waiting for you to decide what to do next.

By repetitively pressing Step Over I can execute through the program until it crashes. This should tell me a lot about what went wrong.

Executing a program one line at a time is commonly known as *single-stepping* the program.

When I try to Step Over the `cin.getline()` command, the debugger does not take control back from MS-DOS window as it normally would. Instead, the program appears to be frozen at the prompt to enter the first string.

On reflection, I realize that the debugger does not take control back from the program until the C++ statement finishes executing — the statement containing the call to `getline()` cannot finish until I enter a string of text from the keyboard.

I enter a line of text and press Enter. The rhide debugger stops the program at the next statement, the `cout << "Enter string #2"`. Again I single–step, entering the second line of text in response to the second call to `getline()`.

If the debugger seems to halt without returning when single-stepping through a program, your program is waiting for something to happen. Most likely, the program is waiting for input, either from you or from external device.

Eventually I single-step down to the call to `concatString()`, as shown in Figure 16-2. When I try to Step Over the call, however, the program crashes as before.

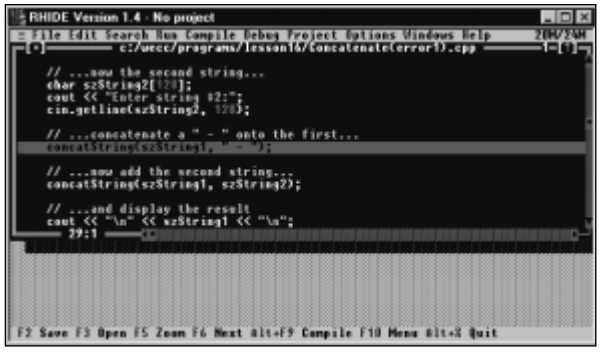


Figure 16-2
Something in the concatString() function causes the program to crash.

This doesn't tell me a lot more than I knew before. What I need is to be able to execute into the function rather than simply "stepping over" it.

Single-Stepping into a Function

I decide to start over. First, I press Program Reset in order to start the debugger at the beginning of the program.

Always remember to press Program Reset before starting over. It doesn't hurt to press the key too often. You might get into the habit of entering Program Reset before starting in the debugger every time.

Again, I single-step through the program using the Step Over key until I reach the call to `concatString()`. This time rather than step over the call, I use the Step In command to move into the function. Immediately the pointer moves to the first executable line in `concatString()` as shown in Figure 16-3.

There is no difference between the Step Over and Step In commands when not executing a function call.

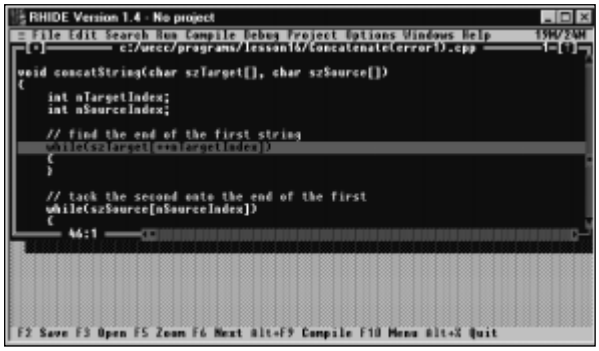


Figure 16-3
The Step In command moves control to the first executable line in concatString().

If you accidentally Step In to a function that you did not mean to, the debugger may ask you for the source code to some file that you've never heard of before. This asked-for file is the library module that contains the function you just stepped into. Press Cancel to view a listing of machine instructions, which are not very useful to even the most hardened techies. To return to sanity, open the edit window, set a break point as described in the next section to the statement after the call, and press Go.

With high hopes, I press the Step Over key to execute the first statement in the function. The rhide debugger responds by reporting a Segmentation violation as shown in Figure 16-4.

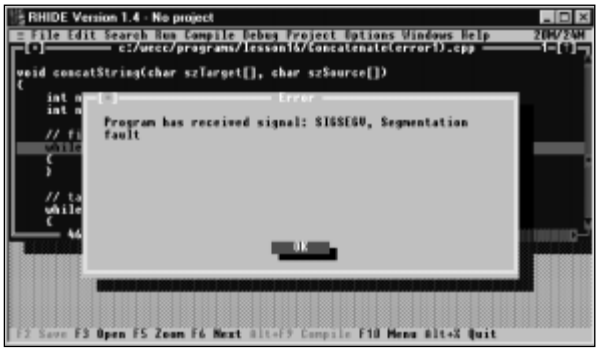


Figure 16-4
Single-stepping the first line of the concatString() function generates a Segmentation violation.

A segmentation violation generally indicates that the program has accessed an invalid section of memory either because a pointer has gone awry or because an array is being addressed way beyond its bounds. To keep things interesting, let's pretend that I don't know that.

Now I know for sure that something about the `while` loop is not correct and that executing it even the first time crashes the program. To find out what it is, I need to stop the program right before it executes the offending line.

Using Breakpoints

Again, I press Program Reset to move the debugger back to the beginning of the program. I could single-step back through the program to the `while` loop as I did before. Instead, I decide to employ a shortcut. I place the cursor on the `while` statement and enter the Set breakpoint command. The editor highlights the statement in red as shown in Figure 16-5.

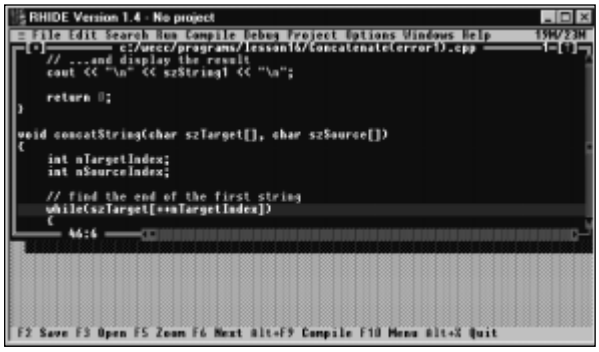
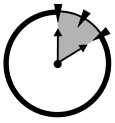


Figure 16-5
The breakpoint command

A *breakpoint* tells the debugger to halt on that statement if control ever passes this way. A breakpoint enables the program to execute normally up to the point that we want to take control. Breakpoints are useful either when we know where to stop or when we want the program to execute normally until it's time to stop.

With the breakpoint set, I press Go. The program appears to execute normally up to the point of the `while` call. At that point, the program obediently jumps back to the debugger.



10 Min. To Go

Viewing and Modifying Variables

There isn't much point in executing the `while` statement again—I know that it will crash. I need more information about what the program is doing to determine why it crashed. For example, I would like to see the value of `nTargetIndex` immediately prior to the execution of the `while` loop.

First, I double-click the variable name nTargetIndex. Next, I press View Variable. A window appears with the name nTargetIndex in the upper field. I click Eval in order to find the current value of the variable. The results, which are shown in Figure 16-6, are obviously nonsensical.

Looking back at the C++ code, I realize that I neglected to initialize either the `nTargetIndex` or `nSourceIndex` variables. To test this theory, I enter a 0 in the New Value window and click Change. I repeat the process for `nSourceIndex`. I then close the window and click Step Over to continue executing.

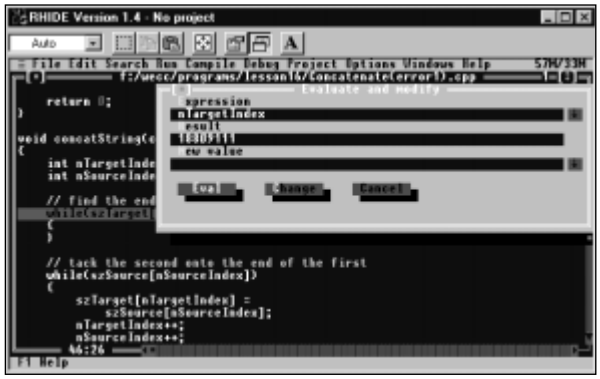


Figure 16-6
The Evaluate and Modify window enables the programmer to evaluate and optionally change the value of in-scope variables.

With the index variables initialized, I single-step into the `while` loop. The program does not crash. Each Step Over or Step In command executes one iteration of the `while` loop. Because the cursor ends up right where it started, there appears to be no change; however, after one loop `nTargetIndex` has incremented to 1.

Because I don't want to go through the work of reevaluating `nTargetIndex` on each iteration, I double-click on `nTargetIndex` and enter the Add Watch command. A window appears with the variable `nTargetIndex` and the value 1 to the right. I press Step In a few more times. `nTargetIndex` increments on each iteration through the loop. After several iterations, control eventually passes outside of the loop.

I set a breakpoint on the closing brace of the `concatString` function and press Go. The program stops immediately prior to returning from the function.

To check the string generated, I double-click on `szTarget` string and press View Variable. The results shown in Figure 16-7 are not what I expected.

The 0x6ccee0 is the address of the string in memory. This information can be useful when tracking pointers. For example, this information would be extremely helpful in debugging a linked-list application.

It would appear as if the target string has not been updated, yet I know that the second `while` loop was executed. On the off chance that the second string is in fact there, I attempt to look past the initial string. `szTarget + 17` should be the address of the first character after the `null` at the end of "this is a string," the value I entered. In fact, the " - " appears there, followed by a "T" which appears to be incorrect. This is shown in Figure 16-8.

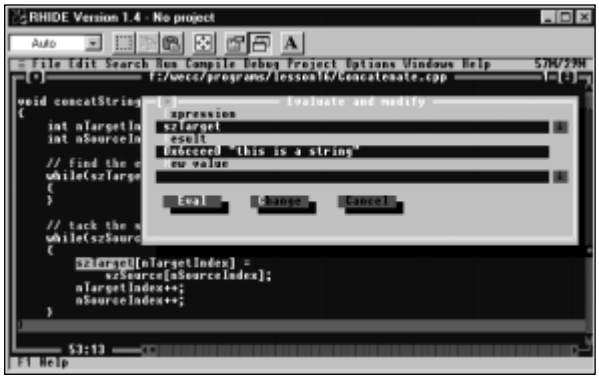


Figure 16-7
The target string does not appear to be modified upon returning from the concatString() function.

Figure 16-8
The target string appears appended to the source string in the wrong place.

After careful consideration, it is obvious that `szSource` was appended to `szTarget` after the terminating `null`. In addition, it is clear that the resulting target string was not terminated at all (hence the extra "T" at the end).

Modifying a string after the terminating `null` or forgetting to terminate a string with a `null` are by far the two most common string-related errors.

Because I now know two errors, I press Program Reset and fix the `concatString()` function to append the second string in the correct place and to append a terminating null to the resulting string. The updated `concatString()` function appears as follows:

```
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex = 0;
    int nSourceIndex = 0;

    // find the end of the first string
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // tack the second onto the end of the first
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }

    // terminate the string properly
    szTarget[nTargetIndex] = '\0';
}
```

Because I suspect that I may still have a problem, I set a watch on `szTarget` and `nTargetIndex` while executing the second loop. Sure enough, the source string appears to be copied to the end of the target string as shown in Figure 16-9. (The second call to `concatString()` is shown in Figure 16-9 because it is the more obvious to understand.)

You really need to execute this one yourself. It's the only way you can get a feel for how neat it is to watch one string grow while the other string shrinks on each iteration through the loop.

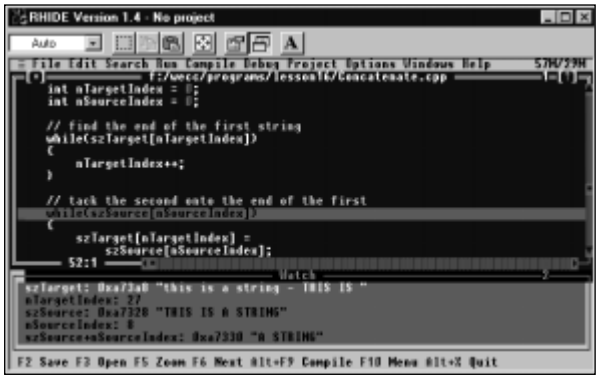


Figure 16-9

The appropriate watch variables demonstrate how the source string is appended to the end of the target string.

Reexamining the string immediately prior to adding the terminating `null`, I notice that the `szTarget` string is correct except for the extra characters at the end as shown in Figure 16-10.

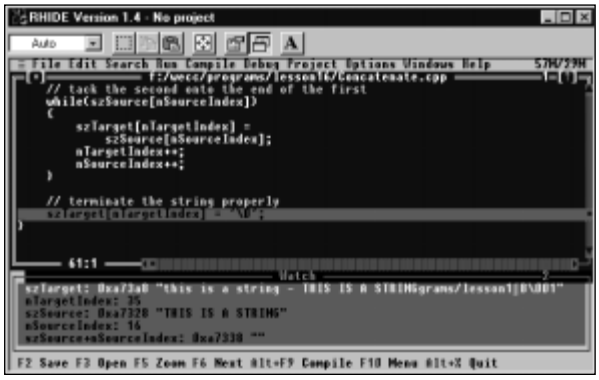


Figure 16-10

Prior to adding the terminating null, the string resulting from concatenating the target and source strings displays extra characters.

As soon as I press Step Over, the program adds the terminating `null` and the noise characters disappear from the watch window.

Using the Visual C++ Debugger

The steps used to debug the `Concatenate` program using the Visual C++ debugger are similar to those used under rhide. A major difference, however, is that the Visual C++ debugger opens the program under execution in a separate MS-DOS window rather than as part of the debugger itself. When you press Go, a new tab appears along the Windows Task Bar bearing the name of the program, in this case Concatenate. The programmer can view the user window by selecting the program window.

A second difference is the way that the Visual C++ debugger handles the viewing of local variables. When execution is halted at a breakpoint, the programmer may simply place the cursor on a variable. If the variable is in scope, the debugger displays its value in a popup window as shown in Figure 16-11.

Figure 16-11
Visual C++ displays the value of a variable by placing the cursor on it.

In addition, the Visual C++ debugger offers a convenient view of all "local" variables (these are variables declared locally to the function). Select View, then Debug Windows, and finally Variables. From within the Variables Window, select the Locals tab. Alternatively, you may enter Alt+4.This window even highlights the variable which has changed since the last break point. Figure 16-12 shows this window while single-stepping through the copy of the source to the destination string.

The "I" characters at the end of `szTarget` reflect the fact that the string has yet to be terminated.

Figure 16-12
The Variables window in the Visual C++ debugger tracks values of variables as you single-step through the code.

REVIEW

Let's compare the debugger approach to finding problems as shown here with the write approach demonstrated in Session 10. The debugger approach is not as easy to learn. I'm sure that many of the commands entered here seem foreign. Once you feel comfortable with the debugger, however, you can use it to learn a great deal about your program. The capability to move slowly through a program while viewing and modifying different variables is a powerful feature.

I prefer to use the debugger the first time I execute a newly written program. Stepping through a program provides a good understanding of what's really going on.

I was forced to restart the program several times using the debugger approach; however, I only edited and rebuilt the program once, even though I found more than one problem. This is a distinct advantage when debugging a large program that may take several minutes to rebuild.

I have been on projects where it took the computer the entire night to rebuild the system. While this was an extreme case, 5 to 30 minutes per rebuild is not out of the ordinary in real-world applications.

Finally, the debugger gives you access to information that you couldn't easily see using the write approach. For example, viewing the contents of a pointer is straightforward using the debugger. While possible, it is clumsy to repeatedly write out address information.

QUIZ YOURSELF

1. What is the difference between Step Over and Step In? (See "Single-Stepping into a Function.")
2. What is a breakpoint? (See "Using Breakpoints.")
3. What is a watch? (See "Viewing and Modifying Variables.")