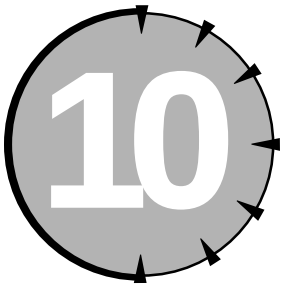
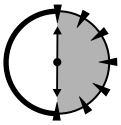
SESSION



Debugging I

Session Checklist

- ✔ Differentiating types of errors
- ✔ Understanding the "crash messages" returned by C++ environment
- ✔ Mastering the write-statement technique of debugging
- ✔ Debugging both Visual C++ and the GNU/C++



30 Min. To Go

You may have noticed that the programs that you write as part of the exercises in earlier chapters often don't work the first time. In fact, I have seldom, if ever written, a nontrivial C++ program that didn't have some type of mistake the first time I tried to execute it.

A program that works the first time you try to execute it is known as a *gold-star program*.

Types of Errors

There are two different types of errors. Those errors that the C++ compiler can catch are relatively easy to fix. The compiler generally points you to the problem. Sometimes the description of the problem is not quite correct — it's easy to

confuse a compiler — but once you learn the quirks of your own C++ package, it isn't too difficult.

A second set of errors is comprised of those errors that the compiler can't catch. These errors first show up as you try to execute the program. Errors that are first noticed when you execute the program are known as *run-time errors*. Those errors that C++ can catch are known as *compile-time errors*.

Run-time errors are much more difficult to find, because you have no hint of what's gone wrong except for whatever errant output the program might generate.

There are two different techniques for finding bugs. You can add output statements at key points and rebuild the program. You can get an idea of what's gone wrong with your program as these different output statements are executed. A second approach that is more powerful is to use a separate program called a debugger. A debugger enables you to control your program as it executes. This session examines the "output statement" technique. In Session 16, you learn to use the debugger that is built in Visual C++ and GNU C++.

The Output-Statement Technique

The approach of adding output statements to the code to debug is known as the outputstatement approach.

This technique is often called the writestatement approach. It gained this name back in the days of early programs, which were written in FORTRAN. FORTRAN's output is through its `WRITE` statement.

To see how this might work, let's fix the following buggy program.

```
// ErrorProgram - this program averages a series
//                of numbers, except that it contains
//                at least one fatal bug
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << "This program is designed to crash!\n";

    // accumulate input numbers until the
    // user enters a negative number, then
    // return the average
```

```
    int nSum;
    for (int nNums = 0; ;)
    {
        // enter another number to add
        int nValue;
        cout << "\nEnter another number:";
        cin  > nValue;

        // if the input number is negative...
        if (nValue < 0)
        {
            // ...then output the average
            cout << "\nAverage is: "
                 << nSum/nNums
                 << "\n";
            break;

        }

        // not negative, add the value to
        // the accumulator
        nSum += nValue;
    }
    return 0;
}
```

After entering this program, I build the program to generate the executable ErrorProgram.exe like usual. Impatiently, I guide the Window Explorer to the folder containing the program and confidently double-click ErrorProgram to begin executing it. I enter the usual values of 1, 2, and 3 followed by –1 to terminate input. Instead of producing the much-anticipated value of 2, the program terminates with the not very friendly error message shown in Figure 10-1 and without any output message at all.

Don't execute ErrorProgram from the C++ environment just yet.

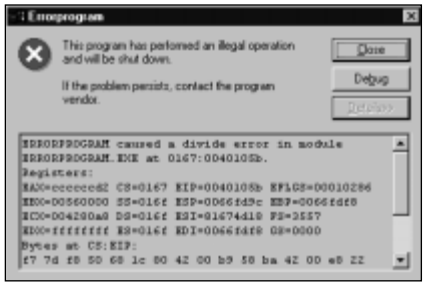
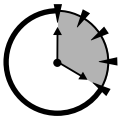


Figure 10-1
The initial version of ErrorProgram terminates suddenly instead of generating the expected output.



20 Min. To Go

Catching Bug #1

The error message shown in Figure 10-1 displays the ubiquitous "This program has performed an illegal operation. . ." error message. Even with the Registers information, you have nothing to start debugging with.

Actually there is one little tidbit of information: at the top of the error display appears the message "ERRORPROGRAM caused a divide error." This wouldn't normally be much help except that there's only one divide in the entire program, but let's ignore that for the sake of argument.

Hoping to get more information, I return to execute ErrorProgram from within the C++ environment. What happens next is slightly different in Visual C++ than GNU C++.

Visual C++

I enter the same 1, 2, 3, and –1 values as I did before in order to force the program to crash.

One of the first things you would like to do when tracking down a problem is to find the set of operations that cause the program to fail. By reproducing the problem, you know not only how to recreate it for debug purposes, but you also know when it's fixed.

Visual C++ generates the output shown in Figure 10-2. This window indicates that the program has terminated abnormally (that's the Unhandled exception part) because of a Divide by Zero error.

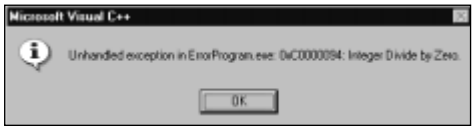


Figure 10-2
The error message from Visual C++ is slightly better than the Windows error message.

The Divide by Zero error message is only slightly better than the division error message shown in Figure 10-1; however, when I click OK, Visual C++ displays the ErrorProgram program with a yellow arrow pointing squarely at the division as shown in Figure 10-3. This is the C++ line of code on which the error originated.

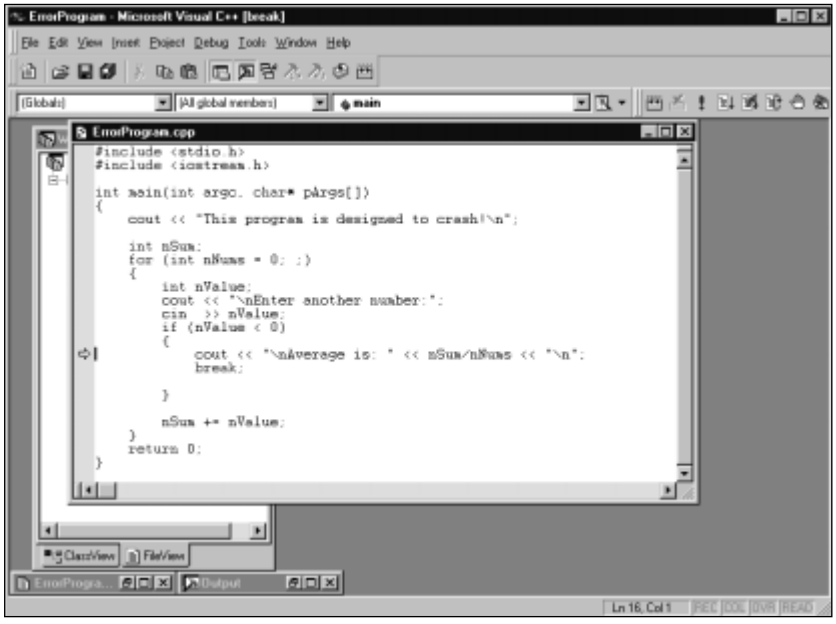


Figure 10-3
Visual C++ points the blame clearly at the division using nNums.

Now I know that at the time of the division, nNums must have been equal to 0. (Otherwise, I wouldn't have gotten a Divide by Zero error.) I can see where nNums is initialized to 0, but where is it incremented? It isn't, and this is the bug. Clearly nNums should have been incremented in the increment clause of the for loop.

To fix the problem, I replace the original for with the following:

```
for (int nNums = 0; ;nNums++)
```

GNU C++

Tracking the crash from using the GNU C++ suite is similar. From within rhide, I run the program. In response to the 1, 2, 3, and –1 input, ErrorProgram terminates with the message shown in Figure 10-4. Instead of the normal exit code of 0x00 (0 decimal), I now see the abnormal exit code of 0xff (or 255 decimal). This doesn't tell me anything other than that there was an error of some type encountered when executing the program (that much I knew already).

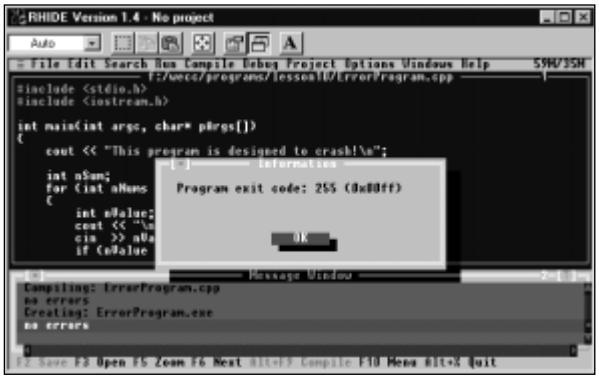


Figure 10-4
The exit code for the ErrorProgram in rhide is 0xff.

After clicking OK, however, I notice that rhide has opened a small window at the bottom of the display. This window, shown in Figure 10-5, tells me that the error was generated at ErrorProgram.cpp(16) in function main. This slightly cryptic message indicates that the error occurred on line 16 of ErrorProgram.cpp and that this line is in the function main() (the latter I could tell by looking at the listing, but it's nice information anyway).

From this I know all that I need to know to solve the problem the same way I did in the Visual C++ case.

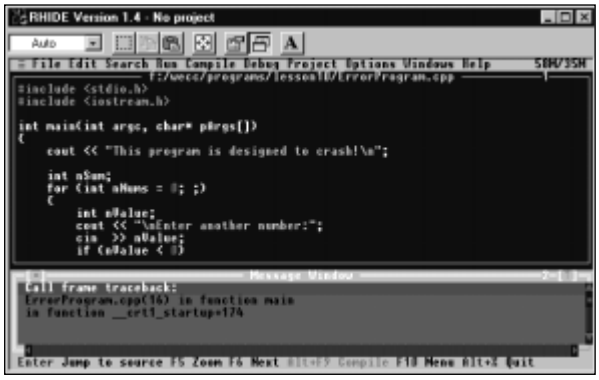


Figure 10-5
The error message from rhide is just as informative as that from Visual C++, albeit slightly cryptic.

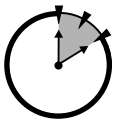
How Can C++ Tie an Error Message Back to the Source Code?

The information I received when executing the program directly from Windows or from an MS-DOS window was not very informative. By comparison, both Visual C++ and `rhide` were able to direct me to the line from whence the problem originated. How did they do that?

C++ has two modes when building a program. By default, C++ builds the program in what is called debug mode. In debug mode, C++ adds line-number information that maps the lines of C++ code to the corresponding lines of machine code. For example, this map might say that line 200 of machine code in the executable code was created from the C++ source code on line 16.

When the Divide by Zero error occurred, the operating system knows that the error occurred on offset 0x200 of the executable machine code. C++ tracks this back to line 16 using the debug information.

As you might imagine, this debug information takes a lot of space. This makes the executable file larger and slower. To address this problem, both GNU and Visual C++ support a build mode called release mode in which the debug information is stripped out.



10 Min. To Go

Catching Bug #2

Flush with success and pride, I confidently execute the program using the same 1, 2, 3, and –1 input that caused the program to crash earlier. This time, the program doesn't crash, but it doesn't work either. The output shown below is ridiculous:

```
This program is designed to crash!

Enter another number:1

Enter another number:2

Enter another number:3

Enter another number:-1

Average is: -286331151
Press any key to continue
```

Apparently, either `nSum` or `nNums` (or both) is not being calculated properly. To proceed, I need to know the value of these variables. In fact, it would help if I knew the value of `nValue` as well, because `nValue` is used to calculate `nSum`.

To learn the values of the `nSum`, `nNums` and `nValue`, I modify the `for` loop as follows:

```
for (int nNums = 0; ;nNums++)
{
    int nValue;
    cout << "\nEnter another number:";
    cin  > nValue;
    if (nValue < 0)
    {
        cout << "\nAverage is: " << nSum/nNums << "\n";
        break;

    }

    // output critical information
```

```
        cout << "nSum = " << nSum   << "\n";
        cout << "nNums= " << nNums  << "\n";
        cout << "nValue= "<< nValue << "\n";
        cout << "\n";

        nSum += nValue;
    }
```

Notice the addition of the output statements. These three lines display the value of `nSum`, `nNums`, and `nValue` on each iteration through the loop.

The results of executing the program with the now standard 1, 2, 3, and –1 input sequence are shown below. Even on the first loop, the value of `nSum` seems unreasonable. In fact, at this point during the first loop, the program has yet to add a new value to `nSum`. At this point, I would think that the value of `nSum` should be 0. That appears to be the problem.

```
This program is designed to crash!

Enter another number:1
nSum = -858993460
nNums= 0
nValue= 1

Enter another number:2
nSum = -858993459
nNums= 1
nValue= 2

Enter another number:3
nSum = -858993457
nNums= 2
nValue= 3

Enter another number:
```

On careful examination of the program, `nSum` is declared but it is not initialized to anything. The solution to is change the declaration of `nSum` to the following:

```
int nSum = 0;
```

Until a variable has been initialized, the value of that variable is indeterminate.

Once I have convinced myself that the result is correct, I "clean up" the program as follows:

```
// ErrorProgram - this program averages a series
//                of numbers
//                (This version has been fixed.)
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << "This program works!\n";

    // accumulate input numbers until the
    // user enters a negative number, then
    // return the average
    int nSum = 0;
    for (int nNums = 0; ;nNums++)
    {
        // enter another number to add
        int nValue;
        cout << "\nEnter another number:";
        cin  > nValue;

        // if the input number is negative...
        if (nValue < 0)
        {
            // ...then output the average
            cout << "\nAverage is: " << nSum/nNums << "\n";
            break;
        }

        // not negative, add the value to
        // the accumulator
```

```
            nSum += nValue;
        }
        return 0;
    }
```

I rebuild the program and retest with the 1, 2, 3, and –1 sequence. This time I see the expected average value of 2:

```
This program works!

Enter another number:1

Enter another number:2

Enter another number:3

Enter another number:-1

Average is: 2
```

Done!

After testing the program with a number of other inputs, I convince myself that the program is now executing properly. I remove the extra output statements and rebuild the program to complete the debugging of the program.

REVIEW

There are two basic types of errors: compile-time errors generated by the C++ compiler when it encounters an illogical coding construct, and run-time errors created when the program executes some illogical sequence of legal instructions.

Compile-time errors are relatively easy to track down because the C++ compiler is good at pointing you right to the error. Run-time errors are more difficult. The Visual C++ and GNU/C++ environments try to give you as much help as possible. In the example program in this session, they actually do a good job at pointing to the problem.

When the run-time error messages generated by the C++ environment are not enough, it's left to the programmer to find the problem by debugging the code. In this session, we used the so-called write-statement technique.

- C++ compilers are very fussy in what they accept in order to be capable of rooting out programmer errors during the build process where things are easier to fix.
- The operating system tries to catch run-time errors as well. When these errors call the program to crash, the operating system returns error information that the Visual C++ and GNU/C++ environments try to interpret.
- Output statements generated at critical points in the program can vector the programmer to the source of run-time errors.

Although crude, the write-statement technique is effective on small programs. Session 16 demonstrates more effective debugging techniques.

QUIZ YOURSELF

1. What are the two basic types of errors? (See "Types of Errors.")
2. How can output statements be used to help find run-time errors? (See "The Output-Statement Technique.")
3. What is debug mode? (See "How Can C++ Tie an Error Message Back to the Source Code?")