DEBUGGING

INTRODUCTION

Debugging† is the art of finding the errors in program source code that prevent the program form working according to its specification. In a sense, it is the culmination of programming methodology. It brings together the skills introduced earlier--code reading and hand simulation, in conjunction with the design that led to the program in the first place, and the testing of it once it was written. It is the hardest part of programming, and the part that requires the most experience. Good code writers can be poor debuggers. Code writing requires constructive, synthesis-oriented skills; debugging requires deconstructive, analytic skills. People who are good at synthesis can be poor at analysis, although the reverse is rarely the case. Good programmers eventually become good at both sorts of skill.

WHY PROGRAMS DO NOT WORK

If we could guarantee that programs that we write from a good, careful design would work first time, there would be no need for debugging. There would still be a need for testing, however, because of the possibility that the behavior of the program does not match its specification. Programs that do not work fail for an infinite variety of reasons, but the *types* of failure are very few:

1. The crash, giving a message like “segmentation error”, or “general processor fault”. sometimes this type of failure will “dump core”, i.e. record an image of the executing program in a disk file. These are usually very large, possibly many megabytes.

2. The lock-up, when the whole computer becomes frozen. Nothing will work, and it is almost always the case that furiously pressing keys and clicking the mouse button just makes it worse.

3. The infinite loop. This can occur with output messages, when it is pretty obvious what is going on, or without messages when it looks like a lock-up. Usually pressing a special “interrupt” key (ctrl-C on UNIXand DOS) will break into the loop and terminate the program.

4. The system exception message. If you are lucky the operating system will catch an error before it ruins the execution, but these are very few and far between. Dividing a number by zero will produce this kind of error, as will trying to read past the end of a file. Programs that use a windowing system (such as X Windows, or MS Windows) are an order of magnitude more complex than “ordinary” programs and often produce system exceptions because the program is abusing the normal way that windows are created and managed. A common one in this case is “out of resources”, which means that that limited portion of memory allocated to interface objects has been exhausted.

† The word bug refers to a chance occurrence in one the very old computers that used vacuum tubes and lots of electric power. A faulty program in one of these behemoths was traced to a squashed bug shorting out one of the circuit boards. Although this was clearly a hardware bug, the term is now used for software as well.

5. Bad, strange, or missing output. This is almost the fault of the design of the program, or a misunderstanding of either program constructs or library procedures. These errors are perhaps the most common, and, fortunately among the easiest to catch.

6. Execution that is too slow. Sometimes a program is expected to terminate in a few seconds, but instead takes a couple of minutes. In extreme cases, poor programming can lead to programs do not complete in a reasonable amount of time (30 minutes?). Again this is probably faulty design, although there are algorithms that just take a long time to complete. This error can look like a lock-up to an impatient programmer. These too are easy to catch.

TECHNIQUES FOR CATCHING BUGS

The first and last word on catching bugs is information. You cannot catch bugs without it, and very often programmers try to get by on too little. The information you need is about the *program state*, i.e. the totality of the values of all the program's variables at each point in the execution, before and after every simple statement. The problem is that this amount of information can rapidly become overwhelming. Even in a fifty-line program with a handful of variables and a couple of loops, this information could cover several pages of output.

So, although more information is better than less, the goal is to collect only that information necessary for finding the bug. How is this done? The answer, as with testing, is divide and conquer - locating the bug in a region of the source code by verifying the program state at the top and bottom of the region, then narrowing down to a smaller and smaller region until the bug shows itself.

There are three main debugging techniques:

1. tracing with print statements,

2. the assertion statement, and

3. using a symbolic debugger.

We will look at the general principles of the first two approaches, but we only give the third a glance, as it is the most dependent on operating system and langauge environment. An example using the two first techniques willbe presented after the discussion of how they work.

TRACING WITH PRINT STATEMENTS.

This is the oldest and possibly best technique - if it is done right. This can essentially do the same job as the hand simulation. Values of variables at any point can be printed out by inserting a print statement for these values at the chosen point. For instance, if we want to find the values of the variable `total` each time round the following loop, we would place a print statement at the top of the loop. Before inserting the trace we might have:

See Source code Example #11

After the trace is inserted we have:

See Source code Example #12

If there is an error with the way total is initialized, or the way the totaling is done, then the error should reveal itself. Better yet would be to print the value of `num` as well, so that the totaling can be verified each time a different number is read in.

The more print statements you insert, the more information you have, but there are a few points to remember:

- Take the opportunity to print values, not just messages. Printing a message will verify that the program execution reached a particular point, but it is the values that are ultimately more important.
- Try to put them at the top and/or bottom of sections, so that the results of particular calculations or procedures can be verified. A section should be obvious from your design; in fact there should already be a comment in the source code to identify the different sections.
- When you have many subprograms, make part of the trace message the name of the subprogram. It is much easier then to read a long program trace, because it also shows the calling structure of the program (i.e. which subprogram calls which others).
- Too may print statements make the trace too hard to decipher. Keep the print statements to the minimum you really need. As debugging proceeds, you should take out old print statements that do not help any more.

THE ASSERTION STATEMENT.

A newer technique is related to the program design even more so than using print statements. The idea of the assertion is that if we understand our design well enough, then we should be able to predict what will happen when the program executes. We should be able to state what we expect to happen before and/or after each section or statement. In order to use this in debugging we proceed as follows:

Find a point in the source code where you are pretty sure what is going on and you can make an assertion of truth about the values that variables have at that point. For instance, we may have an array index that should always lie between 0 and 9 (for indexing a ten-element array).

Then insert the following assertion at that point:

See Source code Example #13

Run the program. If the value of `i` ever strays outside the range, a message will be printed to the effect that the assertion has failed, giving the line number where the assertion was inserted, and then the program will terminate safely, without crashing (note that the program stops at this point--no attempt is made to recover from the failure). If the assertion succeeds, because the value of `i` is always between 0 and 9, then nothing happens; it is as if the assertion was not there.

Thus, putting in assertions to a program that works fine will merely slow it down a little. The assertion, however, is just sleeping. It can wake up at any time and catch a bug! Assertions have the advantage of not printing out reams of useless information such as is common with trace statements. However, it only makes a test on values or ranges of values; it does not help with discovering why the program failed. Thus, a combination of tracing and assertions is needed to really track down bugs.

WHAT ARE ASSERTIONS?

In fact, assertions are simply conditional statements inserted into the source code. Even if the language system you are using does not have the standard assertion macro that C and C++ now have (you may find older C systems without this facility) it is easy enough to add code like this:

See Source code Example #14

Again, if the value of x is inside its range, the program executes just as if the test were not there, but will print the message if the value of x is wrong.

PROGRAMS THAT HELP DEBUGGING.

This is the most sophisticated of the techniques, and is used by professional programmers (yes, even professionals write programs that do not work!) A debugging program is essentially an interpreter that can monitor the program's execution at the statement level. There are several available on UNIX, but DOS has no such standard, easily available program apart from the primitive DEBUG program. More often, programmers using PCs will use an integrated development environment (IDE) that has a built-in debugger that is easy to use. The simplest UNIX debugger, dbx (or gdb) is hard to use because it so cryptic. The most complex ones, like CodeCenter, are hard to use because they are very complex. Most beginners do not need to use a debugger, but knowing they are there in the future when programs get more and more complex is nice to know. All debuggers provide the following facilities:

- The ability to insert a breakpoint in the program so that execution will pause when this point is reached.
- The ability to single-step the program form a breakpoint, one statement at a time.
- The ability to inspect the values of variables at any breakpoint, and even to change them.

A DEBUGGING EXAMPLE

This example is finding the number of occurrences of the pair of letters 'ie' and 'ei' in a piece of text:

See Source code Example #15

This program prints out incorrect numbers of IEs and EIs when a pair is followed by the same letter. In particular, the program prints three IEs for the word IEEE, instead of one, as it should. However, it gives correct numbers for isolated pairs, and also for text with no IE or EI pairs. These tests are as follows (input is underlined):

TEST 1

```
Type any number of lines followed by EOF
ie

the number of IEs is 1

and the number of EIs = 0
```

TEST 2

```
Type any number of lines followed by EOF
ei

the number of IEs is 0

and the number of EIs = 1
```

TEST 3

```
Type any number of lines followed by EOF
aieb

the number of IEs is 1

and the number of EIs = 0
```

TEST 4

```
Type any number of lines followed by EOF
aeib

the number of IEs is 0

and the number of EIs = 1
```

TEST 5

```
Type any number of lines followed
by end-of-file
what a relief!

the number of IEs is 1

and the number of EIs = 0
```

TEST 6

```
Type any number of lines followed
by end-of-file
many characters received

the number of IEs is 0

and the number of EIs = 1
```

TEST 7

```
Type any number of lines followed
by end-of-file
ie ie ie ei ei ei ie ie ie

the number of IEs is 6

and the number of EIs = 3
```

TEST 8

```
Type any number of lines followed
by end-of-file
nothing in here
at all!

the number of IEs is 0

and the number of EIs = 0
```

TEST 9

```
Type any number of lines followed by EOF
iiiieeee

the number of IEs is 4

and the number of EIs = 0
```

TEST 10

```
Type any number of lines followed by EOF
eeeeiiii

the number of IEs is 0

and the number of EIs = 4
```

THE TRACE METHOD

The program is simple enough (not always the case!) that we can print out the values of all variables inside the loop in an attempt to uncover the bug. The revised program with the print statement underlined, is:

See Source code Example #16

If we do this, and re-run the troublesome tests, we get these results:

```
Type any number of lines followed by end-of-
file
i
c = i lastc = i totalIEs = 0 totalEIs = 0
i
c = i lastc = i totalIEs = 0 totalEIs = 0
i
c = i lastc = i totalIEs = 0 totalEIs = 0
i
c = i lastc = i totalIEs = 0 totalEIs = 0
e
c = e lastc = i totalIEs = 1 totalEIs = 0
e
c = e lastc = i totalIEs = 2 totalEIs = 0
e
c = e lastc = i totalIEs = 3 totalEIs = 0
e
c = e lastc = i totalIEs = 4 totalEIs = 0

the number of IEs is 4

and the number of EIs = 0
```

Very often, as here, finding the bug from an execution trace such as this is a problem of looking for patterns. In this case, the pattern emerges after the first ‘e’ after the fourth ‘i’ has been input. The totals are correct after the first ‘e’ but totalIEs continues to increase for each subsequent ‘e’ that is input. The problem is with the variable lastc. It does indeed keep the last character read, but after ‘ie’ is found, it sticks at ‘i’ instead of changing to ‘e’. The bug has been found and can now be corrected. We will leave the corrected version until after the assertsion merhod has been shown.

THE ASSERTION METHOD

Adding an assertion to this program is easy. The purpose of the variable `lastc` is to record each character read so that it can be tested with the next one as part of a pair. Before reading another character, the values of `lastc` and `c` should be the same. Thus we add an assertion at the end of the loop: ...

See Source code Example #17

The next time the characters IE or EI are read, the assertion will fail. The program is not keeping

```
Type any number of lines followed by end-of-file
iiiieeee
Assertion failed: lastc == c, file d:\cs\iande.c, line 19
```

the last character when a pair has been found. The result is:

Notice, that the program does not terminate normally, but exits after printing the error message. Putting in a print statement completes the debugging by giving the same information as the trace method:

See Source code Example #18

At this point there is enough information to correct the error, by removing the last ‘else’ from inside the while loop:

See Source code Example #19