Mark as done

In this article, we will learn some basic debugging techniques for you to survive the first few weeks. Later after we have learned "pointer", we will learn how to use professional debuggers such as "DBG".

Debug by checking against expected input-output

Testing a program against a well-chosen set of input tests gives the programmer confidence that the program is correct. During the testing process, the programmer observes input-output relationships, that is, the output that the program produces for each input test case. If the program produces the expected output and obeys the specification for each test case, then the program is successfully tested.

But if the output for one of the test cases is not the one expected, then the program is incorrect -- it contains errors (or defects, or "bugs"). In such situations, testing only reveals the presence of errors but doesn't tell us what the errors are, or how the code needs to be fixed. In other words, testing reveals the effects of errors, not the cause of errors. The programmer must then go through a debugging process, to identify the causes and fix the errors.

Debug by checking against expected input-"intermediate outputs"

A program, if not as simple as a "Hello World", usually comprise of a number of lines of code. The lines usually separate into multiple logic blocks that the output of the previous block becomes the input of the next block. In each block, there are variables that store the input from the previous block, variables that store the intermediate results of the block and variables that store the output of the block.

Using the function `printf` (print the formatted string to the standard screen output in C/C++, `#include<stdio.h>`, http://www.cplusplus.com/reference/cstdio/printf/) or `cout` (standard output stream in C++, `#include<iostream>`, http://www.cplusplus.com/doc/tutorial/basic_io/), we can print the input-"intermediate output" to the screen for inspection to see starting from which block does the program started to behave incorrectly.

Going through each block, we'll be able to pinpoint which block is the culprit. Drilling into a block, we could print the variables that store the intermediate results to the screen for inspection. Within each block, loops might exist. Printing the variables in a large loop will flush the screen with nonsense outputs. To reduce outputs, we might print the variables every 1000 loops or a larger number, say for example,

```
for(int i = 0; i < 10000; ++i) {
  if(i % 1000 == 0) {
    printf(...);
  }
}
```

Bug Prevention and Defensive Programming

Surprisingly, the debugging process may take significantly more time than writing the code in the first place. A large amount (if not most) of the development of a piece of software goes into debugging and maintaining the code, rather than writing it.

Therefore, the best thing to do is to avoid the bug when you write the program in the first place! It is important to sit and think before you code: decide exactly what needs to be achieved, how you plan to accomplish that, design the high-level algorithm cleanly, convince yourself it is correct, decide what are the concrete data structures you plan to use, and what are the invariants you plan to maintain. All the effort spent in designing and thinking about the code before you write it will pay off later. The benefits are twofold. First, having a clean design will reduce the probability of defects in your program. Second, even if a bug shows up during testing, a clean design with clear invariants will make it much easier to track down and fix the bug.

It may be very tempting to write the program as fast as possible, leaving little or no time to think about it before. The

programmer will be happy to see the program done in a short amount. But it's likely he will get frustrated shortly afterward: without good thinking, the program will be complex and unclear, so maintenance and bug fixing will become an endless process.

Once the programmer starts coding, he should use **defensive programming**. This is similar to defensive driving, which means driving under worst-case scenarios (e.g, other drivers violating traffic laws, unexpected events or obstacles, etc.). Similarly, defensive programming means developing code such that it works correctly under the worst-case scenarios from its environment. For instance, when writing a function, one should assume worst-case inputs to that function, i.e., inputs that are too large, too small, or inputs that violate some property, condition, or invariant; the code should deal with these cases, even if the programmer doesn't expect them to happen under normal circumstances.

Remember, the goal is not to become an expert at fixing bugs, but rather to get better at writing robust, and mostly error-free programs in the first place. As a matter of attitude, programmers should not feel proud when they fix bugs, but rather embarrassed that their code had bugs. If there is a bug in the program, it is only because the programmer made mistakes.

Last modified: Friday, 16 February 2024, 3:19 PM

Back to Course