**Introduction to Debugging**

*Debugging* means locating (and then removing) *bugs*, i.e., faults, in programs. In the entire process of program development errors may occur at various stages and efforts to detect and remove them may also be made at various stages. However, the word debugging is usually in context of errors that manifest while running the program during testing or during actual use. The most common steps taken in debugging are to examine the flow of control during execution of the program, examine values of variables at different points in the program, examine the values of parameters passed to functions and values returned by the functions, examine the function call sequence, etc. In the absence of other mechanisms, one usually inserts statements in the program at various carefully chosen points, that prints values of significant variables or parameters, or some message that indicates the flow of control (or function call sequence). When such a modified version of the program is run, the information output by the extra statements gives clue to the errors.

**Debugging Techniques**

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 (or symptoms) of errors, not the cause of errors. The programmer must then go through a debugging process, to identify the causes and fix the errors.

**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 afterwards: 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, (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.

**Classes of Defects**

Even after careful thought and defensive programming, a program may still have defects. Generally speaking, there are several kinds of errors one may run into:

* **Syntax or type errors.**These are always caught by the compiler, and reported via error messages. Typically, an error message clearly indicates the cause of error; for instance, the line number, the incorrect piece of code, and an explanation. Such messages usually give enough information about where the problem is and what needs to be done. In addition, editors with syntax highlighting can give good indication about such errors even before compiling the program.
* **Typos and other simple errors** that have pass undetected by the type-checker or the other checks in the compiler. Once these are identified, they can easily be fixed. Here are a few examples: missing parentheses, for instance writing x + y \* z instead of (x + y) \* z; typos, for instance case t of ... | x::tl => contains(x,t); passing parameters in incorrect order;  or using the wrong element order in tuples.
* **Implementation errors.** It may be the case that logic in the high-level algorithm of a program is correct, but some low-level, concrete data structures are being manipulated incorrectly, breaking some internal representation invariants. For instance, a program that maintains a sorted list as the underlying data structure may break the sorting invariant. Building separate ADTs to model each data abstraction can help in such cases: it can separate the logic in the algorithm from the manipulation of concrete structures; in this way, the problem is being isolated in the ADT. Calls to repOK() can further point out what parts of the ADT cause the error.
* **Logical errors.**If the algorithm is logically flawed, the programmer must re-think the algorithm. Fixing such problems is more difficult, especially if the program fails on just a few corner cases. One has to closely examine the algorithm, and try to come up with an argument why the algorithm works. Trying to construct such an argument of correctness will probably reveal the problem. A clean design can help a lot figuring out and fixing such errors. In fact, in cases where the algorithm is too difficult to understand, it may be a good idea to redo the algorithm from scratch and aim for a cleaner formulation.

**Difficulties**

The debugging process usually consists of the following: examine the error symptoms, identify the cause, and finally fix the error. This process may be quite difficult and require a large amount of work, because of the following reasons:

* *The symptoms may not give clear indications about the cause.* In particular, the cause and the symptom may be remote, either in space (i.e., in the program code), or in time (i.e., during the execution of the program), or both. Defensive programming can help reduce the distance between the cause and the effect of an error.
* *Symptoms may be difficult to reproduce.* Replay is needed to better understand the problem. Being able to reproduce the same program execution is a standard obstacle in debugging concurrent programs. An error may show up only in one particular interleaving of statements from the parallel threads, and it may be almost impossible to reproduce that same, exact interleaving.
* *Errors may be correlated.* Therefore, symptoms may change during debugging, after fixing some of the errors. The new symptoms need to be re-examined. The good part is that the same error may have multiple symptoms; in that case, fixing the error will eliminate all of them.
* *Fixing an error may introduce new errors.*Statistics indicate that in many case fixing a bug introduces a new one! This is the result of trying to do quick hacks to fix the error, without understanding the overall design and the invariants that the program is supposed to maintain. Once again, a clean design and careful thinking can avoid many of these cases.

**Types of Debugging**

Following are types of debugging:

* Trace-based debugging
* Spectrum-based debugging
* Delta debugging

Different techniques are better suited to different cases. There is no precise best method. It all depends on the type of codebase you’re testing and your preferences.

Trace-Based Debugging

Trace-based debugging is traditional and the most common debugging technique used in most debugging tools today. Trace-based debugging is predicated on the concept of breakpoints.

A breakpoint is a pausing or stopping point added to the program to examine the state of program execution up until that point. After rectifying the current bug, the developer usually sets the next breakpoint and repeats the same process until all the bugs are corrected.

Traced-based debugging has four sub-techniques

1. Trace debugging (TD)
2. Omniscient debugging (OD)
3. Algorithmic debugging (AD)
4. Hybrid debugging (HD)

All of these sub-techniques are based around the same concept: program slicing.

Slicing

Program slicing is a technique used in computer science to simplify big programs by focusing on a smaller, selected section of them. The process of slicing deletes parts of the program that have no or little effect on the aspect being tested or the overall output of the program.

Program slicing is used in many applications in computer science:

1. Debugging.
2. Reverse-engineering a program.
3. Measuring a program’s coverage, overlap, or clustering.
4. To gain a deeper understanding of the inner workings of a program.

There are several possible approaches to slicing a program, such as static slicing, dynamic slicing, conditional slicing, and amorphous slicing.

* Static slicing:Static slices are constructed for a subset of the program’s variables without any assumptions about the input of the program. After picking a slice’s variable set, we can construct one of two types of slices: a backward slice or a forwardslice. A backwardslice contains the sections of the program that could affect the slice’s variables, while a forwardslice contains those sections that are affectedby the slice’s variables.
* Dynamic slicing: Dynamic slices are formed based on the variable set and the point in the code we want to inspect in addition to the sequence of input values for which the program produced an error. This information is called the dynamic slicing criterion.
* Conditional slicing: In static slicing, we provide nothingabout the input. In dynamic slicing, we construct it based on a specificinput. Conditional slicing lays somewhere in the middle. In a conditional slice, we can provide informationto the slicing tool about the input withoutbeing so specific about precise values.
* Amorphous slicing: All previous slicing discussed so far represents syntax preserving. These approaches are constructed by deleting the statements that don’t affect the set of variables in question. However, amorphous slices are constructed using any program transformation, which simplifies the program while preserving the functionality of the overall program.

Among those four types, dynamic slicing is the most used technique in debugging. That is because when we debug, we have a specific test case that produced the error, so we can use that test case as an input to construct our slices.

Trace Debugging

In trace debugging (TD), the debugger can set a breakpoint, and when a breakpoint is reached during program execution, the program is suspended. Then, the programmer can examine the program state by running it line by line. The debugger takes control over the interpreter only, and hence the scalability of TD is the same as that of the interpreter. During the process of finding the location of a bug, if a bug is found before the breakpoint, then the debugging process needs to restart to catch it.

Omniscient debugging

In omniscient debugging (OD), also known as back-in-time debugging, the debugger can trace the computations of the program both backward and forward. In this type of debugging, execution traces are enormous, and storing them is a challenging task, so the scalability of OD is quite tricky.

Algorithmic debugging

Algorithmic debugging (AD) is a semi-automatic debugging technique that produces a dialog between the debugger and the developer to find bugs. So, there is no need to see the code to perform the actual debugging. That’s why the level of abstraction of AD is high. AD is done in two phases: First, it builds the execution tree of the given program. Then, the execution tree made is explored in full detail.

Hybrid debugging

Hybrid debugging (HD) is a debugging technique that combines trace debugging, omniscient debugging, and algorithmic debugging. First, it uses trace debugging to find out the part of the program where the error occurs. Then, it uses algorithmic debugging on only that part of the program and produces its execution tree. Afterward, it applies omniscient debugging to a single method where algorithmic debugging identified a bug.

Spectrum-Based Debugging

In spectrum-based debugging, also known as spectrum-based fault localization (SFL), the debugging process is done by monitoring the statements included in a particular execution tree. This is achieved by using the program spectrum to identify the active part of the program during its run. The program spectrum isa collection of runtime informationthat gives a view of the dynamic behavior of the program. It includes some flags corresponding to different parts of the program. Different types of program spectra exist, such as block hit/miss and function hit/miss. These spectra are used to pinpoint the exact sections of the code running for specific or abstract input.

Delta Debugging

The process of delta debugging (DD) is to minimize automated test cases. It takes test cases that may cause the error and prepares an error report. From that error report,minimal test cases are selected based on their high probability of producing the error. The minimum test cases will regenerate the same error and thus help the developer locate the bug behind it.

Online Notes

<https://cds.cern.ch/record/1100526/files/p71.pdf>

<https://www.itu.dk/people/slauesen/Papers/DebuggingTechniques.pdf>

<http://umu.diva-portal.org/smash/get/diva2:652873/FULLTEXT01.pdf>

<https://cds.cern.ch/record/1100526/files/p71.pdf>

<https://www.itu.dk/people/slauesen/Papers/DebuggingTechniques.pdf>

<http://umu.diva-portal.org/smash/get/diva2:652873/FULLTEXT01.pdf>

Video Links

<https://www.youtube.com/watch?v=waO_UAH6aqo>

<https://www.youtube.com/watch?v=h3CUDL6Qsc4>

<https://www.youtube.com/watch?v=Kmx_NL4_2Fk>

<https://www.youtube.com/watch?v=waO_UAH6aqo>

<https://www.youtube.com/watch?v=h3CUDL6Qsc4>

<https://www.youtube.com/watch?v=Kmx_NL4_2Fk>