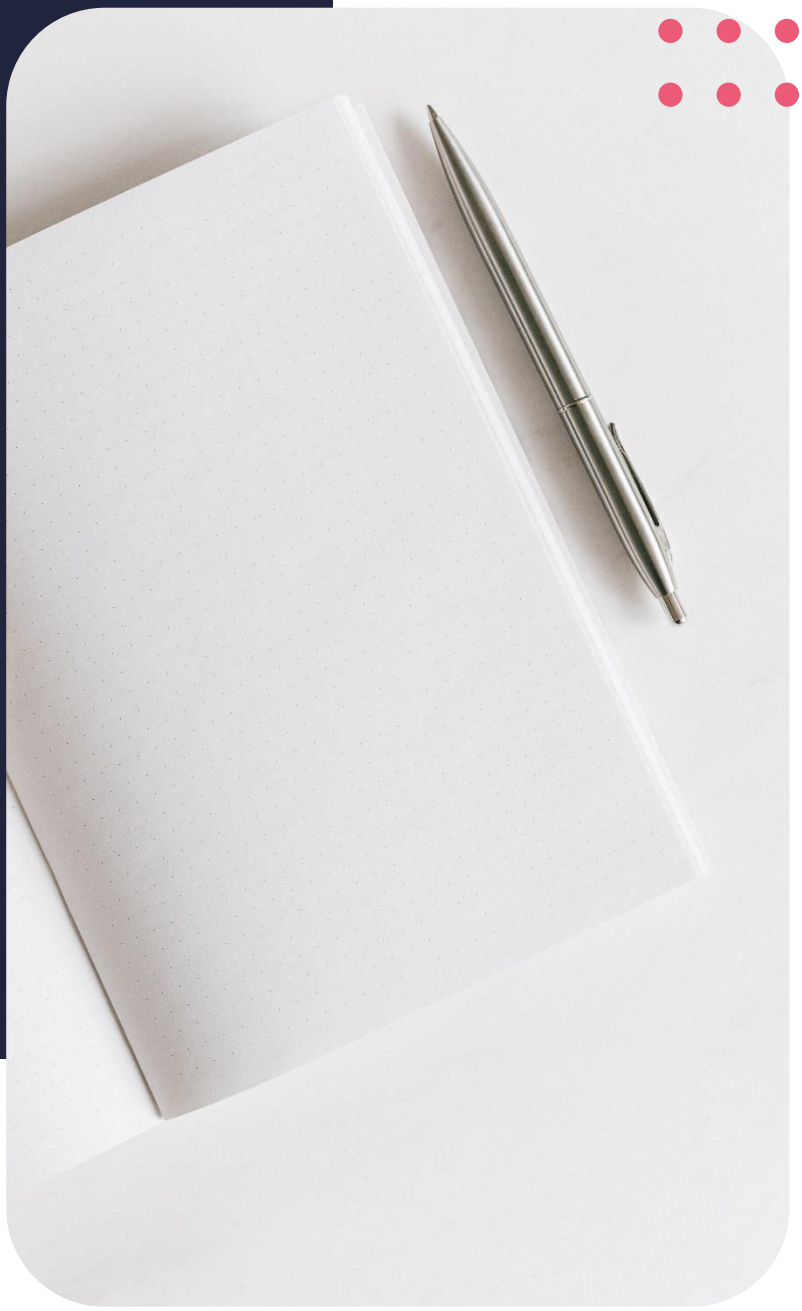
Diploma in Computer Science

Debugging

Contents



Lesson objectives

By the end of this lesson, you should be able to:

- Identify bugs and how they affect code
- Explain the most common debugging resources
- Appreciate strategies used for debugging

Introduction

Welcome to our fifth lesson in our series. In this lesson, we look at an important step in the development of software, namely debugging. This is a major step in software development, which ensures that everything works as expected. As expected, errors creep into your code from time to time, and these have varying effects on the product. It falls on the developer to make sure that things work exactly as the customer asked. Testing is a process that uses quite a lot of strategies, and in this lesson, we will look as some of the most popular ones

Bugs in the computer world

Let's face it, human beings are really bad at a lot of things. It's very hard for a human being to set out to do a task and complete it error free. This also applies to writing code , which is barely ever a perfect process. Software nearly always has bugs. You probably already know that we are not talking about bugs fly around in your room while you're trying to sleep, but with that said, did you know that the first computer bug was an actual bug!

What is a computer bug?

A computer bug is an error, flaw, or fault that causes the program to behave unexpectedly or produced an unexpected result.

It is essentially the behaviour that is observed when a program does not perform as intended. This applies to both computer hardware and software.

Bugs are typically the result of errors made by developers during the development process.

Errors can also occur when a program is translated from one language to another; what works in one language may not necessarily work in another.

Bugs will literally bug you , just like a mosquito would, until you get rid of them! A program will produce unexpected results or behave unexpectedly until the bugs are removed from it.

The first bug...

The term bug has its origins from the 1800s, where it became popular in engineering circles to term things that were not working correctly as bugs.

If you remember from module 1, early computers used vacuum tubes and a lot of mechanical parts.

The very first instance of a computer bug was recorded at 15:45 on the 9th of September 1947. This bug was an actual moth, which had a 5cm wingspan, was extracted from the number 70 relay, Panel F, of the Harvard Mark II Aiken Relay Calculator.

The bug was carefully removed and taped to the logbook, and the term computer bug was used to describe the incident. 'The first actual case of a bug being found' are the exact words that were used in the logbook.

This logbook, complete with attached moth, is part of the collection of the Smithsonian National Museum of American History.

The Harvard Mark II operators did not coin the term 'bug', but it seems as though the incident contributed to the widespread use and acceptance of the term within computer software circles and vocabulary.

Bugs vs baddies

That said, there's a common misconception that we need to address before we move further. Finding out that your computer has a virus can be a frustrating experience, and often causes your computer to behave in unexpected ways, stealing your passwords and wiping your bank account clean. It can make your computer's fans sound like a jet that's about to take off while the CPU and GPU are busy mining bitcoin for someone else at your expense! It is, however, technically not a bug, because the malicious developer actually meant for your computer to behave like that! A bug, on the other hand, is purely unintentional. It's an error on the part of the programmer.

Bugs can cost a fortune – Ariane 5

Bugs can turn out to be very expensive mistakes. Here are some notable bugs that cost a fortune. Ariane 5 Rocket Failure (1996):

- The Ariane 5 explosion was one of the most expensive software mistakes in history. It was a famous European rocket that was used to launch satellites, and the project cost around 8 billion. The rocket exploded a mere 40 seconds after launch. The rocket crashed because of an integer overflow error, which is a pretty common bug in computer programming. It occurs when the programmer overlooks the amount of memory that is required, which is exactly what happened during the development of the project. The investigation revealed that the code tried to fit a 64-bit number into a 16-bit space. It was a pretty expensive bug, which caused a loss of around $370 million. That is quite a lot of money!

Bugs can cost a fortune – Y2K

Another example is a pretty recent one from 2000. In the 1960s, it was standard practice to represent dates, for example, 1960, as only '60' and leave out the '19' back then, memory was scarce, and many programmers a before the variable "year" was an unnecessary waste of memory. It was all roses till December 31, 1999, then reality hit. Once January 1, 2000 came, many computers read the year as 1900 because the 2 digits represented 19 instead of 20. If you're kind of old (pun intended NINA: you can say this jokingly instead of saying the words), you probably remember the bug that was called Y2K or the millennium bug. It affected banks that calculate interest rates on a daily basis, centres of technology such as power plants, transportation, and a lot of other things. It was a spectacular show, which ran into billions of dollars worldwide as various entities around the world upgraded their systems.

Bugs can cost a fortune – NASA

Yet another example is NASA's Mars Climate Orbiter from 1988: Due to the different units of measurement, met its unfortunate demise after travelling towards Mars for 286 days. The project cost about $125 million with the intention of studying the Martian surface and climate. The orbiter lost communication with the earth station and entered the Mars atmosphere at a wrong angle, yet it was not supposed rather than entering the planet's orbit. The software that controlled the Orbiter's thrusters used imperial units instead of metric units.

Types of bugs

Bugs are typically classified by nature, priority, and severity. This constitutes the several types of software defects that are present in software in one form or the other. Let's look at each of these closely.

Software defects classified by nature

We will first look at classification by nature.

Functional defects: These are errors that days when the software behaves in a way that is not compliant with functional requirements. This type of defect is usually discovered during functional testing. An example of this is of a system that should only accept integers but its phones to accept floating point numbers as well.

Performance defects: When the software does not perform at inacceptable speed and stability, it is said to have performance defects. This is what is tested for during performance testing. You remember from previous lessons that any software

package should be able to use as little resources as possible while having the maximum possible throughput, which equates to efficiency. The software package should also be as stable as possible in the target environment.

Usability defects: These occur when a software package is difficult to use or navigate or is unnecessarily complex. Think of a sign-up process that has seven stages instead of just two. These are validated against usability requirements during usability testing.

Compatibility defects: A software package should show consistent performance and behaviour across all target systems. An example of this is a website that doesn't show all elements on a mobile device but works just fine on a desktop display. Guidelines for websites typically state that the experience should always be uniform regardless of the end user's platform. This is also expected across versions of operating systems. A software package is deemed defective if it works on Android 9 but starts to exhibit different behaviour on Android 11. Defects can incorporate things such as what size and type, placement of user elements and response of buttons, right up to back-end functions.

Security defects: A software package might work correctly but come with security defects. If you remember from our previous module, it is always good programming practice to write code in such a way that the attack surface is kept at a minimum. This includes keeping the scope of variables in check and making sure that elements that are not being used are removed from memory. The software package is tested against known threats and vulnerabilities such as weak authentication, buffer overflows and encryption errors.

Software defects classified by severity

Like we mentioned before, software defects are classified by severity. There are typically four levels of severity

Critical defects: are the highest level and should be addressed immediately, since testing cannot proceed without fixing these. An example of this is an application that does not store input that is required by the next procedure. There is no point testing this package as there's simply no data to work with beyond the point at which the error has been detected.

High severity defects: are a level below critical defects. These affect the key functionality of an application. The software package may work in some way, for example, a system that should allow two people to log in at the same time only allows one person to log in and use all the functionality of the software package but does not allow the second person to go past login while the first person is still using this software package.

Medium severity defects: refer to minor defects. A minor function exhibits behaviour that is contrary to what is specified in the requirements, such as about page that doesn't work. The information contained on this page is still important back does not affect the core functionality of the program.

Low severity defects are the lowest rank of defects. These are typically minor defects that the production team can get away with and release the software anyway, such as text that is not properly aligned.

Software classification by priority

The last aspect that we will look at is bug classification by priority.

This has a direct link with the effect that the bug has on the end user, both in terms of productivity and throughput. It is also directly linked to the severity ranking of a bug.

Urgent defects: need to be fixed as soon as they are reported. It is not limited to severe bugs only, as low severity defects can be added to the list as well. The project manager, product owner or stakeholder can identify the priority of a bug.

High priority defects: These are slotted to be fixed in an upcoming release, since the user can make do with the system as it is and wait for a fixed release. These are typically in the software package's exit criteria in the said release.

Medium-priority defects: are the errors that may be fixed after an upcoming release or in the subsequent release. The development team can forgo these changes in the current release and take care of them in the next release.

Allocating the correct prioritisation and severity levels speeds up defect fixes and increases the overall capacity and efficiency of the testing system. This in turn streamlines the whole testing process. This is crucial for the milestones and deadlines of the project.

Avoiding bugs

CONTENT must be introducing two common ways to avoid bugs:

- Keep code simple! and
- Use well-tested libraries

Keep code simple

Code simplification: It is good programming practice to make sure that your code does not try to do too much. The best way of avoiding bugs is to avoid code that has internal structure. In simpler language, a function should do just one thing. This makes it easier to trace errors in a block. This goes hand in hand with modularity. Code is contained in functions and macros. The functions and macros are each responsible for a single component of the program. The functions are interconnected using shared variables and parameters. As a programmer, you should make sure that the functions depend on each other in clearly defined ways. Putting code in 'containers' makes it a lot easier to isolate any problems in the code. You can isolate a module and test it independently.

Use well-tested libraries

Another way to avoid bugs is to use existing libraries. We learnt in previous lessons that C and other programming languages allow you to create your own libraries. This comes in handy when you want your code to be portable and yet still all of those clever little ways of solving problems that you created in your code, but that could come with problems. If there is a well-tested library available that will carry out the same function you were planning to code, it is generally better to use that code instead of writing your own code.

Debugging

Debugging applies to both hardware and software. It is a multi-stepped process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or looking for a way to navigate around it if it proves too difficult to remove or it simply can't be removed at all. Debugging is part of the software testing process and is an integral part of the entire software development lifecycle. The final step of debugging is to test the correction or workaround and make sure it works. Let's now take a closer look at this process.

Debugging is an important part of determining why an operating system, application or program is misbehaving. Bugs are pretty commonplace, even if you take due care and use the same coding standard as other team members, bugs will still pop up in your new software package. In many cases, The process of debugging a software package can take more time than it initially took to write the software package! Bugs typically start to pop up in pre-deployment testing and continue right well into the maintenance stage.

Steps involved in debugging

1. Identify the error: This is naturally the first step in addressing a bug, or any problem for that matter. A badly-identified error can lead to wasted development time, which in turn leads to missed deadlines and a generally sloppy product. Production errors are typically reported by users and are often hard to interpret and sometimes the information that the users provide is vague and doesn't really pinpoint the problem. It is important to identify the actual error in order to effectively solve the problem.

2. Locate the error: After positively identifying the error correctly, it needs to be located in the code. No fancy bells and whistles here, the focus is purely on locating the error. After all, there really is no point trying to fix an error when you don't know where it is coming from.

3. Analyse the code: In the third step, the code is analysed so as to understand what exactly it is doing against what it is supposed to do. This helps the developers understand the error. There are two main goals in the analysis stage; checking if the error does not cause a ripple effect, essentially checking if a bug does not affect any other parts of the program. The second part assesses the risks of incurring any collateral damage in fixing the bug.

4. Prove the analysis: this stage looks at the bug and all the problems that it is solving, and documents it. This information is used for writing automated tests for these areas with the help of a test framework. This means that these bugs can be tested for automatically in future builds of the software.

5. Build the fix: the actual code for the fix is then written and appended to the existing code.

6. Test and validate: the entire program is then tested again to see if it now performs as expected. Validation seeks to determine whether added code does not cause new problems. You probably don't want to hear this, but it is quite common that new errors sprout in other places after you fix a known error! The errors might even go unnoticed, only to be discovered by the user after deployment!

Basic steps for debugging

Before we attempt to use any tools for debugging, it's important to know the FOUR basic steps. These steps are extremely general and apply to pretty much everything that needs to be built. They are:

1. Know what your program is supposed to do.
2. Detect when it doesn't.
3. Fix it.
4. Repeat until everything works

It is really just as simple as that! No fancy words and complicated processes. This is the basis of all other debugging methods.

Don't make these mistakes...

A common misconception is that step one is not that important, and we should focus on making whatever code we have work in whatever way. Many people just go into the code and randomly change bits of the code until it works. While you can certainly get away with this with small programmes that just span a few dozen lines of code, this quickly becomes extremely inefficient and untenable when you are working with large blocks of code. If you do not understand one function of a program, it is most likely that you do not understand the rest of the program, and you will spend hours and hours of meaningless clicks that will not get you anywhere.

Another common 'trick' that people try is to debug the code by just looking at the output. Again, if you do not understand what the program is doing to get to that result in the first place, then you likely are not going to get anywhere just by looking at the output. You are literally just guessing what the computer did instead of going into the code and following along the path that the computer followed to get to that result.

Remember these basic guidelines before you start using any fancy debugging tools.

Debugging tools

Add asserts to your code

As soon as your code starts getting a bit complex, you should make it standard practise to add asserts into your code. You recall the assert header file from our lesson on header files, which gives you access to the assert macro, which in turn allows you to test if a condition is true and halts your program if it isn't. This code snippet shows how you can use the assert macro:

```
 #include <assert.h>


Int
Main(int argc, char **argv)
{
    Assert(1+1 == 10);
    Return 0;
}
```

This pretty simple assert test if 1 + 1 is equal to 10, which is obviously false, so the program fails get that line and returns that error. Try running it's on your own pcs and see. This makes it very easy to pinpoint exactly where the error occurred.

Use the gdb debugger

If you're using a Linux system, then you're in luck . You can use the gdb debugger to basically follow along as your program runs. Using the GDB debugger, you can see exactly what the program did every step of the way. There is some good news for Windows users as well; some versions of Windows compilers coming with the GDP debugging tool

Use valgrind program with pointers and dynamic storage allocation

When you’re working with pointers and dynamic storage allocation, the valgrind program can come in handy. It essentially produces a report of any allocations and the allocations that occur in your program. The programme can be used with the argument -q so that it supports all other output that is not necessary for debugging and only display errors. it can also be used for other things such as checking for memory leaks, which, if you remember from our previous lessons, can potentially cause security problems after deployment.

Insert breakpoints

The places in which all these tools stop when they encounter problems are called breakpoints. A breakpoint is appointing a programme where execution is halted intentionally, either by pausing or terminating the program if an unexpected result is encountered. the tools that we have just discussed employ breakpoints by testing for certain conditions at certain points in the program. Breakpoints can be set in a lot of variable ways that address specific needs, both for the programmer, and in compliance with the compiler.

It is worth noting here that the methods that we have just discussed are not the only tools that are available out there. There are many, many, many debugging tools, even within see programming itself. Each programming language and IDE typically has its own tools that address its syntax and constraints.

Debugging strategies

In a large program that has thousands and thousands of lines of code, the debugging process can be made easier by using strategies such as unit tests, code reviews and pair programming. Testing strategies allow programmers to deal with bugs a lot faster than they would if they just read through the entire code blindly. We have already sort of discussed testing strategies when we looked at the methods that are used to classify bugs. Let's look at some of the most common ones.

Automated testing

Automated testing is used to test for routine bugs that are typically found in software. This is done by another software that goes through the software that has just been built and tests it for expected outcomes. If the software does not produce the desired outcome, the function or module in question is then highlighted an inspected further. It makes it easier to without obvious bugs and leave the programmers with more time to deal with more complex bugs. Automated testing is widely used in the software industry and is now used in a wide range of projects. Testing is carried out using a testing framework that can test things such as the graphic user interface using recorded keystrokes and mouse clicks that generate known output. This makes routine testing take far less time than it would if it were done by actual humans. API driven testing validates the program's behaviour using a variety of arguments and checking if the results are correct. This is a key feature of agile development methods as we saw in our previous lesson.

Change development strategy

Another method of eliminating bugs is to change the development strategy of the software. The incremental and bottom of program development allows programmers to isolate error since the program is developed incrementally and test it as often as possible after adding each piece of the project. This is the direct effect of reducing the number of errors that the programmers have to work on at any given time, since a unit is tested before another piece is added to it.

Logging

Another method is logging, which refers to the insertion of a number of printf statements that write to a file called a log. You may have seen such files if you have installed a software package before. At each critical step in the program, a line of text acknowledging the step that has just been executed is appended to the file. This is extremely useful for the end user to pinpoint errors in the deployment environment, especially if the error is due to a conflicting software package that can easily be removed without re compiling the program. If you're a member of the Android beta programme, you'll see that people are often asked for logs if they encounter problems with the software. This allowed the programmers to see exactly what went wrong during execution and the development environment, and therefore it takes less time for them to locate and deal with the bug.

Cluster errors

If there is a large number of errors that are reported, it is easy to cluster them, grouping them as similar bugs, then examining only one bug from each class. The reasoning here is that bugs from the same class being caused by one root problem, so addressing the root problem will probably fix the bug. While it doesn't always work, it sometimes saves developers a lot of time by helping them localise problems.

Debugging output

This is usually not recommended as it takes much more time to look at the output then just to go back and look at the source code. In my experience, it is only really useful if you understand the source code that you wrote and know how it got to the output that you're looking at. If you prefer to look at the output, it probably means that the code that you're running away from is not as readable as you think it is. That said, if your output has certain traits or patterns, especially if it is repetitive, or has a lot of dependencies that are Daisy chained, you can quickly pick up where the error occurred, and this makes it easier to go back to the source code, since you will know exactly where in the source code you need to go and fix

Change your attitude

Debugging can be thought of as an art. You might spend hours and hours looking for bugs and end up extremely frustrated after not finding them. Here are some pointers on debugging that may prove useful to you.

A number of other strategies can be viewed as a matter of attitude about where to expect the errors.

- The bug may not be where you expect it. If a large amount of time has been spent unsuccessfully inspecting a particular piece of code, the error may not be there. Keep an open mind and start questioning the other parts of the

program.

- Ask yourself where the bug is not. Sometimes, looking at the problem upside-down gives a different perspective. Often, trying to prove that the absence of a bug in a certain place actually reveals the bug in that place.

- Explain to yourself or to somebody else why you believe there is no bug. Trying to articulate the problem can lead to the discovery of the bug.

- Inspect input data, test harness. The test case or the test harness itself may be broken. One has to check these carefully, and make sure that the bug is in the actual program.

- Make sure you have the right source code. One must ensure that the source code being debugged corresponds to the actual program being run, and that the correct libraries are linked. This usually requires a few simple checks; using make files and make programs (e.g., the compilation manager and .cm files) can reduce this to just typing a single command.

- Take a break. If too much time is spent on a bug, the programmer becomes tired, and debugging may become counterproductive. Take a break, clear your mind; after some rest, try to think about the problem from a different perspective.

Demo

Today we're going to look at a short program and experiment with one of the debugging techniques that we just learned about. You'll see how you can pinpoint a bug and deal with it without fumbling through the code and wondering what's going on. It's all about making you coding smart!

Conclusion

On that note, we wrap up our fifth lesson. In this lesson, we looked at what bugs are and the implications that they have on our code. We also looked at the various techniques that are used to identify bugs and remove them. We had a brief look at the tools that are included in most ides for use by the programmer to identify bugs. And briefly looked at how you can use these tools to your advantage. We also looked at the common things that you need to look out for when writing code in order to avoid bugs in the first place. That's all for now, and remember, practice makes perfect. See you next time!

References

Colostate.edu. (2017). The Debugging Process. [online] Available at: https://www.cs.colostate.edu/~cs165/.Fall17/recitations/W11L2/doc/debugging.html

Debugging Strategies. (2012). [online] . Available at: https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1134/handouts/250%20Debugging%20Strategies.pdf.

GeeksforGeeks. (2019). 10 Famous Bugs in The Computer Science World - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/10-famous-bugs-in-the-computer-science-world/

HyperionDev (2019). HyperionDev Blog. [online] HyperionDev Blog. Available at: https://blog.hyperiondev.com/index.php/2019/01/24/good-ways-avoid-bugs-programming/

Nataliia Vasylyna (2011). Types of Bugs in Software Testing. [online] QATestLab Blog. Available at: https://blog.qatestlab.com/2011/03/24/3-types-of-bugs-in-software/

Nataliia Vasylyna (2011). Types of Bugs in Software Testing. [online] QATestLab Blog. Available at: https://blog.qatestlab.com/2011/03/24/3-types-of-bugs-in-software/

Romero, P., du Boulay, B., Cox, R., Lutz, R. and Bryant, S. (2007). Debugging strategies and tactics in a multi-representation software environment. International Journal of Human-Computer Studies, [online] 65(12), pp.992–1009. Available at: https://www.sciencedirect.com/science/article/abs/pii/S1071581907001000

Sidorova, T. (2020). Software Testing Basics: Types of Bugs and Why They Matter. [online] Scnsoft.com. Available at: https://www.scnsoft.com/blog/types-of-bugs