RTDM: Read the Debugging Manual

Table of Contents

0 Introduction and Summary of Sections

We all make mistakes when we code and unfortunately, keeping bugs out of our system is a never-ending journey for every programmer. Nevertheless, we shouldn't be afraid of it. Instead, lets fight it head on, with a clear mindset, an organised plan, and a bunch of strategies that up our sleeves. This manual is about how to fix bugs in a way that allows you to learn and grow as a programmer. It involves several strategies that you can implement whenever you code in order to make it easier for you to understand your program, understand what you are trying to solve, and fix it in an efficient and productive way.

This document is divided into six chapters and an appendix. There are no prerequisites or prior knowledge needed to understand this manual. The content of the manual is organised as follows:

Section 1: Reducing Bugs - Reducing the probability of bugs in your program is one of the greatest things you can do for your future self as it makes the locating/fixing bugs process a lot more effortless. This section introduces two methods to reducing bugs: pair programming and program planning.

Section 2: Emotional Awareness – Often we feel frustrated after trying to debug for 3 hours without any progress. But by checking in with how we are feeling, we can feel more motivated with how we deal with bugs. Methods such as peer discussions and using creative outlets are discussed in this section.

Section 3: Reading the Compiler - Using the compiler messages to your advantage is a great first step in getting rid of bugs. This section teaches you to change that mindset and understand that compiler messages are actually here to help you.

Section 4: Code Tracing - Finding the source of a bug can be a particularly troublesome task. By properly reading your code and understanding your variables, it's possible to find errors much more quickly. This section teaches you how to 'trace' your code and find the error quickly.

Section 5: Print Statements - While print statements are one of the most used debugging tools, it is not only possible but incredibly common to use them poorly. This section gives advice to make the most of this powerful debugging technique.

Section 6: Testing - Testing is a vital part of creating a functional program. The ability to create good test cases not only helps you see if your bug fixes worked, but may help you find new bugs too! This section will show you how to create a good test case and how to test effectively.

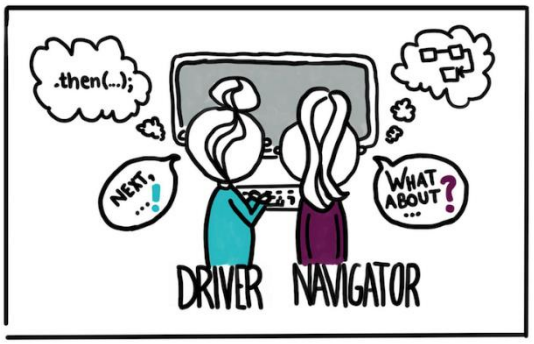
1 Reducing Bugs

Surprisingly, the debugging process takes significantly more time than the code writing process itself. This is because a lot of effort is needed to fix the bugs and maintain the program overtime. Therefore, the greatest thing to do is to reduce the probability of bugs from arising in the first place. This section introduces 2 ways you can achieve this: pair programming and program planning.

1.1 Pair Programming

Pair programming, as the name suggests, is when you program in pairs with another person. So, instead of one person writing all the code, two people work together on one machine. One person, who is the 'driver', writes the actual code, typically thinking aloud while doing so. The other person, the 'navigator', focuses on the bigger picture by checking the written program, and making suggestions for improving and progressing the code.

1. Start with a small, well-defined task and assign the driver and navigator between you and your partner.
2. Communicate the roles and split the workload so everyone knows what each person is doing.
3. Experiment with different combinations of work modes that suit both you and your partner. Some might prefer being the driver or navigator, or others might prefer to switch between different roles depending on the task.



https://martinfowler.com/articles/on-pair-programming.html

1.2 Program Planning

When you start working on a program, it is beneficial to think through the problem to understand what needs to be achieved, and how to go about accomplishing that by designing a clear plan. Although it is tempting to jump straight into coding, having a clean, straight-forward design not only reduces the probability of bugs, but makes it easier to track down and fix defects if they do show up later in the process.

1. Start by understanding the problem description by writing down what you already know and what you need to achieve. Clear up any questions or assumptions that you may have.
2. Using diagrams or pseudo-code (a simple way of writing code using plain English), come up with potential solutions.
3. Once you have decided on a solution, logically plan the steps that you need to get to your solution using diagrams, flow charts, or pseudo-code. This allows you to enter the coding stage with confidence and a strong idea of what you need to do.

Examples of ways to plan a program are shown in the Appendix.

2 Emotional Awareness

Emotional awareness is the process of recognising emotions. In the case of debugging, feelings of frustration and stress are common when you are locating or fixing a bug, especially when you go through an entire code line by line only to realise that it was a little semicolon that was preventing your program from executing. Having techniques to transform these emotions into more positive ones can help you calmly tackle your debugging process and successfully fix bugs. This section introduces 2 ways to become aware of your emotions: Peer discussions and creative outlets.

2.1 Peer Discussions

Having someone to talk to when you are frustrated is always a good way to relieve tension in your body and clear your mind, and the same thing applies when you are debugging. Engaging in discussions with peers about how and why you are feeling stressed about a piece of code is an effective way to vent out all the frustrated energy without causing more errors in your code. Simply by having conversations in which you can openly express the problems can allow beneficial interpersonal interactions with peers. It can also allow you to observe and relate to other people's programming performances, and maybe even share your ideas and give constructive feedback on how to improve each other's code.

2.2 Creative Outlets

If talking to others seems scary or nerve-wrecking to you, fear not. There are plenty of different ways you can still vent out your debugging frustrations on your own if you prefer to do it individually or silently. One of them is through creative outlets. Similar to voicing your debugging frustrations to peers, expressing them through artistic channels is something that might seem unusual at first, but is equally just as helpful in relieving tension and making you aware that these disturbing emotions are emerging. Some creative ways to vent out your struggles include:

1. Journaling your debugging experiences
2. Create artworks, like painting, drawing, doodling
3. Simply taking a break

There really is no 'proper' way to write in a debugging diary or draw. It is all up to you on how you want to express your frustrations. The key thing is to find an outlet that works for you in order to remove any clouded chaos from your mind. That way you can come back to your code all fresh and motivated, which believe it or not, does make it easier in tracking down and fixing a bug.

3 Reading the Compiler

Getting compiler error messages is not always fun, especially when you compile a piece of code thinking it is all well and good, and your terminal suddenly blasts dozens of error messages in front of you! As a result of this fear, many students actually end up ignoring the compiler altogether, which most of the time, really is not the best thing to do. But, no matter how you feel about the compiler, one thing is for certain: These compiler error messages are not here to intimidate you. Rather, they are here to help you fix any errors that is preventing your code from compiling.

The best things that you can do when dealing with compiler logs are:

1. Read the messages: it may seem discouraging and confusing when you read them messages, but reading the messages (even if you do not understand them) is the biggest step you can possibly take in getting closer to fixing your bugs.
2. If you have multiple compiler messages, scroll up to the first one and solve them top-down: Fixing bugs at the top of your terminal can help remove some of the later ones.
3. Fix warnings! Many students tend to ignore warnings that show up in the terminal. Our advice: treat warnings as errors and solve them the same way you would solve a bug. Fixing warnings is beneficial and a helpful way in becoming a well-rounded programmer.
4. Check the line numbers where the errors and warnings occur: This greatly fast-forwards the process of trying to locate bugs.

To make the last step easier for you, a table of the most common compiler error messages that you may come across when programming is shown in the Appendix. This table shows the compiler error messages on the left that you would normally see in a terminal, and an interpreted meaning is displayed on the right for you to easily understand what the messages mean.

If a compiler message is not in the table, then copy and paste the message into Google to find out what they mean. But venture carefully as some sites may be too advanced for you or have a complicated explanation.

4 Code Tracing

Tracing code is an effective way to both isolate an error and to improve your understanding of your code. In fact, this process is the main idea behind some debuggers, tools that can help make debugging easier. In those tools, this is referred to as 'stepping'.

Tracing works by stepping through your code as a computer would so you can see where the issues may occur. Use a piece of paper to track what values variables have at each line of code so that you can compare them to what their values should be at that moment. Doing it without paper is slow and likely to result in mistakes or losing track of values - if you don't have paper, try having Excel or Notepad open on the side, or combining this with Section 5 (print statements)!

Another version of tracing is backwards tracing, where you start at your output and read through each line in reverse order. This is good for finding mistakes quickly, but you should fully understand your code when you implement this method – otherwise, you can get lost and make mistakes yourself.

To test this out, let's find the bug in this piece of code- why does it output 4 instead of 10?

```
int addArray(int * array){
    int total = 0;
    for (int i=0; i< array.size(); i++) {
        total = array[i];
}
return total;
}
int main(){
    int array[4] = {1,2,3,4};
    int total = addArray(array);
    cout << total << endl;
}
```

By tracing our code, we can see that whenever we call the for-loop, total is reset to array[i] instead of adding array[i] - the line should read: total += array[i]; . It's an easy mistake, but hard to spot if there's lots of code to search, especially since your compiler likely won't warn you about this!

Make sure to read your code in the order your computer does – while not as necessary in a short piece of code like the above example, if many objects and functions are used, it's easy to get confused about the order of the function calls and lose track of where the error could be. Reading it as your computer does helps isolate the error and improves the speed at which you debug.

So, be sure to trace and read your code carefully; otherwise, you may miss the error you're looking for!

5 Print Statements

Print statements are a key component of debugging, and likely make up a lot of the debugging you currently do. They're used by novices and professionals alike, due to their simplicity and usability. Simply write `std::cout <<'Something Important'<< std::endl` and you're ready!

As some general rules:

1. Put values in your Print statements. Just printing a statement only tells you that a section of code is being run - but that doesn't tell you why your code might not be running. Is it because you didn't meet the requirements for the while() loop, or the if() statement? Since printing values always gives more information than just printing a set statement, it's almost always best practice to print a value.
2. Ensure that you don't print the exact same string in two areas. Putting the same output into both parts of an if() else() statement will give you no information about whether the if() or else() is running.
3. Combine print statements with other debugging strategies, such as code tracing. This ensures you get values directly from the code, preventing the chance of misreading code, and you'll be able to quickly find where a variable isn't what it should be.
4. Use Assert() statements. These are similar to print statements as they allow you to check a value at a given time, but they only print when an error is occurring. For example:

```
int addArray(int * array){
   int total = 0;
   for (int i=0; i< array.size(); i++) {
      assert( i>=0 && i<array.size() );
      total = array[i];
   }
   assert( total == 10);
   return total;
}
```

This code would return an error if i had a value greater than or lesser than it should have at any point, or if the total equalled any value other than 10. These can be left in for submission as they don't output unless something goes wrong; however, check that your `assert()` won't trigger due to any of the test cases on WebSubmission, otherwise you may be marked as having failed that case.

6 Testing

Ideally, you've already begun writing test cases before starting debugging - writing test cases is a powerful form of planning, as it allows you to consider how your program should react in unusual situations and therefore what data structures and algorithms are best. But how do you write a good test case? And how do you test methodically?

Here's our advice:

- Use the sample data given in the question as test cases to begin with, but go beyond these.
- Consider your boundary cases. If you're using numbers as input, what are the maximum and minimum values? If you're using strings, does an empty string work? How about a long one?
- Think about invalid inputs. This is not always necessary for assignments, but addressing these inputs is a good habit to get into, especially for larger projects. If your code is expecting a number less than 10, what happens if '11' is inputted, or 'twelve', or '$'?
- Check what the response should be! Testing a piece of code that is designed to add numbers when you are unsure of what the sum actually is will be pointless. Figure out what the response should be without reading your code- if it's complex, use paper and a calculator to track your thoughts.

Once you're actually beginning to test against these cases, different advice applies:

- If you change your code, test it against **all** your test inputs again, in case your change has affected the response. This is good practice, even if your change shouldn't affect the response, as you may have accidentally changed a vital piece of code. Makefiles and .txt files can be used to simplify and speed up this process, but that's beyond the scope of this manual.
- If there are many errors, fix the simplest one first- fixing it may make the larger one smaller!
- Test each function and object individually before combining them, so that you can discover faults early and know where they are located.
- Make sure all your code is being tested. If there's an if-statement, test both branches.
- Figure out the cause of the error before moving on. If the input is 10 and the output should be 20, don't simply write `if (10) { return 20;}`. Not only do you learn nothing from this, but you are also at a disadvantage for fixing further bugs as your code is now more complicated.

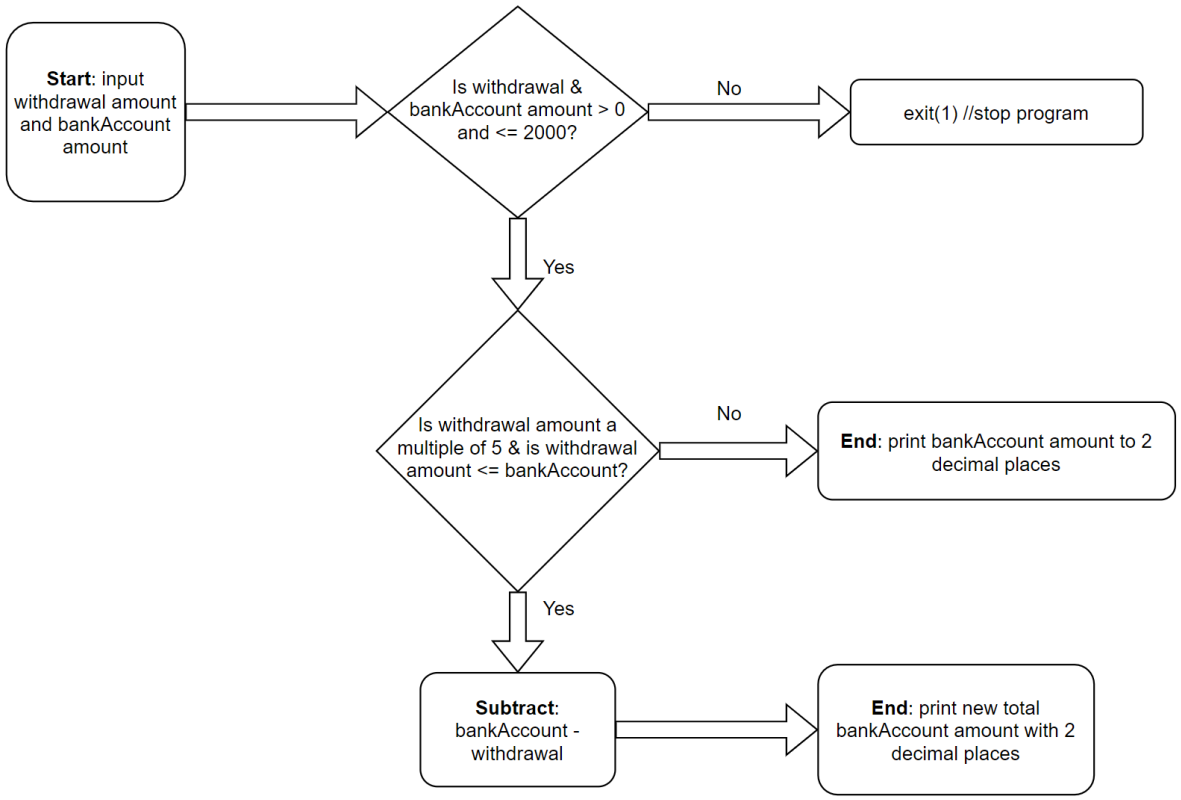
Don't just rely on WebSubmission to test your code- while it is good at doing this relatively quickly, testing yourself allows you to combine this technique with tracing and print statements, and writing your own cases allows you to better understand your program.

Good luck, and happy debugging!

Appendix

Examples of code planning

A flowchart showing an ATM program with 2 inputs: a withdrawal amount and bankAccount amount



A bubble sort algorithm written in pseudocode (obtained from ADDS lecture 7-3)

```
v[] <- {numbers}                    // Start with an array of numbers
n <- length{v}                      // Get the length
  for (i = n-1; i > 0 ; i--){       // Start from the end
    for(j = 0; j < i ; j++){        // Scan up to this point
      if(v[j] > v[j+1]){            // If this value is bigger than the next one
        temp <- v[j]                // Swap them
        v[j] <- v[j+1]
        v[j+1] <- temp
      }
    }end for loop
  }end for loop
```

Compiler error messages you may come across and their meanings

Error Message	What do they mean?	Possible Causes and Solutions
Undefined symbols for architecture x86_64:	The compiler can't define what some things are – maybe it doesn't have the information to define a function, or it's not sure where an attribute comes from.	Check that you're compiling the right files. Maybe you're using .h instead of .cpp?
Duplicate symbol(s) for architecture x86_64	This is the opposite of the above error – there's two or more definitions for something, and it can't figure out which to use!	Check that all your function names are unique- perhaps there's two main functions? Also ensure you've only included .h files within your .cpp files – including a .cpp file causes this error.
Segmentation Fault	You're accessing memory outside of what your computer wants you to access at that moment. Maybe you're dereferencing a null pointer, or using a deallocated pointer. This error also occurs when you overflow an array.	Segmentation faults are generally caused by going beyond the bounds of arrays in our experience – check this isn't the case for you! Make sure how you're using pointers in your code, and that you're not dereferencing pointers that point to nothing.
Bus Error	This one's similar to segmentation faults, but it occurs when you try to access memory that can't be found in the first place; for example, because the memory doesn't exist! Luckily, this error is pretty rare.	Check that you're using pointers correctly. We've had this occur once when using strings previously – check if any strings might be related to the cause.
SOMETHING does not name a type	This one's fairly understandable – your compiler thinks you're declaring a type (eg. int, char) but what you've written isn't a type (eg. cha, cout).	There may be a mistake in your type name. If you're not trying to name a type at all, check that you're calling your functions within a larger function (eg. your main function).
No such file or directory	You have asked the compiler to compile a file that does not exist	Before you compile, make sure that the files you are compiling are spelled correctly and you are compiling them in the right directory
Expected 'x' before 'y'	You're most likely missing a semicolon or bracket that is preventing your program from compiling. This is commonly a syntax error where the compiler cannot translate your program into executable code	Generally compiler error messages will tell you which line the error is occurring. Use that to your advantage and go through that line to see if there is a missing semicolon or parenthesis

Was not declared in this scope	When a type, variable, or function is declared not in a scope, it means that your program cannot find its definition	Make sure that you include any standard libraries so that functions are declared and ensure that your variables are declared and initialised properly, ideally at the beginning of your program