RTDM: Read the Debugging Manual

**Table of Contents**

0 [Introduction and Summary of Sections](#introduction_and_summary_of_sections)3

1 [Reducing Bugs](#reducing_bugs)4

1.1 [Pair Programming](#pair_programming)4

1.2 [Program Planning](#program_planning)4

2 [Emotional Awareness](#emotional_awareness)5

1.1 [Peer Discussions](#peer_discussions)5

1.1 [Creative Outlets](#creative_outlets)5

3 [Reading the Compiler](#reading_the_compiler)6

4 [Code Tracing](#code_tracing)7

5 [Print Statements](#print_statements)8

6 [Testing](#testing)9

[Appendix](#appendix)10

**0 Introduction and Summary of Sections**

As a programmer, no doubt one of your great tasks in life is to debug bugs. We all make mistakes when we code

This manual is about how to fix bugs

Definition of debugging

Summary of Sections

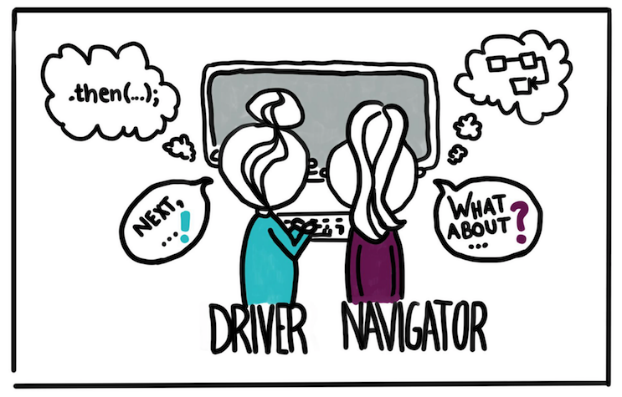
**1 Reducing Bugs**

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.



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

1. Communicate the roles and split the workload so everyone knows what each person is doing.
2. 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.

**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 in plain English), come up with potential solutions. Do not worry about any coding conventions, simply jot all your ideas down.
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](#examples_of_code_planning).

**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 very best thing that you can do when dealing with compiler logs is:

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](#compiler_error_messages_and_their_meanin). 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**

**5 Print Statements**

**6 Testing**

**Appendix**

**Examples of code planning**

**Compiler Error Messages and their Meanings**