Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia

Using Quicksand to Improve Debugging Practice in Post-Novice Level Students

Joel Fenwick
The University of Queensland,
Earth Systems Science
Computational Centre,
joelfenwick@uq.edu.au

Peter Sutton
The University of Queensland,
School of ITEE
p.sutton@itee.uq.edu.au

Abstract

The ability to debug existing code is an important skill to develop in student programmers. However, debugging may not receive the same amount of explicit teaching attention as other material and the main expression of debugging competence is students' ability to undo problems which they themselves have injected into their assignments. Further, as the literature points out, debugging skills do not necessarily develop at the same rate as code writing skills.

This paper discusses an intervention in a second year course designed to improve students' application of simple debugging techniques. We use a puzzle based approach where students are graded based on the number of attempts they take to locate misbehaving code in a program which they did not write but whose function they understand. An existing assignment component addresses another aspect of debugging practice.

1 Introduction

The context for this work is a second year course in systems programming (networks and operating systems). Because of its place in the degree program, it also does triple duty as a means to force students to improve their programming skills and to learn the language used in the course (`C`). All students enrolling in the course have some exposure to `C` but much of their basic training has been in `Python` or `Java`.

This setting is a little different from the typical setting in the literature (McCauley et al. 2008, Fitzgerald et al. 2008), in that we are not (or should not be) dealing with absolute novices any more. These are students who have some level of programming skill even if they do not have much initial familiarity with `C`. However, "debugging is a skill which does not immediately follow from the ability to write code." (Kessler & Anderson 1986)[p208]. Following on from an earlier working group, an ITiCSE 2004 group (Lister et al. 2004) considered whether deficiencies in programming ability *after* initial programming courses were due to lack of problem solving skill or were in fact due to a fragile knowledge of programming and code reading ability. However, "reports on interventions designed to improve students' debugging skills have not been common in recent literature." (McCauley et al. 2008)[p83]

The course has four assignments. The first, third and fourth assignments are traditional programming assignments where the students must write whole programs (possibly making use of a provided solution for the previous assignment). The focus of the second assignment is debugging. In 2011, it consisted of two components: the binary bomb and quicksand (both explained below). Both parts were conducted electronically and only required a network connection to a school server.

The binary bomb is a modified version of an assignment run at Carnegie Mellon University (Bryant & O'Hallaron 2001). This component tests students' ability to use a debugger. Students are given a precompiled program with some of the debug symbols removed as well as a small part of the source code. They must use the debugger to examine the workings of the program to determine the passwords to "defuse" the bomb. We have used the binary bomb for a number of years and it seems popular with the students with the puzzle solving aspect mentioned in particular.

However, debuggers are only as useful as the questions they are asked and some students start to view the debugger as the first port of call in solving any problem even when they do not know what they are looking for. Further, some students would reach the end of the course and still start their requests for help with "My program doesn't work." That is, they did not seem to be able to locate or describe the problem more precisely. There is also a school of thought [typified by Linus Torvalds' refusal to incorporate kernel debuggers into the Linux Kernel (Torvalds 2000)], that over reliance on a debugger produces sloppy programmers. James et al. note that "the more tools offered, the less students think for themselves. They try to get the tool to do the thinking...". (James et al. 2008)[p28]

Addressing these and other problems is the purpose of the quicksand component, and the focus of this paper. However, since this is not an introductory course, some of the students do already have good debugging technique and care must be taken not to bore these students while trying to lift the others.

1.1 Debugging Challenges in `C`

Debugging programs written in `C` presents some challenges. These are by no means unique to `C` but when they happen in `C` (and similar languages) they tend to be more spectacular and less help is available. Possible problems include:

1. Compilers (more specifically optimisers) can change things and sometimes they get it wrong.
2. Functions can have bugs or undocumented behaviour.

3. A badly written statement can create problems which only appear much later in seemingly unrelated places (e.g. heap corruption).
4. The lack of structured error handling.
5. Corruption from non-thread safe actions.[1]

Regarding the first point, (some) students seemed to believe quite strongly that the compiler was transparent. The idea that what was executing could be different to what they saw when they looked at the source was problematic. One of the authors' first experiences with this as a student was realising that an optimiser had decided to run a loop in reverse.

This is not to say that trusting the compiler or documentation is a bad starting heuristic, but it seemed to be a weakness in the students' understanding that translation failure was unthinkable. It may be that even just allowing for that possibility might enable students to perform the necessary testing to discover the real problem.

2 Quicksand

The quicksand component (created by us) is intended to develop and test a complementary set of skills to that of the binary bomb. Instead of using the debugger to step through and examine variables, the students can make small modifications to a piece of source code (typically printing the values of variables), then request that the code be recompiled and run against a set of tests. The students can use output from this run to locate bugs in the program.

The students were told quite explicitly that they were looking for the original cause of the bug not just the location where symptoms became apparent. For example an uninitialised variable might not cause any obvious effect until much later.

As in the work by Fitzgerald et al. (Fitzgerald et al. 2008), we only require students to identify the line on which problems occur and not to actually fix the problem. Firstly, as they note (citing Kessler and Anderson), "the skills required to understand the system are not necessarily connected to the skills required to locate the error."[p95] Secondly, it may not be possible to theorise about the cause of a problem until its location has been constrained. Much time has been wasted looking for a problem in the wrong place because programmers jump to conclusions. Finally, since the students are supplied with code to debug rather than writing it themselves, we need to address the issues of understanding and context (Fitzgerald et al. 2008). To this end, the supplied code is a (suitably buggy) sample solution to Assignment 1. This means that the students will be familiar with the purpose of the program and what it should be doing. It is also sufficiently large that the entire program cannot be held in mind at one time and some investigation is required to locate bugs precisely.

Lines which may contain bugs have a "tag" at the end in the form of a right-justified comment (e.g. `/* QS:f8y5d */`). We use tags rather than line numbers because line numbers will vary when students add new lines to the source. Also tags are harder to mistype. Lines which are not possible bug locations do not receive a tag but are marked `/* QS: */`. This makes it easy to distinguish lines which are in the supplied source as opposed to lines added by students.

When editing the source, students may insert additional lines but may not modify any existing lines. This is to discourage students from trying to reimplement sections they are suspicious of. In practice such an approach would only work in a limited number of instances.

We also limit the number of extra lines which the students can have in the source at any one time. This is to discourage the creation of massive quantities of debug information which is then impossible to follow. Instead, we want students to focus their attention and refine their theories as to the location of the problem. If they exceed this limit, they must remove some lines or get a clean copy of the source before they will be able to recompile.

[1] This is another thing which can cause seemingly inexplicable behaviour — we do not address thread safety issues in this exercise.

2.1 The `quicksand` tool

The quicksand tool itself has four commands:

- `get` — puts a "clean" copy of the source to be debugged in the current directory.
- `test` — processes and compiles the student's modified source and runs the set of tests. The outputs from the tests are placed in the student's directory. At no time does the student have access to the compiled binary itself.
- `guess` *tag* — Records the student's "guess" that the tagged line contains a bug. It will tell the student if their guess is correct.
- `status` — Reports how many attempts the student required to locate each bug as well as their current and maximum possible marks.

To encourage students to use online documentation, the assignment specification and tool instructions were only available as a `man` page.

2.2 Marking

The marks gained for correctly determining the location of a bug were

$$\frac{T}{B} \cdot 0.9^{g-1}$$

where T is the total marks for this part, B is the number of bugs in the system and g is the number of guesses since the previous correct guess. This follows a similar "exponential decay" scheme used in the bomb and ensures that more wrong answers will decrease the mark but eventually answering correctly still gives more marks than giving up. Since the number of tags in the program is limited though, we did impose a limit of 40 on the total number of guesses for all bugs. Only eleven of approximately 140 students used all 40 guesses.

Note that editing and testing are free actions, the students can do as much testing as they wish without it influencing their marks.

2.3 Bugs

All students were given the same piece of code as a starting point but different combinations of bugs were introduced for each student. The bugs were chosen so that any one of the bugs would cause at least one of the tests to fail. That is, the students could use test failures as the starting point to trace the bug.

Bugs introduced by quicksand fall into two broad categories:

- Visible in source — the source given to the student has a logic error in it. The student could find it by inspection. Examples include:
  - Uninitialised or incorrectly initialised variables.
  - incorrect loop limits or steps
  - inverted if conditionals
  - invalid memory access.
- Hidden — These can not be found by inspection since they do not appear in the source which the students are given. When the student runs quicksand's test command, their source is transformed. Some lines are replaced or modified before the program is finally compiled. For example:
  - Statements can be skipped (removed from the source).
  - Assignments and initialisations changed to assign different values.
  - Loops can finish early or skip iterations (modified limits or step)
  - Variables can be modified unexpectedly.

It is important to understand that while the precise details and causes of these "hidden bugs" are artificial, the symptoms are not unreal. Although some types (e.g. statements being skipped) are thankfully rare[2] "in the wild".

The "hidden" bugs are representative of instances where either the mental model of the programmer (expressed in code) does not match what is actually there; or where the documentation is incorrect or (in rare cases) where the compiler or standard libraries have bugs.

The possibility of bugs in code not written by the students represents a point of difference between beginner and later programmers. Courses for beginners will focus on a relatively small, well tested and well understood subset of the standard libraries for their language. Later on however, programmers need to be able to make a distinction between source and running code; between what documentation or their understanding suggests and what is actually there.

Some work (Lee & Wu 1999, Ahmadzadeh et al. 2007) used debugging exercises as a means to improve general programming skill. There is nothing wrong with this approach and training programmers to write less bugs is a good goal. However, debugging is not solely a means to rectify one's own coding faults, which could be avoided by writing more carefully. Even a perfect programmer needs to be able to debug. Debugging is also an independent skill which may be required whenever code is brought together or when some part of the environment changes.

So how do the students find bugs that they can't see? Actually, whether the bugs are immediately visible is not immediately relevant to finding them. At this stage in their development, students are writing programs which are too large to be completely comprehended at one time. Eisenstadt's "war stories" article (Eisenstadt 1997) contains a number of memorable terms for difficulties in finding bugs: the "Cause/Effect Chasm" where the cause and effect were too far apart to be easily found; and WYSIPIG — "what you see is probably illusory guv'nor" where the programmer misreads or misunderstands what they are looking at. In a large system without knowledge of the (general) location of the problem, finding problems by inspection is not feasible. So initially, visibility is not critical. Instead, all the bugs introduced by quicksand can be located using two generally applicable techniques.

- Strategically placed output statements to trace execution flow. Once the symptom has been identified (e.g. a crash or incorrect output).
- "binary search" — output the values of critical expressions at and before the symptom location, choose a point between them and repeat until the cause of the symptom is found.

These are pretty rudimentary methods but they are useful and (some) students do not seem to be applying them. But why not use more advanced tools for this exercise? After all Lieberman wrote that "It is a sad commentry on the state of the art that many programmers identify 'inserting print statements' as their debugging technique of choice." (Lieberman 1997)[p27] Firstly, we want to encourage students to use hypothesis testing rather than trial and error (Ahmadzadeh et al. 2007). This constrained environment prevents them from relying on other tools too much. Secondly, James et al. (James et al. 2008) suggest that more advanced tools may actually hinder students from learning good technique. Thirdly, there are a number of common environments (such as web programming) where more advanced tools may not be available but simple techniques work everywhere (James et al. 2008).

In part, this is an application (although not a rigorous one) of the `malicious adversary` concept from theoretical computer science. That is, if you can find bugs using these techniques when you are being deliberately sabotaged, then they will be useful under normal conditions as well.

3 Security and Integrity

This exercise combines two factors which make the security of the system a concern. Firstly, quicksand needs sufficient privileges to access the database and record attempts. Secondly, students will be able to inject "arbitrary" code into the test program.

With this in mind we tried to impose limits to make the code they could inject less arbitrary. Lines added by students could not contain #[3], any of a number of system functions nor any raw assembly or system calls.

Privilege issues were dealt with by dropping to student privileges whenever interacting with student code and by preventing students from attaching a debugger to any of the programs involved. Further technical details are beyond the scope of this paper.

There are other issues related to the integrity of the system that don't relate to security. It is not enough that the students work out the answers, we want them to use the correct technique in doing so. An example of an incorrect technique is removing "suspicious" lines and by means of comments, loop bodies which never execute and so on. Our intent is that the students insert small "probes" to determine what is going on, not switch out chunks of code. For this reason the *real* source (as opposed to the version the students edit) contains extra calls to check that statements are not being executed out of order. Further, some bugs add additional calls to enact their

[2] One of the authors has encountered such errors a number of times including, in an unrelated piece of code while developing quicksand.

[3] or any of its equivalents

bad behaviour. If the student inserts an incomplete line immediately before such a call, then the compiler may "helpfully" tell the student that the next line contains `garbleData()`. To avoid this, only the line number of compile errors are reported back to students.

While a number of students noticed the forbidden words and symbols list, only one student (that we know of) accidentally ran into the anti-reordering checks.

A non-technical aspect of assessment integrity is the possibility of collusion. What if the students discuss their bugs? Does this disadvantage students who start work early? While sharing answers is possible[4], it presents a number of difficulties for students. First, students would need to know that they had been assigned the same bug. Even if they establish that they have some bugs in common, this does not allow them to infer that any of their other bugs are the same. Establishing with certainty that you share a bug with someone else (without using an attempt) requires the same skills as finding the bug "the proper way." Now, other types of collusion are possible, such as one student doing much of the work for another student but this risk is (we believe) no higher than a traditional assignment.

Secondly, sharing answers with students who have not put the work in acts against the students' self-interest. Suppose Student A took 3 attempts to identify the location of a bug, and they establish that Student B also has this bug (not withstanding the difficulties in doing so). Now suppose that Student A gives the answer to Student B who has made less than 3 attempts, Student A has given Student B an advantage, B will now get more marks for that bug than A.

Thirdly, students who start work early are perhaps less likely to cheat.

4 Results and Reflection

The biggest teaching challenge here is to get the students to not blindly trust their intuition but rather to test the safety of their ground before relying on it too heavily[5]. At the same time, we need to show them that reason and logic still apply in debugging situations.

Fitzgerald et al. describe students in their study as having a "stubborn desire to understand and debug code through reading alone." (Fitzgerald et al. 2008) This was borne out in this work where students took the source, worked on it and tested it entirely outside of quicksand. This created confusion when the code ran differently under the normal compiler as opposed to the malicious quicksand. In future this would need to be prevented. The easiest approach would be to ensure that the students are not given the source for all routines and hence would not be able to compile independently.

We have a number of sources of information to use in evaluating quicksand.

1. Anonymous surveys taken immediately after the assignment.
2. Logs of the types of questions asked in tutorials.
3. Questions asked on the course online discussion group.
4. Teacher impressions.

[4] We have no evidence that this occurred.

[5] Hence "quicksand".

We also have university end-of-course surveys but the responses did not reveal anything about this assignment.

Since the survey was voluntary and students were not required to attend tutorials or post to the discussion group, we need to be careful about what conclusions we draw. A student who has no problems that they can not fix for themselves will not show up. Also, considering only those students who ask for help in tutorials tends to give a worst case approximation of how things are going. A lack of more sophisticated questions could either mean that people are better able to fix their own problems and do not need to ask or that they are getting stuck at a basic level. Taken together, this means that we may only be able to consider a lower bound on improvements.

We will now discuss each information source in more detail.

4.1 Assignment surveys

This survey asked the students a number of questions about quicksand and the binary bomb. Some were to be answered on a five point scale while others were free text. Of the 77 respondents, 49% said that they had learned a lot from quicksand, while 57% said that quicksand had improved their confidence in their debugging abilities.

Interestingly 12% of students said that they already knew and used such techniques (and as such are unlikely to report learning a lot) and some of them still reported improved confidence.

The importance of retaining the binary bomb component is shown by the fact that 51% of respondents said that they did not know how to use the debugger prior to this course[6].

Taking both parts of the assignment (quicksand and binary bomb) together, 60% said that they had learned techniques that they could have used in the previous programming assignment. 52% said that they were more systematic when debugging now.

The free text responses indicate that some students were not convinced that programs could go wrong in the way that our exercises did. As such a wider set of possible problems is probably indicated. For example, a bigger focus on function calls misbehaving (where they can not see the source) would probably be more acceptable to them. A number of students seemed to believe that using printf wasn't real debugging. This seems to put even simple techniques in the category of fragile knowledge for some students. Students did not seem to be surprised or confused when these techniques are pointed out, but did not seem to have considered applying them to solve their problems so we are dealing specifically with "inert knowledge" (Perkins & Martin 1986).

4.2 Tutorial Logs

Each tutor logged the number of questions they answered by category. Questions about debugging were classified into the following categories. They are ordered by the sophistication of the question (roughly how much work the student has put in before they ask the question):

- debugA — Questions of the form "It doesn't work." or "I'm failing test #5."[7]. Here there

[6] It is not clear from these answers whether all of those students are referring to all symbolic debuggers or just non-IDE ones.

[7] Students were given a set of automated tests which they could use to test their assignments against some parts of the spec.

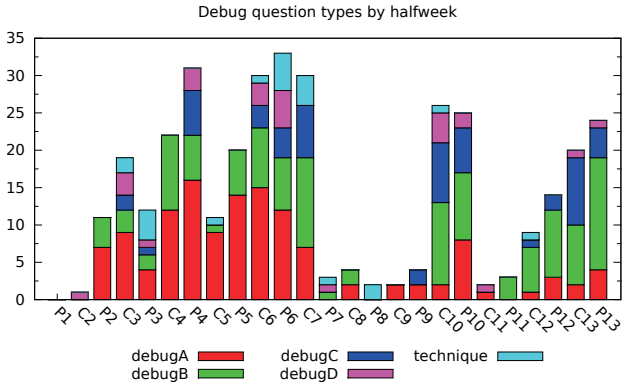

Figure 1: Cn denotes tutorials conducted in the first half of the week while Pn denotes tutorials from the second half.

is no description of the problem and no work apparent towards identifying the location, cause or triggers for the problem.

- debugB — Some basic effort has been made to locate the problem. Some ability to describe the specifics of the problem.
- debugC — The student demonstrated good technique.
- debugD — The student did all the right things but were prevented from finding the problem due to missing some knowledge or a misunderstanding. Essentially, their technique was not the problem.
- technique — the student was not asking for help fixing a particular problem but wanted to know about debugging techniques in general.

Tutorials in this course were run in two half-weeks (Monday to Wednesday morning and Wednesday afternoon to Friday afternoon). Students were required to enrol in one session in each half-week. However, tutorials were not compulsory and while some weeks were set aside for the tutors to teach extra material, the majority were general help sessions. As such, the attendance varied significantly depending on the time to deadline.

The breakdown can be seen in Figure 4.2. The assignments were due at the end of Weeks 4, 7, 10 & 13 respectively. Note that once the questions start to increase again for Assignment 3, the more sophisticated categories dominate the `debugA` questions.

4.3 Discussion Posts

The course had a very active online discussion group. We separated out the threads which asked for help or suggestions for fixing bugs in Assignments 1, 3, 4. These were classified according to the initial question using the same scheme used for tutorial questions. The number of questions (and the number of different students asking them) increased from Assignment 1 to Assignment 3 (the assignment the students seemed to find most difficult) and then dropped below the level of Assignment 1 for Assignment 4. The majority of the questions fit into debugA — either little description of the problem or a request for suggestions about where to get started. In Assignment 4, there was an increase in the proportion of questions which described features of the system the students were trying to fix rather than a particular test failure.

4.4 Teacher Impressions

As well as visiting tutorials from time to time one of the authors conducted intensive help sessions just before the deadline for Assignments 3 and 4. Unfortunately the question types from these sessions were not logged. The first session was not particularly well attended. The second session saw at least 40 students and from memory, the majority of questions were debugB or above.

5 Implementation Considerations

What is required in order to run an assignment like this? In terms of software infrastructure, A database for storing marks and infrastructure (quicksand in our case) for distributing source and compiling student modified versions will be needed. The following should also be considered.

- *The security of the marks record.*
  Students must not be able to coerce the system into modifying marks for themselves or other students. They must not be able to view the marks of other students.
- *Backdoor solutions.*
  What mechanisms are available for bringing other code into the test program? For example, in Java, it would not be sufficient to block `import` since classes could be pulled in via their full name or using Java's reflection API[8]. In Python there is `eval()`, the `pdb` debugger and probably others. This is not to suggest that these languages can not be secured[9] but the implications of these features need to be considered.

  Can your hidden modifications be exposed in compile errors or exception traces?
- *Types of Bugs to inject.*
  Firstly, the bugs must be plausible in your source language[10]. For example a string changing for no apparent reason in Python or Java (where String objects are immutable) would be a bad choice. Secondly, bugs that do not require modification of the student's submitted source require much less machinery and are less fragile than things like skipping particular statements. Limiting oneself to function calls which misbehave under certain conditions would save a lot of work.

6 Conclusions

From the survey, more than half the students reported increased confidence in their debugging abilities as a result of quicksand. This and more than half reporting they were more systematic in their debugging from the assignment as a whole, are encouraging. Looking at the online discussion is less positive but students may have been less willing to post detailed questions. Either because they weren't sure how to express them in text or because of the warnings they were given about posting code. Overall, for non-pathological bugs this approach shows promise.

[8] API = Application Programming Interface
[9] A reviewer suggests `SecurityManager` in the case of Java.
[10] That is, the language of the program the students are editing

The *possibility* that the compiler or libraries misbehave should be part of the students' thinking. It appears though that more explanation needs to be given, perhaps with real world examples, for students to accept this.

Acknowledgements

This work was supported in part under AuScope sustainability funding.

References

Ahmadzadeh, M., Elliman, D. & Higgins, C. (2007), 'The impact of improving debugging skill on programming ability', *ITALICS* **6**(4), 72–87.

Bryant, R. E. & O'Hallaron, D. R. (2001), 'Introducing computer systems from a programmer's perspective', *SIGCSE Bull.* **33**, 90–94.
URL: *http://doi.acm.org/10.1145/366413.364549*

Eisenstadt, M. (1997), 'My hairiest bug war stories', *Commun. ACM* **40**, 30–37.
URL: *http://doi.acm.org/10.1145/248448.248456*

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphyd, L., Simon, B., Thomas, L. & Zander, C. (2008), 'Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers', *Computer Science Education* **2**(18), 93–116.

James, S., Bidgoli, M. & Hansen, J. (2008), 'Why Sally and Joey can't debug: next generation tools and the perils they pose', *Journal of Computing Sciences in Colleges* **24**, 27–35.
URL: *http://portal.acm.org/citation.cfm?id=1409763.1409770*

Kessler, C. M. & Anderson, J. R. (1986), A model of novice debugging in lisp, *in* 'Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers', Ablex Publishing Corp., Norwood, NJ, USA, pp. 198–212.
URL: *http://act-r.psy.cmu.edu/publications/pubinfo.php?id=220*

Lee, G. C. & Wu, J. C. (1999), 'Debug it: A debugging practicing system', *Computers & Education* **32**(2), 165 – 179.
URL: *http://www.sciencedirect.com/science/article/pii/S0360131598000633*

Lieberman, H. (1997), 'Introduction to the special issue on the debugging scandal', *Commun. ACM* **40**, 26–29.
URL: *http://doi.acm.org/10.1145/248448.248455*

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004), 'A multi-national study of reading and tracing skills in novice programmers', *SIGCSE Bull.* **36**, 119–150.
URL: *http://doi.acm.org/10.1145/1041624.1041673*

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simone, B., Thomas, L. & Zanderg, C. (2008), 'Debugging: a review of the literature from an educational perspective', *Computer Science Education* **2**(18).

Perkins, D. & Martin, F. (1986), Fragile knowledge and neglected strategies in novice programmers, *in* S. E. & I. S., eds, 'Empirical Studies of Programmers', Norwood, NJ: Ablex Publishing Co., pp. 213–229.

Torvalds, L. (2000), 'Re: Availability of kdb', Linux Kernel Mailing List.
URL: *http://lkml.org/lkml/2000/9/6/65*