Guide to Debugging

Introduction to Debugging

Debugging is the process of finding and fixing problems in code. For most programs, the time spent debugging far outweighs the time spent writing new code. Whether students or professional engineers, all programmers get bugs, and debugging is a normal part of the programming process.

Although students may see bugs as inconveniences to be eliminated as soon as possible, bugs in student programs should be seen as opportunities to reinforce positive attitudes toward debugging and persistence, identify and address student misconceptions, and further develop good debugging skills. Your role as the teacher is to avoid directly debugging for your students, but to guide students in better taking advantage of these opportunities.

Reinforcing positive attitudes

Finding bugs is the first step toward fixing them. Programmers deliberately test their code in order to uncover any possible bugs. Celebrate the discovery of new bugs as students report them in the classroom, framing finding a bug as the first step to fixing it. Model enjoying the interesting or funny behaviors a bug can cause, such as making a sprite move in unexpected ways, or distorting an image on a web page. Remind students that if programs all worked exactly as they wanted the first time, programming wouldn't be as interesting or fun. Encourage students as they get frustrated, reinforcing the debugging strategies and students' self-efficacy as they improve their debugging skills, and talk about the bugs that you get in your own programs, reminding them that everyone gets bugs, even professional software developers.

Identify and address misconceptions

Often a bug occurs because students have misconceptions around how the computer interprets their code. As part of the debugging process, have students explain their code, line by line, reminding them that the program is really only doing exactly as it was told. If the program displays an error message, ask the student to connect the message to what is happening in the code itself. Prompt them with relevant questions about how the associated programming structures (such as conditionals or loops) work, and refer them back to previous lessons, worked examples, or documentation when needed. After students have found the bug, ask them to reflect on what they learned about the associated programming structures and how they could explain it to a friend with a similar bug.

Develop debugging skills

As students get more experience debugging their programs, they will build the skills they need to debug more independently. Help students to generalize the strategies that they use by asking them to reflect on what processes were effective and reframing those processes as a general strategy. They should also learn ways to simplify the debugging process by making their code more readable with good naming conventions, clear formatting, and relevant comments; organizing code into functions or other logical chunks where appropriate; and testing small pieces of code as they go. Call out these practices as facilitating debugging as you help students with their code.

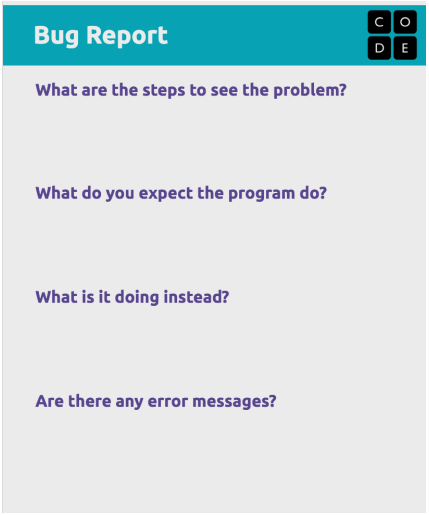
Debugging as problem solving

In computer science, debugging is framed as a form of problem solving, and students can use a version of the four step Problem Solving Process as a framework for debugging their programs. Just as in other forms of problem solving, many students may jump to the "Try" part of the framework and begin making changes to their code before understanding the nature of the bug itself. Remind them that all parts of the process are important, and that ignoring the other three steps will actually make debugging more time consuming and difficult in the long run.

Define - Describe the bug

In the context of debugging, defining the problem is describing the bug. This step can be done by anyone using the program, not just the person who will eventually be debugging it. Students will need to know the following information before they move on to the next step:

- When does it happen?
- What did you expect the program to do?
- What did it do instead?
- Are there any error messages?



Some bugs will keep the code from running at all, while others will run, but not correctly, or will run for a while, then stop or suddenly do something unexpected. All of those things are clues that will help the student find the bug in the next step.

You can encourage students to clearly describe the bugs they find by having them write up bug reports, such as in this worksheet. As you foster a positive culture around debugging, encourage students to write bug reports for their classmates' code as well as their own.

Prepare - Hunt for the bug

In most cases, hunting for the bug (the "prepare" step in the debugging process) will take up most of a programmer's time. Remind students that it's natural to take a long time to find a bug, but that there are things that will make their search easier.

1. **Why is the bug happening?**
   Often students focus on what they want the program to do and why they believe their code is correct, rather than investigating what could be causing the bug. Encourage students to start with the error messages and what is actually happening when the program is run, and try to connect that with the code that they have written, rather than explain why their code "should" be working. They can also "trace" their code, by reading it line by line, not necessarily from top to bottom, but in the order that the computer would interpret it while the program is running. Using debugging tools such as watchers, breakpoints, or the inspector tool may help them to identify why the code is running as it is.
2. **What changed right before the bug appeared?**
   If students have been testing their code along the way (as they should!), they can focus on code that they have recently changed. As they investigate that code, they should follow the logic of their program in the same order that the code runs, rather than reading the code line by line from top to bottom.
3. **How does this code compare to "correct" solutions?**
   Students should also make use of the various resources available to them, such as working projects, examples in previous lessons, and code documentation. Have them compare the patterns that they find in the documentation and exemplars to their own code, differentiating between the programming patterns that should be the same (loops, counter pattern, HTML syntax) and specifics of their program that will be different (variable names, image URLs, coordinates).

Try - Change the code

If students believe that they have found the bug, they can go ahead and try to fix it, but in many cases they may want to make changes to the code to narrow down their search. Some common strategies include:

1. **Commenting out code**
   By commenting out sections of the program, students can narrow down the part of the program that is causing the bug. After commenting out large sections of code, function calls, or html elements, test whether the bug is eliminated. This method is especially helpful when students have used good modular programming techniques.
2. **Print to the console**
   Using the console log command can help students to understand whether a conditional has been triggered, or keep track of a quickly changing variable over a period of time.
3. **Change the starting values of variables**
   Changing the starting values of variables can make it easier to reproduce a certain bug. For example, students may want to change the starting score to 99 to test a bug that only occurs when the score reaches 100. They may want to set their number of lives to a very high number to allow them to test for longer before losing the game.
4. **Amplify small effects**
   Sometimes bugs have such tiny effects that it's difficult to investigate them. Amplifying small effects, such as making elements move further on the screen or giving web page elements a background color to make them more visible, can make it easier to understand what is happening in a program.

In many cases, students will introduce new bugs during the debugging process, especially if they are randomly changing code as part of a "guess and check" method. Prompt them to explain why they are changing the code, and encourage them to make small changes and test them often, making it easier to go back if their solution didn't work.

Reflect - Document what happened

As students make changes and see the effects, they should reflect and document on their experiences. This will help them to build a better model of how the program constructs work and what debugging strategies are most effective. Students should consider the following after they have eliminated a bug from their program:

1. **What caused the bug?**
   The computer had a reason for doing what it did, and students should understand why their code caused the bug itself. There may be a rule that they can share with others in the class ("There is no closing tag for images") or a misconception ("Sprites all get drawn to the screen when `drawSprites` is called, not when they are updated.") that they can clear up.
2. **How did you find the bug?**
   Have students describe the debugging process that they used, paying special attention to how the type of bug and any error messages lent themselves to particular debugging strategies. Help them to generalize their strategies so that they can use them in a variety of situations. (e.g. "I had to capitalize the 'r' in my variable" might become "You double checked the capitalization and spelling of the variable the program didn't recognize.")
3. **What in your code made it easier or harder to debug?**
   Debugging is a great time to reinforce "clean code" practices, such as good naming conventions, use of functions or other ways of "chunking" code into logical sections, commenting, and clear formatting. Point out when comments or well named functions and variables make it easier to trace code and find an error. Debugging is also easier when students have separated out code into logical chunks and functions, which can be commented out individually.

Writing debuggable code

Various practices will make it easier for students to debug their code. Encourage students to neatly format their code, as well as make good use of comments and whitespace. Organizing code into logical chunks, using functions, and having reasonable names for classes, variables, and functions will help them to read and interpret their code as they debug. Point out times when students' good programming practices have made it easier for them to debug their own code.