lab_debug

Disastrous Debugging

Due: Jan 27, 23:59 PM

Doxygen | Lab handout | Handout solution | Lab slides

Preface: How Did We Get Here?

Your lab partner is writing an image recognition program. They are working on the tracing algorithm, which turns the image into a trace of the outlines in the image. After going through all the compiler errors (`sketchify.cpp:33`, etc), the program finally compiles! Overjoyed to have a program, you both decide to test it on a couple images.

```
Segmentation Fault
```

Ouch. What do we do now?

Getting Set Up

From your CS 225 git directory, run the following on EWS:

```
$
```

```
git fetch release
git merge release/lab_debug -m "Merging initial lab_debug files"
```

If you're on your own machine, you may need to run:

```
$
```

```
git fetch release
git merge --allow-unrelated-histories release/lab_debug -m "Merging initial lab_debug
files"
```

Upon a successful merge, your lab_debug files are now in your `lab_debug` directory.

Make this Yours!

Your partner left one thing out of the code because they weren't sure... what's your favorite color?

Use a website like HSLA Color Picker to find your favorite hue (the first slider). Once you found it, edit `sketchify.cpp:32` and replace `-1` with your favorite hue.

Determining What's Going Wrong

You could start and open `sketchify.cpp` and try to figure out what's happening. This is good for logical bugs – when you only rotate half of your image, for example, or the image doesn't rotate at all. Walking through what your code does to yourself or your partner is a good exercise in debugging bugs in your algorithm. However, this is often a poor choice for debugging runtime errors or general code bugs. In this case, you should attempt to use the following workflow to debug your code.

Debugging Workflow

From: *DEBUGGING: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems* by David J. Agans

1. **Understand the System.**
    - Without a solid understanding of the system (the system defined as being both the actual machine you are running on as well as the general structure behind the problem you are trying to solve), you can't begin to narrow down where a bug may be occurring. Start off by assembling knowledge of:
        - What the task is
        - What the code's structure is
        - What the control flow looks like
        - How the program is accomplishing things (library usage, etc)
    - When in doubt, look it up–this can be anything from using Google to find out what that system call does to simply reading through your lab's code to see how it's constructed.
2. **Make it Fail.**
    - The best way to understand why the bug is occurring is to make it happen again–in order to study the bug you need to be able to recreate it. And in order to be sure it's fixed, you'll have to verify that your fix works. In order to do that, you'll need to have a reproducible test case for your bug.
    - A great analogy here is to turn on the water on a leaky hose–only by doing that are you able to see where the tiny holes might be (and they may be obvious with the water squirting out of them!).
    - You also need to fully understand the sequence of actions that happen up until the bug occurs. It could be specific to a certain type of input, for example, or only a certain branch of an if/else chain.
3. **Quit Thinking and Look.**
    - After you've seen it fail, and seen it fail multiple times, you can generally have an idea of at least what function the program may be failing in. Use this to narrow your search window initially.
    - Start instrumenting your code. In other words, add things that print out intermediate values to check assumptions that should be true of variables in your code. For instance, check that that pointer you have really is set to `NULL`.
    - Guessing initially is fine, but only if you've seen the bug before you attempt to fix it. Changing random lines of code won't get you to a solution very fast, but will result in a lot of frustration (and maybe even more bugs)!
4. **Divide and Conquer.**
    - Just like you'd use binary search on an array to find a number, do this on your code to find the offending line! Figure out whether you're upstream of downstream of your bug: if your values look fine where you've instrumented, you're upstream of the bug (it's happening later on in the code). If the values look buggy, you're probably downstream (the bug is above you).
    - Fix the bugs as you find them–often times bugs will hide under one another. Fix the obvious ones as you see them on your way to the root cause.
5. **Change One Thing at a Time.**

    - Use the scientific method here! Make sure that you only have one variable you're changing at each step–otherwise, you won't necessarily know what change was the one that fixed the bug (or whether or not your one addition/modification introduces more).
    - What was the last thing that changed before it worked? If something was fine at an earlier version, chances are whatever you changed in the interim is what's buggy.
6. **Keep an Audit Trail.**
    - Keep track of what you've tried! This will prevent you from trying the same thing again, as well as give you an idea of the range of things you've tried changing.
7. **Check the Plug.**
    - Make sure you're assumptions are right! Things like "is my Makefile correct?" or "am I initializing everything?" are good things to make sure of before blindly assuming they're right.
8. **Get a Fresh View.**
    - Getting a different angle on the bug will often times result in a fix: different people think differently, and they may have a different perspective on your issue.
    - Also, articulating your problem to someone often causes you to think about everything that's going on in your control flow. You might even realize what is wrong as you are trying to describe it to someone! (This happens a lot during office hours and lab sections!)
    - When talking to someone, though, make sure you're sticking to the facts: report what is happening, but not what you think might be wrong (unless we're asking you what you think's going on).
9. **If you didn't fix it, it ain't fixed.**
    - When you've got something you think works, test it! If you have a reproducible test case (you should if you've been following along), test it again. And test the other cases too, just to be sure that your fix of the bug didn't break the rest of the program.

Basic Instrumentation: Print (`std::cout`) Statements!

The easiest way to debug your code is to add print statements. To do this, you can add comments at various points in your code, such as:

```
std::cout << "line " << __LINE__ << ": x = " << x << std::endl;
```

The above line prints out the current line number as well as the value of the variable `x` when that line number executes, for example:

```
line 32: x = 3
```

`__LINE__` is a special compiler macro containing the current line number of the file.

If you're getting compiler errors after trying to use `std::cout` statements, then you need to `#include` the `iostream` library like this:

```
#include <iostream>
```

Print statements work for debugging in (almost) any language and make repeated debug testing easy – to repeat debug testing with a new change, all you need to do is compile and run the program again. They also require nothing new to learn (smile).

What's up with this syntax?

`std::cout` is a *stream*, and `<<` is called the *insertion operator*. If you want to read more about streams and what this syntax means, you can check out this article on C++ input/output.

Debugging Your Code

To make and run the code, type the following into your terminal:

```
$
```

```
make
cp in_01.png in.png
./sketchify
```

Your First Bug

As you can see, your code caused a Segmentation Fault, or segfault. This happens when you access memory that doesn't belong to you – such as dereferencing a `NULL` or uninitialized pointer.

Try adding print statements to lines 39 and 43, before and after the calls to `original->readFromFile()`, `width()`, and `height()`.

Use the `std::cout` statement below on lines 39 and 43.

```
std::cout << "Reached line " << __LINE__ << std::endl;
```

Now run `sketchify` again. You'll see line 39 print out, but not line 43. This means the segfault occurred sometime between executing lines 39 and 43.

Work on getting your program to run to Line 43!

Bug 2

Once you've fixed the first bug, you'll get another segfault. You'll want to narrow down the line it's occurring on and its cause by printing more information. Try putting `std::cout` statements at the beginning and end of the inner for loop.

More debugging!

Like almost most all code written, the code isn't perfect. But fixing it is as simple as repeating the above to learn more about what the program is actually doing at runtime so that you can solve the issues. Good luck!

Checking Your Output

Once you think sketchify is working, you can compare your output (`out.png`) to the expected output by opening each image up using a graphical viewer. If you're on EWS, you can run the following two commands:

```
$
```

```
xdg-open out.png &
xdg-open out_01.png &
```

If you're working on your own machine, you can open the images using the photo viewer of your choice.

If the outputs differ, you've still got a bit more debugging to do – go back and add some print statements to figure out why the outputs differ!

You can also use the `compare` utility to generate a visual comparison of the images. For this to work, you'll have to temporarily change your color's hue to 280, since `compare` will expect the images to match exactly. Once you've recompiled and rerun with that new hue, run the following command:

```
$
```

```
compare out.png out_01.png comparison.png
```

Differences between your image and the expected image will be highlighted in red in `comparison.png`. If you're on EWS, you can open that image like you did before:

```
$
```

```
xdg-open comparison.png &
```

If you're working on your own machine, you can open `comparison.png` using the photo viewer of your choice.

Autograder Testing

You can run a subset of the test cases that will be used in the autograder with the following commands:

```
$
```

```
make test
./test
```

Submitting Your Work

The following files are used in grading:

- `sketchify.cpp`

All other files including any testing files you have added will not be used for grading.

Guide: How to submit CS 225 work using git

Additional Resources

- Pointers refresher