Bugs, and Debugging

For all of your projects up to now, you've likely been writing code, running it, and then viewing the output. This is a fine way to write smaller, toy projects, but as you progress, your projects will become more and more complex. Printing out variables to debug can be useful, however, it is inefficient, and it's not always 100% clear as to what's happening!

Approaches to Developing Software

Up to now, you've probably been writing your projects by thinking of a given problem, coding it out, and then trying to fix compiler issues and bugs after the fact. The software development process is significantly different than just "writing code". Writing code is just one portion of how we develop our software. There are a number of differing mentalities on how to write software, but they all ultimately boil down to three steps:

- Planning
- Implementation
- Testing/Debugging

Let's break these down a little further:

Planning:

1. **Read the requirements:** Requirement gathering is crucial. Be sure to read the prompts for your projects to the letter. Ensure that you're marking requirements as you're reading. It's not always the case that you'll be given a bulleted list of what to do in school, and even less so in an industry setting.
2. **Consider Input/Output** Users of programs, both software developers and non, often times either mis-type or mis-click. Be sure you have your types set up (if you're using a typed language) to match your inputs, but also be sure to gracefully fail if a user doesn't input proper types, or be sure to throw an error if you specifically want the program to crash at that exact moment. inputs happen constantly.
3. **Draw It Out** Creating a visual representation of a program can be incredibly helpful when trying to conceive of how to code something out. It could be anything from a flow chart to just some basic pseudocode, but so long as you know where you're going and what you want to do, you may realize that there could be some pitfalls that you hadn't initially expected!
4. **Break it Down** Once you've broken your project down, what can you tackle in small, easily testable steps? By planning ahead, you'll avoid the pitfalls of having to go backwards and delete much of your hard work!

Coding

1. **Small Iterative Steps** Instead of taking your chunks that you've drawn out and then starting to code all of it, write small pieces, and then test them! By doing this, you can be sure that your code works the ways that you've intended as you're coding (instead of going back and attempting to tape it together).
2. **Use Proper Formatting** It's not always the case that you'll have an editor that does the indentations for you, but if you're writing in a language that doesn't enforce indentation rules, then misaligned code can be incredibly confusing!
3. **Proofread Your Code** There are times when you'll run your code and wonder why something's working "even though you coded it to do something else". Computers are wonderful machines, but they only follow the directions we give them.
4. **Refactor** Your first attempt at writing a function or code block will always be messy. Once you get your code working, go back and try to find ways to make it cleaner so that it's more streamlined. Often times, refactoring results in the cutting out of a ton of similar code to be put into a function.
5. **Version Control** Using version control is a must. Once you get a piece of code working, you should immediately commit it. By doing so, you guarantee that no matter what you do (short of deleting the entire program), you can always go back and check out your old code.
6. **Don't Leave Compiler Warnings** Compiler (or interpreter) warnings don't often result in errors on the smaller scale, however, the larger your projects get, the problems that the compiler is warning you about can grow exponentially, resulting in your program crashing!

Common Mistakes During Coding:

Some of the most common mistakes during coding out your implementation are some of the easiest to get in the habit of catching!

- **Semicolons**: Be extra aware of semicolons. In compiled languages that require semicolons, you're likely hyper aware of them and are making sure to include semicolons after every statement, however, sometimes that can come back to bite you. Try running the following code: `hello.c`

  ```c
  #include <stdio.h>

  int main(int argc, char** argv){

    if (1 < 0);
    {
      printf("Hello! Math doesn't work!\n");
    }
  }
  ```

  or

```c
#include <stdio.h>

int main(int argc, char** argv){

  int i = -1;
  while ( i < 0);
  {
    printf("Will you ever reach me?!");
    i++;
  }
}
```

In the first example, it doesn't matter whether or not the argument inside the if is true. The semicolon after the statement says "this if is done, move on", and you then move into the curly braces (which at this point are acting like a code block).

The second example is much more frustrating, because it's not that you're accessing the code block that's frustrating, but you'll find yourself inside of an infinite loop since the semicolon is blocking the curly braces from being registered as the `while` loop's code block.

- **Uninitialized variables**: It's often tempting to declare a variable with no initialization. Variables are not automatically initialized to zero, though often times they will already be zero. Initialize your variables to absolutely ensure that you're not accidentally storing garbage data. There's no need to gamble!
- **Array Indices**: Arrays are used everywhere. To loop over arrays, you'll need to absolutely ensure that you're hitting all of your arrays' elements. Array indices are always from `0` to `length - 1`. A common mistake that happens is using a counter such as:

```c
...
for (int i; i <= LENGTH; i++) {
  // array accessing here
}
...
```

  By declaring your index as stopping at the SIZE, you're attempting to access a memory location outside of the array. It's not always the case that this is going to cause your program to crash, but it can be an incredibly frustrating bug to attempt to track down! The correct implementation is replace the `<=` with a `<`.
- **Bad Types**: Make sure to be as explicit as you can with your types.

Testing/Debugging

1. **Write Tests** While writing unit tests (i.e. functions that call other functions to see if they behave as they should) can seem like a huge pain (and unnecessary work), it automates you having to go back and test the code by hand. Additionally, tests can often catch simple mistakes that you may have overlooked in coding out the material. There are differing movements on the best ways of writing tests (e.g. post hoc unit testing vs test driven development), the point isn't to get hung up on how you write tests, but that you just write tests at all. Many failures in production software occur because tests were not written (or were not maintained).

   It is highly recommended to write your tests in a separate file, however, for now, just try to get into the habit of writing tests for you code at the bottom of your main file. Consider the program: `doMath.c`

   ```c
   #include <stdio.h>
   #include <stdlib.h>

   int addition(int firstNum, int secondNum){
     return firstNum + secondNum;
   }

   int main(int argc, char** argv){

     if (argc != 3) {
       fprintf(stderr, "Bad Input!");
       return EXIT_FAILURE;
     }

     int num1 = atoi(argv[1]);
     int num2 = atoi(argv[2]);

     int result = addition(num1, num2);

     printf("%d\n", result);
     return EXIT_SUCCESS;
   }
   ```

   How would you write a test for the function "addition". A simple implementation of a test would be something along the lines of:

   ```c
   #include <stdio.h>
   #include <stdlib.h>

   #define TEST 1


   int addition(int firstNum, int secondNum){
   ```

```c
  return firstNum + secondNum;
}

void testAddition() {
  int firstNumber = 10;
  int secondNumber = 20;
  int expectedOutput = 30;

  int response = addition(firstNumber, secondNumber);

  if (expectedOutput != response) {
    fprintf(stderr, "testAddition has failed! actual: %d  expected %d", response, expectedOutput);
    exit(1);
  }

  return;
}

void runTests() {
  testAddition();
}

int main(int argc, char** argv){

  if (TEST) {
    runTests();
    printf("ALL TESTS PASS!");
    return;
  }

  if (argc != 3) {
    fprintf(stderr, "Bad Input!");
    return EXIT_FAILURE;
  }

  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);

  int result = addition(num1, num2);

  printf("%d\n", result);
  return EXIT_SUCCESS;
}

```

In the above, we're running the tests because we've defined the constant `TEST` as 1, which then runs the function `runTests`. This function can be a place for you to put all of your tests so that you're not clogging up your main function! From there, we have only one test, where we have two inputs, and an expected output, which we then check against the actual response. If the test fails, we want to say which test failed.

2. **Manually Test** Automated unit tests of your program will save you a ton of development time, however, unit tests only test what we ask them to. While you can attempt to predict what will happen in your code, something will always come up to break it! By you going back and intentionally trying to break your code, you'll be able to catch even more bugs in the process! By doing this, you can go back, fix the issue, and even write another unit test to ensure that you don't make that same mistake twice!

   Supposing we were trying to break the tests by inputting different values. What if you tried subtraction? What if you tried floats? What happened?

3. **Use the Debugger**

   When you catch a bug in your program, or you think your program is not behaving as you had expected, it's easy to just write print statements to see what's happening to your variables. This can work for smaller programs, but as your programs get larger, do you really want to write a print statement after every single time you change a value, or receive a new value from a function? This is why there are debuggers! They let you step through the code, line by line, as well as view what is stored in each variable.

   The GNU debugger is what allows for us to debug C programming. However, because C is a compiled language, we'll need to add a special flag when we compile. GDB allows for you to interactively debug a program's source code. Let's try this with our code `doMath.c`:

   ```
   gcc -ggdb doMath.c
   ```

When running GDB, there are a number of commands, but the simplest are:

- `break #`: Where you define a specific line number to break on.
- `break functionName`: Where you define a specific function on which to break.
- `display expression`: Displays the C expression at the breakpoint.
- `print expression`: Displays the value of an expression
- `whatis expression`: Displays the type of an expression
- `list startingLineNum endingLineNum`: Where GDB will list the code and its line numbers. No provided arguments will list the program in 10 line increments.
- `bt`: Where GDB displays the call stack.
- `cont`: Continues execution for the program.
- `kill`: Stops execution for the program.
- `step`: Steps line by line through the code, even down into function calls.
- `next`: Steps line by line through the code, but not into functions.

Now, let's run our code:

```
gdb a.out
```

> GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
>
> Copyright (C) 2013 Free Software Foundation, Inc.
>
> License GPLv3+: GNU GPL version 3 or later http://gnu.org/licenses/gpl.html
>
> This is free software: you are free to change and redistribute it.
>
> There is NO WARRANTY, to the extent permitted by law. Type "show copying"
>
> and "show warranty" for details.
>
> This GDB was configured as "x86_64-redhat-linux-gnu".
>
> For bug reporting instructions, please see:
>
> http://www.gnu.org/software/gdb/bugs/...
>
> Reading symbols from /home/mjlny2/2750Materials/module7/gdb/a.out...done.

Set the breakpoint at Addition:

```
(gdb) break addition
```

> Breakpoint 1 at 0x400607: file math.c, line 8.

Run the Debugger

```
(gdb) run
```

> Starting program: /home/mjlny2/2750Materials/module7/gdb/./a.out
>
> Breakpoint 1, addition (firstNum=10, secondNum=21) at math.c:8 8 return firstNum + secondNum; 1: addition = {int (int, int)} 0x4005fd < addition > Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x86_64

Display the Breakpoint's Location

```
(gdb) display
```

> 1: addition = {int (int, int)} 0x4005fd < addition >

Print The Variables!

```
(gdb) print firstNum
```

> $1 = 10

```
(gdb) print secondNum
```

> $2 = 20

Print the Stacktrace:

```
(gdb) bt
```

> #0 addition (firstNum=10, secondNum=20) at math.c:9
>
> #1 0x000000000040063d in testAddition () at math.c:16
>
> #2 0x0000000000400682 in runTests () at math.c:27
>
> #3 0x000000000040069d in main (argc=1, argv=0x7fffffffe308) at math.c:33

We're still clearly in test mode, even if we forgot when we compiled the program for debugging! We can tell that by looking at our stack! The very bottom is our `main` function (which is the first to go onto the stack), then comes our `runTests` function, which then calls `testAddition` which sets up our our test environment for the function `addition` which is on the top of the stack!

Stepping

To step through the program, type

```
(gdb) step
```

This will step you line by line through your entire program! Try stepping a few times and calling the stack! What does it show? You should see the functions popping off of the stack one by one as you go along!

Kill the Program

```
(gdb) kill
```

> Kill the program being debugged? (y or n) y

Quit the Session

```
(gdb) quit
```

It's always a good idea to kill the program before you quit gdb!