How to Debug, a Quick Overview of Debugging

Honors Contract Class Project

Vincent Climer

University of Central Arkansas

Spring 2023

Introduction:

Computers are not as smart as many people think they are. Until AI, a computer could not guess what you were saying or typing. For a computer to understand your instructions, the instructions must be formatted in a way that the computer can understand. The only problem is that the format that the computer uses has many rules and there is no way for a programmer to write an entire program perfectly every time.

This is why we have debugging tools and software. Debugging tools help a programmer know what went wrong and how to fix it. Every program will break and have problems, and it's up to a programmer to figure out how to fix it.

In this paper, we will go over many different debugging software and techniques of debugging. Odds are, you are already using some of these forms of debugging. There is no best style of debugging, only different situations that they are best suited for. It would be best for all programmers to know these styles of debugging.

While we will be talking about debugging, there will be some examples that will help you better understand each style of debugging. This GitHub will contain all the programs that we will be debugging, or you could also just copy them as we go!

GitHub: https://github.com/Origamiboss/Dubugging-Samples.git

Rubber Ducky Debugging

Introduction

Rubber Ducky Debugging isn't a software or tool, but a way of checking your program by yourself. This is one of the first forms of debugging that has been used since the beginning of programming. This helps you understand your program better and makes sure that you're not getting distracted in your program. When we get a big project to work on, especially if we do it with other people, we can often get sidetracked with other goals rather than the goal of the program. We may end up focusing so much on one aspect of the program that we completely miss another part. That's where this debugging style comes in handy.

Instructions

A brief overview of this exercise is that you are explaining what the program is and how it functions to someone else, but if you don't want to bother anyone else. Who are you going to explain your program too? Well, why not a rubber duck? This exercise got its name because a group of programmers were explaining their programs to a bunch of rubber ducks they had sitting at their desk, and it helped them realize what the problem was with their program. Of course, if you can't get a rubber duck then you might be a little stuck. Anything would work though, my favorite alternative being Cyber Duck (an online program that emulates the helpful nature of the duck).

A few points about this technique:

1. You'll have to go line by line down the program explaining it out loud.
2. The explanation of the program needs to be simplified so that a layman can understand it.
3. You just move down through your code explaining line by line.

Example

If you want to give this a shot, you can copy the code from here or go to the GitHub repository.

A video explaining how to do this style of debugging can be found here:

https://youtu.be/UUQQNGKqrvs

This is the code we will be running in your main function:

```
//This is the Rubber Ducky Debugging Main Code
/*
Vincent Climer
GitHub: https://github.com/Origamiboss/Dubugging-Samples.git

The purpose of this code is to get input from the user into three integers x, y,
and z.
We will then calculate the equation x+y*z and display the results in comparison to
50.
The problem here is that the wrong number is being calculated.
Use rubber ducky debugging to solve the problem!
*/

#include <iostream>
using namespace std;

int main() {
```

```
	int overallInt;
	int x;
	int y;
	int z;
	//Gather input
	cout << "Input Number X: ";
	cin >> x;
	cout << "Input Number Y: ";
	cin >> y;
	cout << "Input Number Z: ";
	cin >> z;
	//Do Expression x+y*z
	x *= 7;
	overallInt = x + y * z;
	//See if its over 50
	if (overallInt > 50) {
		cout << "The Number: " << overallInt << " is greater than 50" <<
endl;
	}
	else if (overallInt < 50) {
		cout << "The Number: " << overallInt << " is greater lesss than 50"
<< endl;
	}
	else {
		cout << "The Number: " << overallInt << " equals 50" << endl;
	}
}
```

Conclusion

This is one of the more common forms of debugging. Running your problem step by step is essential for figuring out why a program is not working. In this program, a test value probably got left behind before a programmer added in the input interface, which threw everything off. Similar things may happen in your program, so developing a habit of walking through your program is essential! Just remember to keep a little ducky at your side so you can have someone cheering you on!

Resources (Here are extra resources if you want to look into this more!)

Rubber Duck Debugging – Rubber Duck Debugging – Debugging software with a rubber ducky

Cyber Duck Resource: Rubber Duck Debugging – Debugging software with a rubber ducky

Rubber Duck Debugging - How to Solve a Problem: https://www.youtube.com/watch?v=NBgIHOrjSxs

Talk to Your Neighbor

Talking to your neighbor is exactly what the name of the skill implies, seek help from an outside source. Sometimes, your own debugging skills and your own brain aren't enough to come up with the solution to the problems with your code, so you must seek help from someone else. It's typically advised to use Rubber Ducky Debugging first, so when you go talk to your neighbor, you at least know how your program functions in a way that you can describe it to your friend. That way your friend won't have to spend a lot of time trying to understand what your program does.

Example:

For this example, we'll be using the same code steps that we did with Rubber Ducky Debugging:

```
//This is the Rubber Ducky Debugging Main Code
/*
Vincent Climer
GitHub: https://github.com/Origamiboss/Dubugging-Samples.git

The purpose of this code is to get input from the user into three integers x, y,
and z.
We will then calculate the equation x+y*z and display the results in comparison to
50.
The problem here is that the wrong number is being calculated.
Use rubber ducky debugging to solve the problem!
*/

#include <iostream>
using namespace std;

int main() {
	int overallInt;
	int x;
	int y;
	int z;
	//Gather input
	cout << "Input Number X: ";
	cin >> x;
	cout << "Input Number Y: ";
	cin >> y;
	cout << "Input Number Z: ";
	cin >> z;
	//Do Expression x+y*z
	x *= 7;
	overallInt = x + y * z;
	//See if its over 50
	if (overallInt > 50) {
		cout << "The Number: " << overallInt << " is greater than 50" <<
endl;
	}
	else if (overallInt < 50) {
		cout << "The Number: " << overallInt << " is greater lesss than 50"
<< endl;
	}
	else {
		cout << "The Number: " << overallInt << " equals 50" << endl;
	}
}
```

Since you already know how this program works, let’s go take it to our poor neighbor who will now try and figure out what’s wrong with it.

Visual Studio Debugger:

The Visual Studio Debugger is the best tool for any Visual Studio programmer. There are many different tools that the Visual Studio Debugger gives us, and we'll just go over some of the main ones here: the Output window and Breakpoints.

Instructions:

Build Errors:

1. Find the little red circle on top of your output window.
2. Click on it.
3. You can now see the build errors and where they are located.

Breakpoints:

1. Move your mouse over the vertical bar, left of the line numbers.
2. Once a white circle appears, click. The red dot that appears is a breakpoint.
3. When you run the program, the program will stop where the red dot is.
4. You can navigate through the program with the breakpoint tools in the top right of the program.
5. You can see variable values in the lower left of the screen where the output used to be.

Example:

The Visual Studio debugger has many tools, but we will mainly be talking about two of them, Build Errors and Breakpoints.

For Build Errors, here is a video explaining it:

https://youtu.be/4wqqzIprWLg

Here is a video to explain breakpoints from Visual Studio:

https://youtu.be/YZ5Mqa3HNf8

Here is our example code!

```
/*This is the visual studio example code

We will be looking at the breakpoints, output and many other tools that visual
studio provides in this program.
In this program we will be making a guessing game. The user will guess a number
between 1-100 and the program will tell them if
they are getting closer or not. Needless to say, there are some errors in it.
*/

#include <iostream>
#include <Math.h>
using namespace std

//declaring functions
int randomNumber(int);
string checkNumber(int, int);

int main() {
	char goAgain = 'y';
```

```
	//get a random number
	int ran = randomNumber(99);
	//check if user wants to quit
	while (goAgain == 'y' || goAgain == 'Y') {
		int guess;
		cout << "Guess a number between 1 and 100: ";
		cin >> guess;
		string response = checkNumber(guess, ran);
		cout << response;
		//reset game
		if (response == "You Got the Answer Right!\n") {
			cout << "Do you want to play again (y=yes|n=no)? ";
			cin >> goAgain;
			ran = randomNumber(100);
		}
	}
}
int randomNumber(int max) {
	int ran;
	ran = rand() / max;
	return ran;
}
string checkNumber(int num, int ran) {
	string returnstring;
	if (num > ran) {
		//the guess is greater than the random number
		return "Lower\n";
	}
	else if(num < ran) {
		return "Higher\n";
	}
	else {
		return "You Got the Answer Right!\n";
	}
}
```

Resources:

Using Breakpoints in Visual Studio - [Use breakpoints in the debugger - Visual Studio (Windows) | Microsoft Learn]

Print:

Sometimes when you are debugging your program, you want to make sure that your program is running properly, but you don't want to stop your program with breakpoints. With our last example, we gained a lot of information by using our breakpoints, but it took a long time, clicking through the whole program. As your program gets bigger and bigger, it will become harder and harder to click through the whole program with breakpoints. Printing is a solution to that problem. By logging different parts of the program to our console, we can tell what's going on with the program.

Instructions:

1. Wherever you think the problem in the program is, put a print statement there.
2. If you need to know a variables value, you have the print statement print it.
3. You tweak the code until the print code prints what it needs to print.

Example:

Here is a video explain the print style of debugging:

https://youtu.be/sMkenX7gK6I

Here is our code!

```
/*
Vincent Climer
GitHub: https://github.com/Origamiboss/Dubugging-Samples.git

This program is broken in such a way so that the debugger can use print functions
to figure out what's wrong.
The game is supposed to be a rock paper Scissors game where you can choose to play
against a computer or a person.
*/
#include<iostream>
#include <math.h>
using namespace std;
/*Rock Paper Scissors Game*/
string calculateResult(int, int);
int getPlayerInput(int);
int computerChoice();

int main() {
	char again = 'y';
	//game loop
	while (again == 'y' || again == 'Y') {
		int porc;
		cout << "Play against a player or computer (1 | 2): ";
		cin >> porc;
		if (porc == 1) {
			//player vs player
			int p1 = getPlayerInput(1);
			int p2 = getPlayerInput(2);
			cout << calculateResult(p1, p2);
		}
		else {
			//player vs computer
			int p1 = getPlayerInput(1);
			int c2 = computerChoice();
			cout << calculateResult(p1, c2);
```

```
		}



		cout << "Play again? (y for yes | n for no): ";
		cin >> again;
		cout << endl;
	};

}
//calculates the computers choice
string calculateResult(int c1, int c2) {
	string result;
	//generate result
	if ((c1 == 1 && c2 == 2) || (c1 == 2 && c2 == 1)) {
		result = "Rock beats Scissors, Player ";
		if (c1 == 1) result += "1";
		else result += "2";
		result += " wins\n";
	}
	else if ((c1 == 2 && c2 == 3) || (c1 == 3 && c2 == 2)) {
		result = "Scissors  beat Paper, Player ";
		if (c1 == 2) result += "1";
		else result += "2";
		result += " wins\n";
	}
	else if ((c1 == 3 && c2 == 1) || (c1 == 1 && c2 == 3)) {
		result = "Paper beats Rock, Player ";
		if (c1 == 2) result += "1";
		else result += "2";
		result += " wins\n";
	}
	else {
		result = "Both players have chosen ";
		switch (c1) {
		case 1:
			result += "Rock";

		case 2:
			result += "Scissors";

		case 3:
			result += "Paper";

		default:
			result = "ERROR";

		}
		result += "\n";
	}
	return result;
}
//get Player Input
int getPlayerInput(int player) {
	//reset console screen
	for (int i = 0; i < 40; i++) {
		cout << endl;
	}
	cout << "Choose a item, 1- Rock, 2-Scissors , 3-Paper\n";
	cout << "Player " << player << ": ";
```

```
	int p;
	cin >> p;
	return 0;

}
//calculate the computers choice
int computerChoice() {
	int c;
	c = rand() % 3;
	cout << "Computer: " << c << endl;
	return c;
}
```

Cut In Half:

This style of debugging is good if there is an error somewhere in your program, and you don't know where it is. If there is an infinite loop somewhere in your program or something that breaks your program, you can comment it out to figure out exactly what is wrong.

Instructions:

1. Comment out the code that you think the problem may be in.
2. One by one, uncomment parts of the code and run the program.
3. If the program runs correctly then you know that part of the code is okay, otherwise you know that the code you uncommented is broken.

Example:

Here is a video explain the cut in half method of debugging:

https://youtu.be/Th2fM7w3l9Q

Here is our code!

```
/*
Vincent Climer
GitHub: https://github.com/Origamiboss/Dubugging-Samples.git

The purpose of this program is to show off a debugging style known as cut the
problem in half.
Often times, your program can get so large, you have no idea where the problem in
the problem lies.
The purpose of this program is to find all of the factors of a number;
*/
#include <iostream>
using namespace std;
//reset array
void resetArray(int*, int);
//get the first number
int getNumber();
//find the numbers
void generateFactors(int*, int);
//print the numbers
void printNumber(int*);
//check if you need to go again
char goAgainCheck();

int main() {
	char goAgain;
	const int CAPACITY = 100;
	do {
		int num = getNumber();
		int* ar  = new int[CAPACITY];
		resetArray(ar, CAPACITY);
		generateFactors(ar, num);
		printNumber(ar);
		goAgain = goAgainCheck();
		delete[] ar;
	} while (goAgain == 'y' || goAgain == 'Y');
}
//reset array
```

```
void resetArray(int* ar, int cap){
	for (int i = 0; i < cap; i++) ar[i] = 0;
}
//get the first number
int getNumber() {
	cout << "Number: ";
	int n;
	cin >> n;
	return n;
}
//find the numbers
void generateFactors(int* ar, int num) {
	int temp = num;
	int temp2 = 1;
	int s = 0;
	while (temp2 < num / 2) {
		if (temp2 * temp == num) {
			ar[s] = temp2;
			ar[s+1] = temp;
			s+=2;
		}
		if (temp >= num) {
			temp2++;
			temp = 0;
		}
		temp+1;
	}
}
//print the numbers
void printNumber(int* ar) {
	int s = 0;
	while (ar[s] != 0) {
		cout << ar[s] << ", " << ar[s+1];
		cout << endl;
		s+=2;
	}
}
//check if you need to go again
char goAgainCheck() {
	char a;
	cout << "Use program again? (y = yes | n = no): ";
	cin >> a;
	return a;
}
```

Conclusion:

Learning how to debug programs is essential for all programmers. Programs will not always work, and programmers are expected to fix them. The debugging styles depicted in this document are not the only ones that exist. There are many ways to find out what's wrong with a program, and it's up to the programmer to choose how to fix it.

References:

Rubber Duck Debugging – Rubber Duck Debugging – Debugging software with a rubber ducky

Using Breakpoints in Visual Studio - Use breakpoints in the debugger - Visual Studio (Windows) | Microsoft Learn