3.4 — Basic debugging tactics

In the previous lesson, we explored a strategy for finding issues by running our programs and using guesswork to home in on where the problem is. In this lesson, we'll explore some basic tactics for actually making those guesses and collecting information to help find issues.

Debugging tactic #1: Commenting out your code

Let's start with an easy one. If your program is exhibiting erroneous behavior, one way to reduce the amount of code you have to search through is to comment some code out and see if the issue persists. If the issue remains unchanged, the commented out code probably wasn't responsible.

Consider the following code:

```
int main()
{
    getNames(); // ask user to enter a bunch of names
    doMaintenance(); // do some random stuff
    sortNames(); // sort them in alphabetical order
    printNames(); // print the sorted list of names

    return 0;
}
```

Let's say this program is supposed to print the names the user enters in alphabetical order, but its printing them in reverse alphabetical order. Where's the problem? Is *getNames* entering the names incorrectly? Is *sortNames* sorting them backwards? Is *printNames* printing them backwards? It could be any of those things. But we might suspect doMaintenance() has nothing to do with the problem, so let's comment it out.

```
int main()
{
    getNames(); // ask user to enter a bunch of names
//    doMaintenance(); // do some random stuff
    sortNames(); // sort them in alphabetical order
    printNames(); // print the sorted list of names

    return 0;
}
```

There are three likely outcomes:

- If the problem goes away, then *doMaintenance* must be causing the problem, and we should focus our attention there.

- If the problem is unchanged (which is more likely), then we can reasonably assume that *doMaintenance* wasn't at fault, and we can exclude the entire function from our search for now. This doesn't help us understand whether the actual problem is before or after the call to *doMaintenance*, but it reduces the amount of code we have to subsequently look through.
- If commenting out *doMaintenance* causes the problem to morph into some other related problem (e.g. the program stops printing names), then it's likely that *doMaintenance* is doing something useful that some other code is dependent on. In this case, we probably can't tell whether the issue is in *doMaintenance* or elsewhere, so we can uncomment *doMaintenance* and try some other approach.

Warning

Don't forget which functions you've commented out so you can uncomment them later!

After making many debugging-related changes, it's very easy to miss undoing one or two. If that happens, you'll end up fixing one bug but introducing others!

Having a good version control system is extremely useful here, as you can diff your code against the main branch to see all the debugging-related changes you've made (and ensure that they are reverted before you commit your change).

Tip

An alternate approach to repeatedly adding/removing or uncommenting/commenting debug statements is to use a 3rd party library that will let you leave debug statements in your code but compile them out in release mode via a preprocessor macro. dbg is one such header-only library that exists to help facilitate this (via the `DBG_MACRO_DISABLE` preprocessor macro).

We discuss header-only libraries in lesson 5.7 -- Inline functions and variables.

Debugging tactic #2: Validating your code flow

Another problem common in more complex programs is that the program is calling a function too many or too few times (including not at all).

In such cases, it can be helpful to place statements at the top of your functions to print the function's name. That way, when the program runs, you can see which functions are getting called.

Tip

When printing information for debugging purposes, use std::cerr instead of std::cout. One reason for this is that std::cout may be buffered, which means there may be a pause between when you ask std::cout to output information and when it actually does. If you output using std::cout and then your program crashes immediately afterward, std::cout may or may not have actually output yet. This can mislead you about where the issue is. On the other hand, std::cerr is unbuffered, which means anything you send to it will output immediately. This helps ensure all debug output appears as soon as possible (at the cost of some performance, which we usually don't care about when debugging).

Using std::cerr also helps make clear that the information being output is for an error case rather than a normal case.

We discuss when to use std::cout vs std::cerr further in lesson 9.4 -- Detecting and handling errors.

Consider the following simple program that doesn't work correctly:

```
#include <iostream>

int getValue()
{
        return 4;
}

int main()
{
    std::cout << getValue << '\n';

    return 0;
}
```

You may need to disable "Treat warnings as errors" for the above to compile.

Although we expect this program to print the value *4*, it should print the value:

```
1
```

On Visual Studio (and possibly some other compilers), it may print the following instead:

```
00101424
```

Related content

We discuss why some compilers print `1` vs an address (and what to do if your compiler prints `1` but you want it to print an address) in lesson 20.1 -- Function Pointers.

Let's add some debugging statements to these functions:

```
#include <iostream>

int getValue()
{
std::cerr << "getValue() called\n";
        return 4;
}

int main()
{
std::cerr << "main() called\n";
    std::cout << getValue << '\n';

    return 0;
}
```

Tip

When adding temporary debug statements, it can be helpful to not indent them. This makes them easier to find for removal later.

If you are using clang-format to format your code, it will try to auto-indent these lines. You can suppress the automatic formatting like this:

```
// clang-format off
std::cerr << "main() called\n";
// clang-format on
```

Now when these functions execute, they'll output their names, indicating that they were called:

```
main() called
1
```

Now we can see that function *getValue* was never called. There must be some problem with the code that calls the function. Let's take a closer look at that line:

```
std::cout << getValue << '\n';
```

Oh, look, we forgot the parenthesis on the function call. It should be:

```
#include <iostream>

int getValue()
{
std::cerr << "getValue() called\n";
        return 4;
}

int main()
{
std::cerr << "main() called\n";
    std::cout << getValue() << '\n'; // added parenthesis here

    return 0;
}
```

This will now produce the correct output

```
main() called
getValue() called
4
```

And we can remove the temporary debugging statements.

Debugging tactic #3: Printing values

With some types of bugs, the program may be calculating or passing the wrong value.

We can also output the value of variables (including parameters) or expressions to ensure that they are correct.

Consider the following program that is supposed to add two numbers but doesn’t work correctly:

```
#include <iostream>

int add(int x, int y)
{
	return x + y;
}

void printResult(int z)
{
	std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
	std::cout << "Enter a number: ";
	int x{};
	std::cin >> x;
	return x;
}

int main()
{
	int x{ getUserInput() };
	int y{ getUserInput() };

	std::cout << x << " + " << y << '\n';

	int z{ add(x, 5) };
	printResult(z);

	return 0;
}
```

Here's some output from this program:

```
Enter a number: 4
Enter a number: 3
4 + 3
The answer is: 9
```

That's not right. Do you see the error? Even in this short program, it can be hard to spot. Let's add some code to debug our values:

```
#include <iostream>

int add(int x, int y)
{
        return x + y;
}

void printResult(int z)
{
        std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
        std::cout << "Enter a number: ";
        int x{};
        std::cin >> x;
        return x;
}

int main()
{
        int x{ getUserInput() };
std::cerr << "main::x = " << x << '\n';
        int y{ getUserInput() };
std::cerr << "main::y = " << y << '\n';

        std::cout << x << " + " << y << '\n';

        int z{ add(x, 5) };
std::cerr << "main::z = " << z << '\n';
        printResult(z);

        return 0;
}
```

Here's the above output:

```
Enter a number: 4
main::x = 4
Enter a number: 3
main::y = 3
4 + 3
main::z = 9
The answer is: 9
```

Variables *x* and *y* are getting the right values, but variable *z* isn't. The issue must be between those two points, which makes function *add* a key suspect.

Let's modify function add:

```
#include <iostream>

int add(int x, int y)
{
std::cerr << "add() called (x=" << x <<", y=" << y << ")\n";
        return x + y;
}

void printResult(int z)
{
        std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
        std::cout << "Enter a number: ";
        int x{};
        std::cin >> x;
        return x;
}

int main()
{
        int x{ getUserInput() };
std::cerr << "main::x = " << x << '\n';
        int y{ getUserInput() };
std::cerr << "main::y = " << y << '\n';

        std::cout << x << " + " << y << '\n';

        int z{ add(x, 5) };
std::cerr << "main::z = " << z << '\n';
        printResult(z);

        return 0;
}
```

Now we'll get the output:

```
Enter a number: 4
main::x = 4
Enter a number: 3
main::y = 3
add() called (x=4, y=5)
main::z = 9
The answer is: 9
```

Variable *y* had value 3, but somehow our function *add* got the value 5 for parameter *y*. We must have passed the wrong argument. Sure enough:

```
int z{ add(x, 5) };
```

There it is. We passed the literal *5* instead of the value of variable *y* as an argument. That's an easy fix, and then we can remove the debug statements.

One more example

This program is very similar to the prior one, but also doesn't work like it should:

```
#include <iostream>

int add(int x, int y)
{
	return x + y;
}

void printResult(int z)
{
	std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
	std::cout << "Enter a number: ";
	int x{};
	std::cin >> x;
	return --x;
}

int main()
{
	int x{ getUserInput() };
	int y{ getUserInput() };

	int z { add(x, y) };
	printResult(z);

	return 0;
}
```

If we run this code and see the following:

```
Enter a number: 4
Enter a number: 3
The answer is: 5
```

Hmmm, something is wrong. But where?

Let's instrument this code with some debugging:

```
#include <iostream>

int add(int x, int y)
{
std::cerr << "add() called (x=" << x << ", y=" << y << ")\n";
	return x + y;
}

void printResult(int z)
{
std::cerr << "printResult() called (z=" << z << ")\n";
	std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
std::cerr << "getUserInput() called\n";
	std::cout << "Enter a number: ";
	int x{};
	std::cin >> x;
	return --x;
}

int main()
{
std::cerr << "main() called\n";
	int x{ getUserInput() };
std::cerr << "main::x = " << x << '\n';
	int y{ getUserInput() };
std::cerr << "main::y = " << y << '\n';

	int z{ add(x, y) };
std::cerr << "main::z = " << z << '\n';
	printResult(z);

	return 0;
}
```

Now let's run the program again with the same inputs:

```
main() called
getUserInput() called
Enter a number: 4
main::x = 3
getUserInput() called
Enter a number: 3
main::y = 2
add() called (x=3, y=2)
main::z = 5
printResult() called (z=5)
The answer is: 5
```

Now we can immediately see something going wrong: The user is entering the value *4*, but main's *x* is getting value *3*. Something must be going wrong between where the user enters input and where that value is assigned to main's variable *x*. Let's make sure that the program is getting the correct value from the user by adding some debug code to function *getUserInput*:

```
#include <iostream>

int add(int x, int y)
{
std::cerr << "add() called (x=" << x << ", y=" << y << ")\n";
	return x + y;
}

void printResult(int z)
{
std::cerr << "printResult() called (z=" << z << ")\n";
	std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
std::cerr << "getUserInput() called\n";
	std::cout << "Enter a number: ";
	int x{};
	std::cin >> x;
std::cerr << "getUserInput::x = " << x << '\n'; // added this additional line of
debugging
	return --x;
}

int main()
{
std::cerr << "main() called\n";
	int x{ getUserInput() };
std::cerr << "main::x = " << x << '\n';
	int y{ getUserInput() };
std::cerr << "main::y = " << y << '\n';

	int z{ add(x, y) };
std::cerr << "main::z = " << z << '\n';
	printResult(z);

	return 0;
}
```

And the output:

```
main() called
getUserInput() called
Enter a number: 4
getUserInput::x = 4
main::x = 3
getUserInput() called
Enter a number: 3
getUserInput::x = 3
main::y = 2
add() called (x=3, y=2)
main::z = 5
printResult() called (z=5)
The answer is: 5
```

With this additional line of debugging, we can see that the user input is received correctly into getUserInput's variable *x*. And yet somehow main's variable *x* is getting the wrong value. The problem must be between those two points. The only culprit left is the return value from function *getUserInput*. Let's look at that line more closely.

```
return --x;
```

Hmmm, that's odd. What's that `--` symbol before x? We haven't covered that yet in these tutorials, so don't worry if you don't know what it means. But even without knowing what it means, through your debugging efforts, you can be reasonably sure that this particular line is at fault -- and thus, it's likely this `--` symbol is causing the problem.

Since we really want *getUserInput* to return just the value of *x*, let's remove the `--` and see what happens:

```
#include <iostream>

int add(int x, int y)
{
std::cerr << "add() called (x=" << x << ", y=" << y << ")\n";
        return x + y;
}

void printResult(int z)
{
std::cerr << "printResult() called (z=" << z << ")\n";
        std::cout << "The answer is: " << z << '\n';
}

int getUserInput()
{
std::cerr << "getUserInput() called\n";
        std::cout << "Enter a number: ";
        int x{};
        std::cin >> x;
std::cerr << "getUserInput::x = " << x << '\n';
        return x; // removed -- before x
}

int main()
{
std::cerr << "main() called\n";
        int x{ getUserInput() };
std::cerr << "main::x = " << x << '\n';
        int y{ getUserInput() };
std::cerr << "main::y = " << y << '\n';

        int z{ add(x, y) };
std::cerr << "main::z = " << z << '\n';
        printResult(z);

        return 0;
}
```

And now the output:

```
main() called
getUserInput() called
Enter a number: 4
getUserInput::x = 4
main::x = 4
getUserInput() called
Enter a number: 3
getUserInput::x = 3
main::y = 3
add() called (x=4, y=3)
main::z = 7
printResult() called (z=7)
The answer is: 7
```

The program is now working correctly. Even without understanding what `--` was doing, we were able to identify the specific line of code causing the issue, and then fix the issue.

Why using printing statements to debug isn't great

While adding debug statements to programs for diagnostic purposes is a common rudimentary technique, and a functional one (especially when a debugger is not available for some reason), it's not that great for a number of reasons:

1. Debug statements clutter your code.
2. Debug statements clutter the output of your program.
3. Debug statements require modification of your code to both add and to remove, which can introduce new bugs.
4. Debug statements must be removed after you're done with them, which makes them non-reusable.

We can do better. We'll explore how in future lessons.