How To Debug

Table of Contents

About these notes

These notes are about debugging small application software. They are written for the benefit of people starting to learn how to write software. There is a lot of study material about learning Java or Python or JavaScript and so on, but little about fixing problems - which are very common when people start to write software. This is what this is about.

It is not about

- Software testing - testing of all aspects of small and large systems
- Security - when someone is trying to make it go wrong
- Formal methods - including proofs that code is correct

It includes examples from several common languages, to try to show the basic common principles involved.

What is a bug?

As discussed here, there are two main classes of bugs;

1. Bugs shown when *making* software

2. Bugs when the software *runs*

For the first case, we are writing code which will be

- interpreted (JavaScript, PHP, Python and so on) or
- compiled to native code (C, C++) or
- compiled to an intermediate form(Java compiled to bytecode)

In all cases, we might get a 'syntax error' before we run the code. Syntax errors are discussed later.

In the second case, when we run the code, the result might be:

- A 'crash'. The application unexpectedly ends, maybe with a numeric error code. C will sometimes do this.

- An exception which is not caught. Java might do this.
- Output which is incorrect. This is when the actual output differs from the expected output - what it is supposed to do. By 'output' is meant generally - it might be to write a file, output a text message to a console, display something on a web page, write to a database and so on. Any language might do this.

The third possibility is the hardest to fix - a 'silent fail' with almost no clue as to why. There are some suggestions here.

JavaScript, usually embedded in an html webpage, is bad here. There are billions of pages on the web, and a very large number contain invalid html. But most browsers still display something - and usually not an error message, as a typical interpreter or compiler would. We usually need to look at the 'console' in the browser 'developer tools' to see any messages. See later.

KISS

KISS is advice. It stands for Keep It Simple, Stupid

If it 'won't work', you have to work out exactly *what* will not work. You might be using any of the following tools:

- A text editor to write code.
- An interpreter - maybe inside a web browser, for JavaScript
- A compiler - maybe Java or C or C++
- An application launcher - such as Java
- A linker - for C or C++
- A make utility, executing a makefile
- A database server, such as MySQL
- A database
- An http server, local or remote, like Apache. If its remote, you are using a whole bunch of software components - a network interface, a gateway, unknown routers and switches and so on.
- An IDE like NetBeans or Eclipse or CodeBlocks or IntelliJ.

Each of these tools might make things 'go wrong'. If you are unlucky, it might be a combination of several of them.

The best strategy is to use the minimum number of tools as possible - to keep things as simple as possible.

If that can be a simple text editor and a compiler, at the command line - use that. So

Learn to use the command line in a 'terminal'

An IDE will probably offer all these tools, probably for several different languages, and use multiple plug-ins.

Do not use an IDE unless you have some experience with the language.

Syntax errors

Suppose we want to run this program, in pseudo-code:

```
x=2
y=3
z=x+y
print z
```

and expect to get 5.

In Java, we write

```
public class Test
{
  public static void main(String[] args)
  {
    int y=3;
    int z=x+y;
    System.out.println(z);
  }
}
```

We try to compile it, and get:

```
walter@mint2 ~/Documents/temp $ javac Test.java
Test.java:6: error: cannot find symbol
  int z=x+y;
        ^
  symbol:   variable x
  location: class Test
1 error
```

So the *error message* is "cannot find symbol". What does that mean? What symbol? It tells us:

```
  symbol:   variable x
```

It literally points to the problem:

```
  int z=x+y;
        ^
```

and it tells us where that problem is;

```
Test.java:6:
```

So it is in the file Test.java, at line 6.

We just need to work out what this means. The problem is that we have not told the compiler anything about x. We have not declared it (as an int), and not given it a value. That means it is logically impossible to work out what x+y is.

So we need to add a line declaring x to be an int and assign it a value:

```
public class Test
{
  public static void main(String[] args)
  {
    int x=2;
    int y=3;
    int z=x+y;
    System.out.println(z);
  }
}
```

We save that, compile it, run it, and get the expected 5.

Suppose we do this in Python:

```
y=3;
z=x+y
print(z)
```

and when we try to run this, we get

```
Traceback (most recent call last):
  File "/home/walter/Documents/temp/test.py", line 3, in <module>
    z=x+y
NameError: name 'x' is not defined
Process terminated with an exit code of 1
```

Here the error message is

```
NameError: name 'x' is not defined
```

and it tells us where:

```
  File "/home/walter/Documents/temp/test.py", line 3, in <module>
    z=x+y
```

In Python we do not need to declare variables. But we must tell it a value for x if it is to work out x+y. So the simple fix is:

```
x=2
y=3
z=x+y
print(z)
```

which outputs 5 as expected.

Or in JavaScript embedded in a web page:

```
<!DOCTYPE html>
<html lang="en">
<script>

var y=3;
var z=x+y;
console.log(z);

</script>
<head>
</head>

<body>
</body>

</html>
```

When loaded into a browser, the console (in developer tools) says:

```
test.html:6 Uncaught ReferenceError: x is not defined
    at test.html:6
```

Here the error message is

```
x is not defined
```

at line 6 in file test.html.

So the fix is to define x:

```
<!DOCTYPE html>
```

```
<html lang="en">
<script>

var x=2;
var y=3;
var z=x+y;
console.log(z);

</script>
<head>
</head>

<body>
</body>

</html>
```

and we get 5 as expected.

What do we need to do?

Read the error message

Syntax errors are the simplest to fix. The compiler or interpreter tells you *what* the error is, and *where* it is.

A central aspect of language design is to have common bugs appear as syntax errrors. Then the system can find the bugs for you, and tell you what they are.

Misleading Error Messages

Suppose our code in C is

```
#include <stdio.h>

int main(void)
{
int x=2
int y=3;
int z=x+y;
printf("%d\n",z);

return 0;
}
```

When we compile this, we get:

```
walter@mint2 ~/Documents/temp $ gcc test.c -o p
test.c: In function ‘main’:
test.c:6:1: error: expected ‘,’ or ‘;’ before ‘int’
 int y=3;
 ^~~
test.c:7:9: error: ‘y’ undeclared (first use in this function)
 int z=x+y;
         ^
test.c:7:9: note: each undeclared identifier is reported only once for each
function it appears in
```

This says the problem is here:

```
 int y=3;
 ^~~
```

but we can see nothing wrong with that.

The clue is

```
error: expected ‘,’ or ‘;’ before ‘int’
```

The problem is the line *before* int y=3;

We missed out the semi-colon after x=2. The corrected code is:

```
#include <stdio.h>

int main(void)
{
int x=2;
int y=3;
int z=x+y;
printf("%d\n",z);

return 0;
}
```

Why?

Interpreters and compilers work by parsing source code - that is, comparing it with the grammar of the language to work out its parts, as statements, which kind of statemenst and so on. In the case of

```
int main(void)
{
int x=2
int y=3;
```

the code is correct up to the 2. Only when it reaches int y=3; without a semi-colon, is there any error. So this is what it reports.

It is quite common for the error to be before the line flagged.

Understanding theory

Suppose in Java we say:

```
public class Test extends String
{
  public static void main(String[] args)
  {
    int x=2;
    int y=3;
    int z=x+y;
    System.out.println(z);
  }
}
```

We get the compile-time error message:

```
walter@mint2 ~/Documents/temp $ javac Test.java
Test.java:1: error: cannot inherit from final String
public class Test extends String
                          ^
1 error
```

What does that mean? We need to understand 3 terms:

- inherit

- final
- extends

We need to know what inheritance is in OOP, that extends means a class is inheriting from another, that String is a final class, and that final means it cannot be sub-classed.

If we are coding with some paradigm such as OOP, we need to undertsand it, or error messages will make no sense.

Bugs which are not syntax errors

When we write code, we have an idea in our heads about what the code does, step by step, and that by the end, we will have the result which is required.

For a bug - that idea in our head is wrong. That is not what our code does.

We must find out what it *actually* does, see the difference from what we thought, and fix it.

Trace tables

A simple technique to see what it actually does it to run through the code on paper, drawing up a trace table.

Suppose we want to exchange the values of two variables. Call then x and y. The obvious way to exchange values is:

```
x=y // copy y to x, and
y=x // copy x to y
```

Try it out in Python:

```
x=2
y=3
print(x,y) # get 2 3
x=y
y=x
print(x,y) # get 3 3
```

and not 3 2, as expected.

How do we debug that? One technique is to write down a trace table on paper, in which we track, instruction by instruction, the values of variables. Here we go:

Step	x	y	Notes
Before we start	?	?	We do not know what x and y are
x=2	2	?	This is *after* we do x=2
y=3	2	3	
x=y	3	3	We have copied y (3) to x
y=x	3	3	We have copied x (3) to y

From the trace table, with a little thought, we can see the problem. When we said x=y, we copied y to x, and that over-wrote the old value of x, which was 2. We have lost the old x, so we cannot move it to y.

So, before we say x=y, we need space to keep a copy of the old value of x, to eventually be moved to y:

The third location is often named temp, for temporary:

```
x=2
y=3
print(x,y)
temp=x
x=y
y=temp
print(x,y)
```

Using a debugger

A debugger is software intended to help debug code They usually provide the ability to

- Execute statements one at a time
- Follow execution into sub-routines, or
- Step over sub-routine calls
- Run to cursor
- Run to next breakpoint = next instruction set as a 'breakpoint'
- Display selected 'watch variables'

and more. They often take longer to set up than finding a bug by other means.

Expected output - requirements specification

The first step in debugging is to know that there is a bug.

That requires having a *requirements specification* - which just means, knowing what the code is supposed to achieve.

By *achieve* we mean what the ouput should be. This is in a general sense - the output might go to the screen, be written to a file, produce a row in a database table, reduce the balance in a bank account, or whatever. We might ask what the *result* should be, or what the *effect* of the code is supposed to be.

This is different from what the software does - *how* it tries to achieve the required output. The software may execute the algorithm as expected - but the algorithm might be wrong. To sort things out, we first need to be clear what we actually want.

This is most clearly shown by having expected output. That is, what correct output is. We compare the actual output with expected output. If they differ, we have a bug. This is the first step on the road to wisdom.

Usually the expected output depends on the input. So we have a set of test input, expected output, and actual output.

For example, the requirement might be

Input an array of ints

Output an array with each element replaced by a count of how many elements after that element which are less than the element.

That is not very clear, but sets of expected outputs makes things clearer.

Input 6,8,2,0,8
Output 2,2,1,0,0

Why? For the 6, there are 2 elements after it less than 6, namely 2 and 0

Same for 8

For the third element, 2, there is just 1 smaller element - 0

And so on.

We would take this further for proper testing, trying to cover all possible situations. This is just an outline.

Optimisation

Optimisation means taking some process and improving it - here doing it faster or with less memory.

A simple algorithm for the problem in the last section is:

for each element, starting at the first..

..count how many elements after that are less

.. replace the element with the count.

Code in C to do that is:

```
#include <stdio.h>

int input[]={6,8,2,0,8}; // input test data

int main(void)
{
for (int start=0; start<5; start++)
{ // go from start to the end
  int count=0; //count ofelements less than start
  for(int where=start+1; where<5; where++)
  { // go from start to the end
    if (input[where]<input[start]) count++;
  }
  input[start]=count; // replace element by count
}
// check result
for (int start=0; start<5; start++)
  printf("%d ",input[start]);

  return 0;
}
```

For an array with n elements, we need n scans through, and each scan takes n-1, n-2,, n-3.. steps, which averages n/2. So this takes $n^2/2$ steps. So the time complexity is $O[n^2]$.

That is not good - it would often be too slow for more than a few thousand items. So we might try to optimise it - make it faster.

But the well-known *First Law of Optimisation* is - don't.

In all cases, to optimise means at least having one method which actally works. Try to get that, before trying to get something faster.

We might do this here using a data structure known as a heap. But let's not.

Thinking

This is a useful technique for getting evidence to be used to work out and fix a bug.

As an example we use a binary search. This is a standard algorithm to find a value in a *sorted* linear list. For example:

```
int data[]={2,3,6,8,9,12,13,15,17,19,20};
int target=9;
```

and we want to search data for the value target, and report where it is. So teh expected output here is 4, because 9 has index 4 in the array.

The idea is to have 2 pointers, initially to the start and end of the list. In our code we call these lo and hi. We work out the middle position, and look there. If we have a match, we have found it. If the middle is too big, it must be in the lower half, and we change hi to middle. Else it is in the upper half, and we change lo to middle. Then recursively call binSearch again:

```
#include <stdio.h>

int data[]={2,3,6,8,9,12,13,15,17,19,20};
int target=9;

void binSearch(int lo, int hi)
{
  int middle=(hi+lo)/2;
  if (data[middle]==target)
  {
    printf("Found at %d\n",middle);
    return;
  }
  if (data[middle]>target)
  {
    hi=middle;
  }
  else
  {
    lo=middle;
  }
  binSearch(lo, hi);

}

int main(void)
{
binSearch(0, 10);
return 0;
}
```

We compile it and run it, and get the expected output:

```
walter@mint2 ~/Documents/temp $ gcc test.c -o p
walter@mint2 ~/Documents/temp $ ./p
Found at 4
```

Change the input a little:

```
int data[]={2,3,6,8,9,12,13,15,17,19,20};
int target=12;
```

Compile and run

```
walter@mint2 ~/Documents/temp $ gcc test.c -o p
walter@mint2 ~/Documents/temp $ ./p
Found at 5
```

and again:

```
int data[]={2,3,6,8,9,12,13,15,17,19,20};
int target=10;
```

and get:

```
walter@mint2 ~/Documents/temp $ gcc test.c -o p
walter@mint2 ~/Documents/temp $ ./p
Segmentation fault
```

'Segmentation fault' gives very little evidence on what happens, except that it has not worked. Most OS divide memory into segments, possibly for code, data, the stack and so on. A OS throws a 'segmentation fault' when there is some kind of error relating to memory segments. But what? Why?

T*his is where thinking helps*. What is the expected output in this case? The target, 10, would be between the 9 and 12 - but it is not actually there. The expected output is that the target is not present in the data. But our code can never produce that output.

Any algorithm to search any data structure for a value must return one of two possible results - the location of the found data, or that it is not present.

We need to modify the code so that it can return 'not found', where appropriate.

Print debugging

This is a simple technique to get more evidence, which we can use to work out the bug.

At a suitable point in the code we simply print out the values of key variables.

```
#include <stdio.h>

int data[]={2,3,6,8,9,12,13,15,17,19,20};
int target=10;

void binSearch(int lo, int hi)
{
  int middle=(hi+lo)/2;
  printf("lo=%d middle=%d hi=%d\n",lo, middle, hi);
  if (data[middle]==target)
  {
    printf("Found at %d\n",middle);
    return;
  }
  if (data[middle]>target)
  {
    hi=middle;
  }
  else
  {
    lo=middle;
```

```
  }
  binSearch(lo, hi);

}

int main(void)
{
binSearch(0, 10);
return 0;
}
```

When we run this we get:

```
walter@mint2 ~/Documents/temp $ gcc test.c -o p
walter@mint2 ~/Documents/temp $ ./p
lo=0 middle=5 hi=10
lo=0 middle=2 hi=5
lo=2 middle=3 hi=5
lo=3 middle=4 hi=5
lo=4 middle=4 hi=5
lo=4 middle=4 hi=5
lo=4 middle=4 hi=5
lo=4 middle=4 hi=5
lo=4 middle=4 hi=5
lo=4 middle=4 hi=5
lo=4 middle=4 hi=5
..
```

and eventually we get a segmentation fault. So it gets stuck in an infinite recursion. Each recursive call uses the stack, so we eventually we run out of stack space. The segmentation fault is the stack segment becoming full.

Why? Check on paper. If lo is 4 and hi is 5, lo+hi is 9. When we divide this by 2, using integer division, we get 4 (remainder 1, discarded). So it looks at location 4, does not find the target, and continues with another call.

We can fix it like this:

```
#include <stdio.h>

int data[]={2,3,6,8,9,12,13,15,17,19,20};
int target=10;

void binSearch(int lo, int hi)
{
  int middle=(hi+lo)/2;
  if (hi==lo+1 && data[middle]!=target)
  {
    printf("Not present\n");
    return;
  }
  printf("lo=%d middle=%d hi=%d\n",lo, middle, hi);
  if (data[middle]==target)
  {
    printf("Found at %d\n",middle);
    return;
  }
  if (data[middle]>target)
  {
    hi=middle;
  }
  else
```

```
  {
    lo=middle;
  }
  binSearch(lo, hi);

}

int main(void)
{
  binSearch(0, 10);
  return 0;
}
```

which gives

```
lo=0 middle=5 hi=10
lo=0 middle=2 hi=5
lo=2 middle=3 hi=5
lo=3 middle=4 hi=5
Not present
```

We need to say

```
  if (hi==lo+1 && data[middle]!=target)
```

to take into account the possibility that we are searching an array of length 2, so hi=lo+1 to start with, but the target is present.

This code still contains a bug. We leave it to the reader to find and fix it.

Exception handling

Sometimes a problem arises because of the environment the code is executing in. For example it might run out of memory, or a file is not found, a network connection fails, or a user inputs garbage.

Some languages have rather cumbersome ways to deal with that. Exactly what 'deal with it' means is part of the program design. But that should include the designer planning 'if that happens, we want this to be the result.' Some exceptions usually cannot be recovered from - like out of memory. Some can - like file not found - use a different file, or use default values.

Java has the try catch control construct for handling exceptions. This is *not* a way to deal with program bugs. It is a bug to not handle exceptions sensibly.

For example, suppose we want to input an integer. We input a string, then parse that to an int:

```
import java.util.Scanner;

public class Test
{
  public static void main(String[] args)
  {
    Scanner scanner=new Scanner(System.in);
    System.out.print("Please enter a whole number");
    String str=scanner.next();
    int integer=Integer.parseInt(str);
    System.out.println(integer);
  }
}
```

This sometimes works:

```
walter@mint2 ~/Documents/temp $ java Test
```

```
Please enter a whole number234
234
```

but sometimes not:

```
walter@mint2 ~/Documents/temp $ java Test
Please enter a whole numberhello
Exception in thread "main" java.lang.NumberFormatException: For input string:
"hello"
  at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at Test.main(Test.java:10)
```

One solution is to use a loop, repeating the input until we get valid data. Inside the loop we use a try catch, and catch a NumberFormatException:

```
import java.util.Scanner;

public class Test
{
  public static void main(String[] args)
  {
    Scanner scanner=new Scanner(System.in);
    boolean inputWrong=true;
    int integer=0;
    System.out.print("Please enter a whole number ");
    while (inputWrong)
    {
      try
      {
        String str=scanner.next();
        integer=Integer.parseInt(str);
        inputWrong=false;
      }
      catch (NumberFormatException ex)
      {
        System.out.print("Please enter a whole number");
        System.out.println(" - try again");
      }
    }

    System.out.println(integer);
  }
}
```

which does this:

```
walter@mint2 ~/Documents/temp $ java Test
Please enter a whole number hello
Please enter a whole number - try again
12.45
Please enter a whole number - try again
12
12
```