Software Engineering - I

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 12 – Debugging

Debugging

As Benjamin Franklin said, “in this world, nothing is certain but death and taxes.” If you are in the software development business, however, you can amend that statement. Nothing in life is certain except death, taxes, and software bugs. If you cryogenically freeze yourself, you can delay death indefinitely. If you move to a country with no income tax, you can avoid paying taxes by not buying anything. If you develop software, however, no remedy known to mankind can save you from the horror of software bugs.

What is a Bug?

We call them by many names: software defects, software bugs, software problems, and even software “features.” Whatever you want to call them, they are things the software does that it is not supposed to do (or, alternatively, something the software doesn’t do that it is supposed to). Software bugs range from program crashes to returning incorrect information to having garbled displays.

A Brief History of Debugging

It would appear that as long as there have been computers, there have been computer bugs. However, this is not exactly true. Even though the earliest known computer programs contained errors, they were not, at that time, referred to as “bugs.” It took a lady named Admiral Grace Hopper to actually coin the term “bug.”

After graduating from Vassar in 1928, she went to Yale to receive her master’s degree in mathematics. After graduating from Yale, she worked at the university as a mathematics professor. Leaving Yale in 1943, with the onset of World War II, Mrs. Hopper decided to work for the Navy. Mrs. Hopper’s first assignment was under Commander Howard Aiken at Howard University, working at the Bureau of Ordinance Computation. She was a programmer on the Mark II, the world’s first automatically sequenced digital computer. The Mark II was used to determine shooting angles for the big guns in varying weather conditions during wartime.

It was during her term with the Mark II that Hopper was credited with coining the term “bug” for a computer problem. The first “bug” was actually a moth, which flew through an open window and into one of the Mark II’s relays. At that time, physical relays were used in computers, unlike digital components we use today. The moth shorted out across two contacts, temporarily shutting down the system. The moth would later be removed (de-bugged?) and pasted into the logbook of the project. From that point on, if her team was not producing numbers or working on the code, they claimed to be “debugging” the system.

From that auspicious beginning, computer debugging developed into something of a hit-or-miss procedure for quite a few years. Early debugging efforts mostly centered around either data dumps of the system or used output devices, such as printers and display lights, to indicate when an error occurred. Programmers would then step through the code line by line until they could determine the location of the problem. The next step in the

evolution of debugging came with the advent of command-line debuggers. These simple programs were an amazing step forward for the programmers. Although difficult to use, even at the time, these programs represented the first real attempt to turn debugging from a hit-or-miss proposition into a reproducible process.
Once the debugger became a part of the programmer's arsenal, the software world began to consider other ways that programs could be more easily debugged. Software projects starting getting bigger, and the same techniques that worked well for small projects no longer worked when the program reached a certain size.

Importance of Debugging

As we mentioned earlier in this course, one of the prime objectives of software engineering is to develop cost effective software applications. According to a survey, when a software application is in the maintenance phase, 20% of its lifecycle cost is attributed towards the defects which are found in the software application after installation. Please bear in mind that the maintenance is the phase in which 2/3rd of the overall software cost incurs. Therefore, 20% of the 2/3rd cost is again a huge cost and we need to understand why this much cost and effort is incurred. In fact, when a software application is installed and being used, any peculiarity in it can cost a lot of direct and indirect damages to the organization. A system downtime is the period in which tremendous pressure is on developers end to fix the problem and make the system running again. In these moments, every second costs hugs losses to the organization and it becomes vital to find out the bug in the software application and fix it. Debugging techniques are the only mechanism to reach at the code that is malfunctioning. In the following subsection, we shall discuss an incident that took place in 1990 and see how much loss the company had to suffer due to a mere bug in the software application.

Problem at AT&T

In the telecommunications industry, loss of service is known as an outage. For most of us, when as outage occurs, we lose our telephone service; we cannot make calls and we cannot receive calls. Outages are accepted and expected hazards in the industry.
On January 15. 1990, AT&T had a US wide telephone system outage that lasted for nine hours. The cause was due to a program error in the software that was meant to make the system more efficient. Eight years later, on April 13, 1998, AT&T suffered another massive failure in its frame-relay network, which affected ATM machines, credit card transactions and other business data services. This failure lasted 26 hours. Again, the bug was introduced during a software upgrade.

Description of the Problem

The code snippet that caused the outage is illustrated as follows

```
1. do {
2.     . . .
3.     switch (expression) {
4.         case 0: {
5.             if (some_condition) {
6.                 . . .
7.                 break;
8.             } else {
9.                 . . .
10.            }
11.           . . .
12.           break;
13.       }
14.     . . .
15.   }
16.   . . .
17. } while (some_other_condition);
```

In this case the break statement at line 7 was the culprit. As implemented, if the logical_test on line 5 was successful, the program should have proceeded to line 6 to execute those statements. When the program stepped to line 7, the break statement caused the program to exit the “switch” block between line 3 and line 15, and proceeded to execute the codes in line 16. However, this path of execution was not the intention of the programmer. The programmer intended the break statement in line 7 to break the if-then clause; so, after the program executed line 7, it was supposed to continue execution in line 11.

AT&T statement about the Problem

“We believe that the software design, development, and testing processes we use are based on solid, quality foundations. All future releases of software will continue to be religiously tested. We will use the experience we've gained through this problem to further improve our procedures.”

We do not believe we can fault AT&T’s software development process for the 1990 outage, and we have no reason to believe that AT&T did not rigorously test its software update. In hindsight, it is easy to say that if the developers had only tested the software, they would have seen the bug. Or that if they had performed a code inspection, they

would have found the defect. Code inspection might have helped in this case. However, the only way the code inspection could have uncovered this bug is if another engineer saw this particular line of code and asked the original programmer if that was his or her intension. The only reason that the code reviewers might have asked this question is if they were familiar with the specifications for this particular code block.

Art and Science of Debugging

Debugging is taken as an art but in fact it is a scientific process. As people learn about different defect types and come across situations in which they have to debug the code, they develop certain heuristics. Next time they come across a similar situation, they apply those heuristics and solve the problem in lesser time and with a lesser effort. While discussing the debugging process we discuss the phenomenon of "you miss the obvious". When a person writes a code, he develops certain impression about that code. One can term this impression as a personal bias that the developer builds towards his creation the "code" and when he has to check this code, he can potentially miss out obvious mistakes due to this impression or bias. Therefore, it is strongly recommended that in order to reach to a defect in the code, one needs "another pair of eyes". That is, start discovering the defect by applying your own heuristics and if you could reach to the problem, fine, otherwise ask a companion to help you in this process. We shall further elaborate this idea based on the following example.

Program at Bulletin Board Example

Following piece of code was once placed on a bulletin board making an invitation to people to discover the problem in this code. Please look at this code and discover the problem

```
1. while (i =0; i < 10; i++) {
2.     cout << i << newl;
3. }
```

Well if you could not guess it by now, the problem lies with the syntax of the while loop. This is so obvious that almost everyone forgot to ponder upon when placed on the bulletin board of a university. The loop is "while" but the syntax used is a "for" loop.

In order to reach such defects, one needs a scientific approach to check and verify the code methodically. Based on this discussion, we are now in a position to introduce a few classes of bugs to the reader. This is not an exhaustive list since there could be a number of other classes of bugs as well but the following classes will help the reader know about some well known bugs that we usually find in the code.

Bug Classes

Memory and resource leak

A memory leak bug is one in which memory is somehow allocated from either the operating system or an internal memory "pool", but never deallocated when the memory is finished being used.

Symptoms

- System slowdowns
- Crashes that occur "randomly" over a long period of time

Example 1

Let's take a look at a simple memory leak error that can occur trivially in a C program. This type of code is found in hundreds of programs across the programming spectrum. It illustrates the simplest possible case of a memory leak, which is when memory is allocated and not deallocated in all cases. This particular form of the bug is the most frustrating because the memory is usually deallocated, but not always

```
char *buffer = new char[kMaxBufferSize+1];
memset(buffer,0,kMaxBufferSize+1);

// Do some stuff to fill and work with the character buffer

if (IsError(nCondition)) // Did we get an error in the processing piece?
{
  Message(("An error occured.
            Skipping final stage"));
  return FailureCode;
}

// Final stage of code

// Free up all allocated memory

delete buffer;
return okCode;
```

Note that in many cases, this code works perfectly. If no error occurs in the processing stage (which is the norm) the memory is freed up properly. If, however, the processing stage encounters an error, the memory is not freed up and a leak occurs.

Example 2

The following code snippet from some old C++ code contains a slightly more insidious bug. It illustrates the point well.

```
Class Foo
{
  private:
    int nStringLength;
    char *sString;
  public:
    Foo()
    {
      nStringLength = 20; //Default
      sString = new
      char[nStringLength + 1];
    }
    ~Foo()
    {
      delete sString;
    }
    void SetString(const char
                    *inString)
    {
      sString = new char
                [strlen(inString+1)];
      if(inString == NULL)
        return;
      strncpy(sString, inString,
              strlen(inString));
      nStringLength =
            strlen(inString)+1;
    }
};
```

Let's discuss what happens. In most cases, the Foo object is created and nothing bad happens. If, however, you call the SetString method, all bets are off. The previously allocated string is overwritten by the new allocation, and the old allocated memory goes nowhere. We have an instant memory leak. Worse, we can do this multiple times if we accidentally call the method with a NULL string because it first allocates the block, and then checks to see if the input string was correct. Don't do things like this. If you see an allocation in a class, check to see if the string can be allocated before it gets to that point.

Logical Errors

A logical error occurs when the code is syntactically correct but does not do what you expect it to do.

Symptoms

- The code is misbehaving in a way that isn't easily explained.
- The program doesn't crash, but the flow of the program takes odd branches through the code.
- Results are the opposite of what is expected.
- Output looks strange, but has no obvious symptoms of corruption.

Example

```
// Make sure that the input is valid. For this value, valid ranges are 1-10 and 15-20

if((input >= 1 && input <= 10) &&
  (input >= 15 && input <= 20))
  {
    // Do something for valid case
  }
else
  {
    // Do something for invalid case
  }
```

In order for the code to enter the valid case, the number must be between 1 and 10 and be between 15 and 20. If a number is between 1 and 10, how can it possibly be between 15 and 20 as well? Seems unlikely, doesn't it? This is a typical logical error.

Coding errors

A coding error is a simple problem in writing the code. IT can be a failure to check error returns, a failure to check for certain valid conditions, or a failure to take into account other parts of the system. Yet another form of a coding error is incorrect parameter passing and invalid return type coercion.

Symptoms

- Unexpected errors in black box testing.
- The errors that unexpectedly occur are usually caused by coding errors.
- Compiler warnings.
- Coding errors are usually caused by lack of attention to details.

Example

In the following example, a function accepts an input integer and converts it into a string that contains that integer in its word representation.

```
void convertToString(int InInteger,
      char* OutString, int* OutLength)
{
  switch(InInteger){
     case 1: OutString = "One";OutLength = 3;
               break;
     case 2: OutString = "Two";OutLength = 3;
               break;
     case 3: OutString = "Three";OutLength = 5;
               break;
     case 4: OutString = "Four";OutLength = 4;
               break;
     case 5: OutString = "Five";OutLength = 4;
               break;
     case 6: OutString = "Six";OutLength = 3;
               break;
     case 7: OutString = "Seven";OutLength = 5;
               break;
     case 8: OutString = "Eight";OutLength = 5;
               break;
     case 9: OutString = "Nine";OutLength = 4;
               break;

     }
}
```

There are a few things to notice in the preceding code. The ConvertToString function does not handle all cases of inputs, which becomes very obvious you pass a zero value. Worse, the function does not initialize the output variables, leaving them at whatever they happened to be when they came into the function. This isn't a problem for the output string, necessarily, but it will become a serious issue for the output length.

Memory over-runs

a memory overrun occurs when you use memory that does not belong to you. This can be caused by overstepping an array boundary or by copying a string that is too big for the block of memory it is defined to hold. Memory overruns were once extremely common in the programming world because of the inability to tell what the actual size of something really was.

Symptoms

- Program crashes quite regularly after a given routine is called, that routine should be examined for a possible overrun condition.
- If the routine in question does not appear to have any such problem the most likely cause is that another routine, called in the prior sequence, has already trashed variables or memory blocks.
- Checking the trace log of the called routines leading up to one with the problem will often show up the error.

Example

This particular example shows not only a memory overrun, but also how most programmers "fix" problems in an application. The ZeroArray function steps all over the array boundaries by initializing 100 separate entries. The problem is that that particular array only has 50 slots available in its allocation. What happens at that point is that the function goes past the end of the array and starts to walk on things beyond its control.

```
const kMaxEntries = 50;
int gArray[kMaxEntries];
char szDummyBuffer[256];
int nState = 10;

int ZeroArray (int *pArray)
{
  for (inti=0;i<100;++i)
    pArray[i] = 0;
}
```

Loop Errors

- Loop errors break down into several different subtypes.
- They occur around a loop construct in a program.
- Infinite loops, off-by-one loops, and improperly exited loops.

Symptoms

- If your program simply locks up, repeatedly displays the same data over and over, or infinitely displays the same message box, you should immediately suspect an infinite loop error.
- Off-by-one loop errors are quite often seen in processes that perform calculations.
- If a hand calculation shows that the total or final sum is incorrect by the last data point, you can quickly surmise that an off-by-one loop error is to blame.
- Likewise, if you were using graphics software and saw all of the points on the screen, but the last two were unconnected, you would suspect an off-by-one error.
- Watching for a process that terminates unexpectedly when it should have continued.

Example

```
bool doneFlag;
doneFlag = false;
while(!doneFlag){
  ...
  if( impossibleCondition )
    doneFlag = true;
}
```

The preceding code fragment contains an indirect looping error such that the loop will continue until the impossibleCondition becomes true which is impossible to happen.

```
int anArray[50];
int i;
i = 50;
while(i >= 0){
  anArray[i] = 0;
  i = i - 1;
}
```

In this example, the programmer was trying to be smart. He worked his way backward though the array, assuming that this way there would be no chance of an error. Of course, if you examine the loop, you will find that it performed 51 times. There are only fifty elements in the array. This will certainly lead to a problem later on in the program, if it doesn't first trigger an immediate error from the system.

```
int nIndex = 0;
for(int i=0; i<kMaxIterations; ++i)
{
  while (nIndex < 20)
  {
    ComputeSomething(i*20 + nIndex);
    nIndex ++;
  }
}
```

The final situation is an improper exit condition for a loop. This is most easily illustrated using a set of nested loops. In the above example, we are trying to compute something within the inner loop for some number of iterations. The code appears to do what we said it should do. It computes whatever it is we are computing in the inner loop for each iteration of the outer loop, 20 times. The problem, however, is that the exit condition simply says that nIndex should be less than 20. The first time through the outer loop, this variable will become 21, and the inner loop will exit. This is correct, and exactly the way you would expect this to happen. The problem, however, is that the next time through the loop, the inner loop will not be executed at all.

Pointer errors

- A pointer error is any case where something is being used as an indirect pointer to another item.
- Uninitialized pointers: These are pointers that are used to point at something, but we fail to ever assign them something to point at.
- Deleted pointers, which continue to be used.
- An Invalid pointer is something that is pointing to a valid block of memory, but that memory does not contain the data you expect it to.

Symptoms

- The program usually crashes or behaves in an unpredictable and baffling way.
- You will generally observe stack corruptions, failure to allocate memory, and odd changing of variable values.
- Changing a single line of code can change where the problem occurs.
- If the problem "goes away" when you place a print statement or new variable into the code that you suspect contains the problem.

Example

Let's take a look at one of the most common forms of the pointer error, using a pointer after it has been deleted. As you can see by the following example, it is not always clear when you are doing this incorrectly:

```
void cleanup_function(char *ptr)
{
  SaveToDisk(ptr);
  delete ptr;
}

int func()
{
  char *s = new char[80];
  cleanup_function(s);
  delete s;
}
```

In this example, the programmer meant to be neat and tidy. He probably wrote the code originally to allocate the pointer at the top of this function, and then by habit put in a deallocation at the bottom of the function. This is, after all, good programming practice to avoid memory leaks. The problem is introduced with the cleanup_function routine. This function was probably written at a later time by another developer and might have been used to ensure that all allocated pointers were stored to the permanent drive and then freed up to avoid any possible leaks. In fact, the code in question comes from a set of functions that was reused from a memory pool system that saved its data to disk before destroying the pointer, so that another pointer could be retrieved from the persistent pool.

The problem is, the reuse did not take into account the existing system. This problem is common when reusing only parts of a system.

When the second delete occurs at the bottom of func(), the results are unpredictable. At best, the delete function recognizes that this pointer has been deleted already and doesn't do anything. At worst, memory is corrupted and you are left with a memory crash somewhat later in the program, which will be very difficult to find and fix.

Boolean bugs

Boolean bugs occur because the mathematical precision of Boolean algebra has virtually nothing to do with equivalent English words.
When we say "and", we really mean the boolean "or" and vice versa.

Symptoms

- When the program does exactly the opposite of what you expect it to. For example, you might have thought you needed to select only one entry from a list in order to proceed. Instead, the program will not continue until you select more than one. Worse, it keeps telling you to select only one value.
- For true/false problems, you will usually see some sort of debug output indicating an error in a function, only to see the calling function proceed as though the problem had not occurred.

Examples

The following code contains a simple example of a function that returns a counter-intuitive Boolean value and shows how that value leads to problems in the rest of the application code. This case performs an action and indicates to the user whether or not the action succeeded.

```
int DoSomeAction(int InputNum)
{
  if(InputNum < 1 || InputNum  > 10)
       DoSomeAction = 0;
  else
  {
    PerformTheAction(InputNum);
    DoSomeAction = NumberOfAction + 1;
    NumberOfAction = NumberOfAction + 1;
  }
  return DoSomeAction;
}
```

After looking at the function, ask yourself this – What is the return value in the case of success? What would it be in the case of failure? How would you know this by simply glancing at the function declaration?

```
int Value;
int Ret;

Value = 11;
Ret = DoSomeAction(Value);
if(Ret)
  cout <<Error in DoSomeAction<<;
```

If you were debugging this application, you might start out by wondering why it didn't work. Further investigation would show that the error statement is never triggered in spite of the fact that the routine did, in fact, encounter an error.. Why did this happen? It happened because the routine didn't return nonzero value for failure, it returned nonzero for success. This Boolean value was not intuitive given the condition of the function and led to a poor assumption by the programmer who wrote the code that used it.

Holistic approach

Holistic means

- Emphasizing the importance of the whole and the interdependence of its parts.
- Concerned with wholes rather than analysis or separation into parts

What this meant is that a holistic approach focuses on the entire system rather than whatever piece appears to be broken. Holistic medicine, for example, concerns itself with the state of the body as a whole, not the disease that is currently attacking it. Similarly, programmers and debuggers must understand that you cannot treat the symptoms of a problem, you must focus on the application system as a whole.
Now that you understand what holistic means, how do you apply holistic concepts to debugging the software programs and procedures? Let's start by thinking about what emphasizing the whole over the individual pieces means in debugging. When you are debugging an application, you often simply look at the problem reported by the user and how you can make that problem go away.

As a simple example, consider the case where the program mysteriously crashes at a particular point in the code each and every time the program is run. The point at which the code crashes makes no sense at all. For example, in C++, you might have a member function of a class that reads:

```
void SetX(int X)
{
  mX = X;
}
```

The program always crashes on the line that assigns X to the member variable mX(mX = X). Looking at this code snippet, we see very few valid ways in which the program could be encountering a problem at this point. Oddly, we might discover that inserting a

message statement makes the problem stop occurring when the indicated steps are followed for reproducing the problem. Is the problem fixed?

Most experienced programmers, managers, or debuggers would say that the problem is not fixed, although few could tell you exactly why that is the case. The fact is, in this case, we are directly treating the symptoms (the program crashing) rather than looking into the actual problem and ignoring the overall problem is endemic in our industry.

The debugging process

In normal circumstances, you will have a user description of the problem. This description might have been given to you directly by the user, or it might have been gathered by a customer support person or other nontechnical person, In any event, this data has to be considered suspiciously until you can get a first hand description of what really happened. First hand accounts of the problem are always useful, so be sure to write down exactly what you are told. That way, you can compare several accounts of the same problem and look for similarities. Consider, for example, the following accounts of a reported bug in a system:

1. "I started by trying to setup a Favorites list. I first went to the home page. Then, I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the keyboard. Then I moved to the fourth entry on the submenu and clicked it with the mouse. Finally, I entered my name as Irving, clicked on OK, and the program crashed."

2. "I went to the programming menu and selected the New menu option. I then clicked on the Create Object menu entry and entered the name HouseObject for the object name field. Then I clicked OK, and the program crashed."

3. "I selected the New Project menu option, and then clicked on the icon that looks like a Gear. I entered the name Rudolph for the project name, and then clicked OK. The program crashed."

    - After all, they appear to relate to three different sections of the program.
      Most people, when reporting bugs, focus on what they think was the important step in the process.
    - Filter out the unimportant information and see what each case really has in common.
      First, each user clicked on the OK button to finalize the process, and the program crashed.
    - It might seem likely, therefore, that the program OK handler contains a fatal flaw.
    - A bit of experimentation will show whether this is the case or not.
    - When we examine the statements of the witnesses, we notice that each was working with a menu.

- We can see that each person used both the keyboard and the mouse to select from the menus before clicking the OK button.

"I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the keyboard. Then I moved to the fourth entry on the submenu and clicked it with the mouse."

"... selected the New menu option. I then clicked on the Create Object menu entry ..."

"I selected the New Project menu option, and then clicked on the icon that looks ..."

- We can infer, therefore, that the user in this case used both the keyboard and mouse.
  The next logical place to look is in the menu handler to see whether it deals with mouse and keyboard entries differently.
- The final clue is in the third entry, where the user did not select anything from the menu with the mouse, but instead clicked on an icon.
- While looking at the code, I would try to see what happens when the program deals with a combination of keyboard and menu entry.
- It is likely, given the user discussion, that you will find the root of the problem in this area.
- After finding such a common bug, it is likely that the fix will repair a whole lot of problems at once
- This is a debugger's dream.

Good clues, Easy Bugs

Get A Stack Trace

In the debugging process a stack trace is a very useful tool.
Following stack trace information may help in debugging process.

- Source line numbers in stack trace is the single, most useful piece of debugging information.
- After that, values of arguments are important
    - Are the values improbable (zero, very large, negative, character strings with non-alphabetic characters?
- Debuggers can be used to display values of local or global variables.
    - These give additional information about what went wrong.

Non-reproducible bugs

- Bugs that won't "stand still" (almost random) are the most difficult to deal with.
- Randomness itself, however, is information.
- Are all variables initialized? (random data in variables could affect output).
- Does bug disappeared when debugging code is inserted? Memory allocation (malloc) problems are probably a culprit.
- Is the crash site far away from anything that could be wrong?
- Check for dangling pointers.

Example

```
char *msg(int n, char *s)
{
  char buf[100];
  sprintf(buf, "error %d: %s\n",
           n, s);
  return buf;
}
...
p = msg(20, "Output values");
...
q = msg(30, "Input values");
...
printf("%s\n",p);
```

