

Debugging

Nigel Horspool
(with some material by Mike Zastre)



Topics

- Kinds of Programming Errors
- What is a Bug?
- Classifying Different Kinds of Bugs
- Bug Avoidance Strategies
- Bug Detection Approaches
- Debugging Tools



History Lesson: The First Bug?

A dead moth found
trapped between
electrical contacts in
the **Mark II Aiken**
Relay Calculator,
9 September 1945.

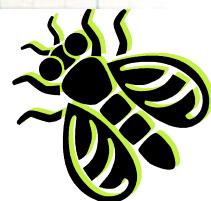
(Some say it was 1947.)

Photo # NH 96566-KN First Computer "Bug", 1945

92

9/9

0800 Antran started { 1.2700 9.037847025
1000 .. stopped - antran ✓ 9.037846995 contact
13' w.c (032) MP-MC ~~1.982147000~~ ~~2.130476415~~ (2) 4.615925059 (-2)
033) PRO 2 2.130476415
contact 2.130676415
Relays 6-2 in 033 failed special speed test
in relay .. 10.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antran started.
1700 closed down.



Source: Wikipedia entry for *bug*

A Bug is NOT:

any kind of
mistake which
causes the
compiler to
generate an
error message

any kind of error
which causes
the linker to
report a problem
(typically an
undefined or
duplicate symbol).



Bug Terminology

failure

something visible to the program's user

fault

the state of the program which leads to a failure

error

the incorrect code fragment which leads to the fault and thus to the failure

Source: Andrew Ko and Brad Myers, "Development and Evaluation of a Model of Programming Errors", HCC'03.



Some Observable Failures

- **Infinite Loop**
- **Deadlock**
- **Incorrect Output**
- **Memory Leak**
- **Run-Time Exception**
 - the exception possibilities are very language and platform dependent



Run-Time Failure Examples

Some Possibilities for C/C++ or Assembler Code

- “Blue Screen of Death”
- Segmentation fault
- Unaligned memory address
- Invalid instruction
- ...

The Infamous Blue Screen of Death

Rapid

A fatal exception 0E has occurred at 0028:C0011E36 in UXD UMM(01) + 00010E36. The current application will be terminated.

- * Press any key to terminate the current application.
- * Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue _

Now just a
memory!

But we have new BSOD flavours

You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワー ボタンを数秒間押し続けるか、リセットボタンを押してください。

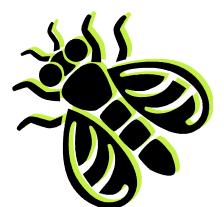


Your PC ran into a problem that it couldn't handle, and now it needs to restart.

You can search for the error online: DPC WATCHDOG VIOLATION

It'll restart in: 9 seconds

```
Call Trace:
[<c041bd7f2>] iounmap+0x9e/0xc8
[<c053480d>] agp_generic_free_gatt_table+0x2e/0x9e
[<c0533991>] agp_add_bridge+0x1a8/0x26f
[<c05439eb>] __driver_attach+0x0/0x6b
[<c04e6bf4>] pci_device_probe+0x36/0x57
[<c0543945>] driver_probe_device+0x42/0x8b
[<c0543a2f>] __driver_attach+0x44/0x6b
[<c054344a>] bus_for_each_dev+0x37/0x59
[<c05438af>] driver_attach+0x11/0x13
[<c05439eb>] __driver_attach+0x0/0x6b
[<c0543152>] bus_add_driver+0x64/0xfd
[<c04e6d22>] __pci_register_driver+0x47/0x63
[<c040044d>] init+0x17d/0x2f7
[<c0403dee>] ret_from_fork+0x6/0x1c
[<c04002d0>] init+0x0/0x2f7
[<c04002d0>] init+0x0/0x2f7
[<c0404c3b>] kernel_thread_helper+0x7/0x10
=====
Code: 78 29 8b 44 24 04 29 d0 8b 54 24 10 c1 f8 05 c1 e0 0c 09 f8 89 02 8b 43 0c
85 c0 75 08 0f 0b 9c 00 77 c8 61 c0 48 89 43 0c eb 08 <0f> 0b 9f 00 77 c8 61 c0
8b 03 f6 c4 04 0f 85 a5 00 00 00 a1 0c
EIP: [<c041bd49>] change_page_attr+0x19a/0x275 SS:ESP 0068:c14f7ec0
<0>Kernel panic - not syncing: Fatal exception
```



Segmentation Fault

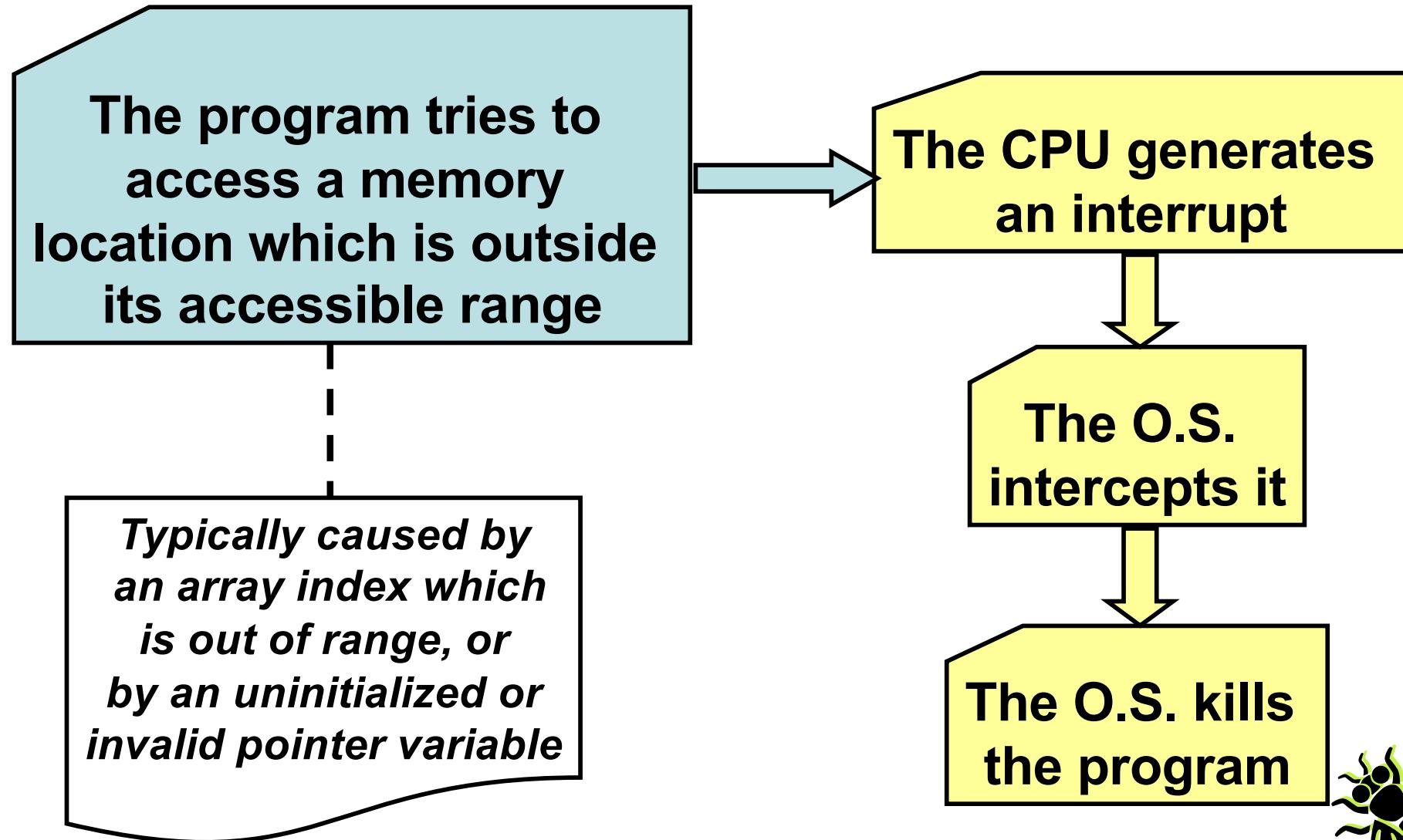
What happens if we try to run this incorrect program?

```
% gcc segfault.c  
% a.out  
* i = 0  
* i = 1  
Segmentation Fault  
(core dumped)  
%
```

```
#include <stdio.h>  
  
int A[100];  
  
int main() {  
    int *p, i;  
    p = A; i = 0;  
    while(1) {  
        printf("* i = %d\n", i++);  
        *p = 0;  
        p -= 100;  
    }  
    return 0;  
}
```



What is a Segmentation Fault?



Bus Error

```
# On Intel x86
```

```
% gcc unaligned.c
```

```
% a.out
```

```
Storing 0 at address  
80608A1
```

```
%
```

```
# On PowerPC
```

```
% gcc unaligned.c
```

```
% a.out
```

```
Storing 0 at address 20901
```

```
Bus Error
```

```
%
```

```
/* C code example */
```

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

```
int A[100];
```

```
int main() {
```

```
    int *p, i;
```

```
    p = A;
```

```
    p = (int*)((int)A + 1);
```

```
    printf("Storing 0 at"
```

```
        " address %x\n", p);
```

```
    *p = 0;
```

```
    return 0;
```

```
}
```



What is a Bus Error?

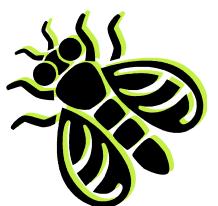
The program has tried
to access a
misaligned data item

e.g. a 4-byte integer
at a location which
is not divisible by 4.

*Typically caused
by uninitialized pointers*

Some computers
generate
an interrupt

Other computers
permit an unaligned
data access (slowly)
and continue



Illegal Instruction (sometimes)

```
# On Intel x86
# But depends on version
# of gcc
% gcc badinstr.c
% a.out
In foo
Illegal instruction
(core dumped)
%
```

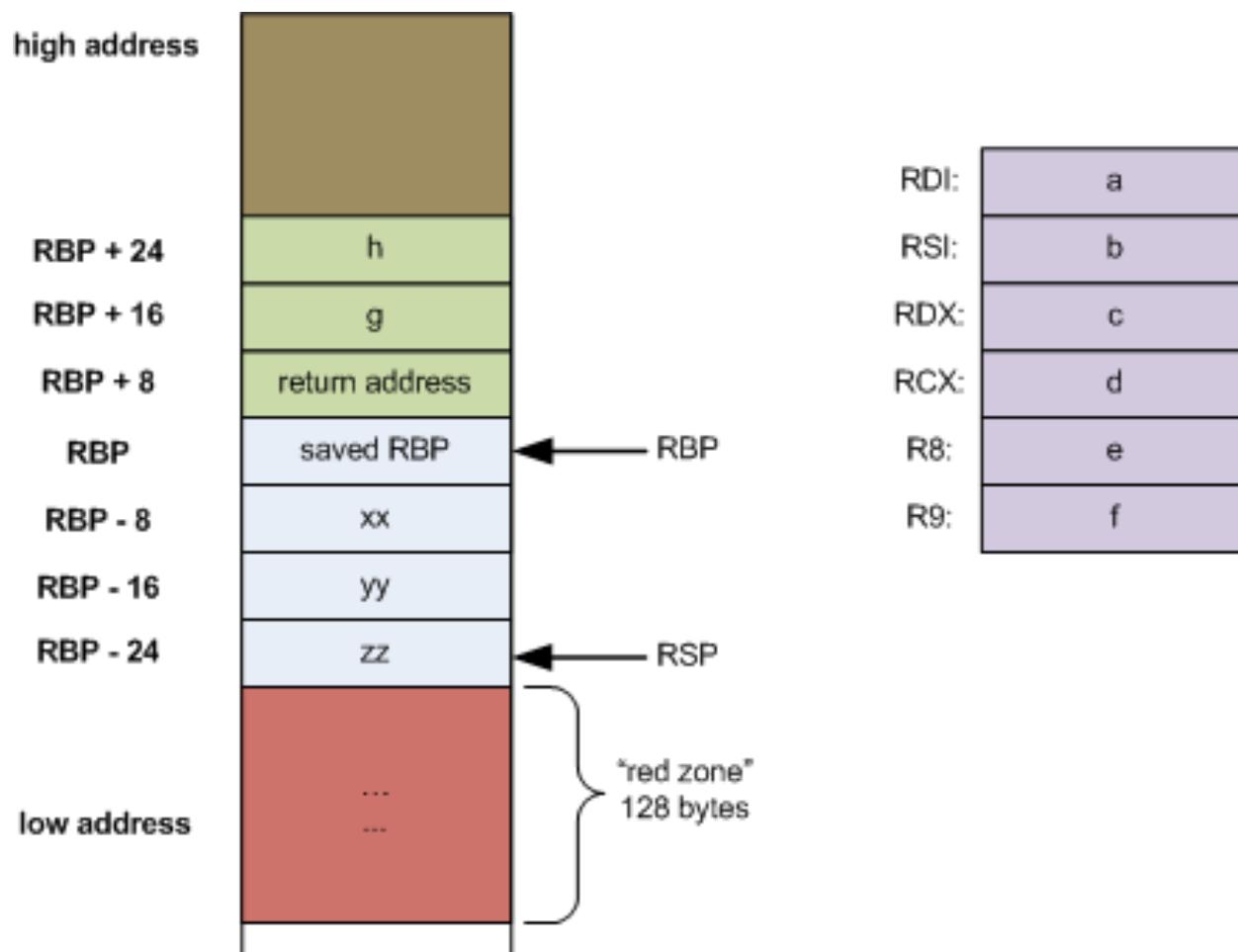
```
/* C code example */
#include <stdio.h>

void foo() {
    int a, *ap;
    ap = &a;
    printf("In foo\n");
    ap[2] -= 3;
}

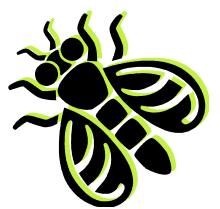
int main() {
    foo();
    printf("I'm back!\n");
    return 0;
}
```



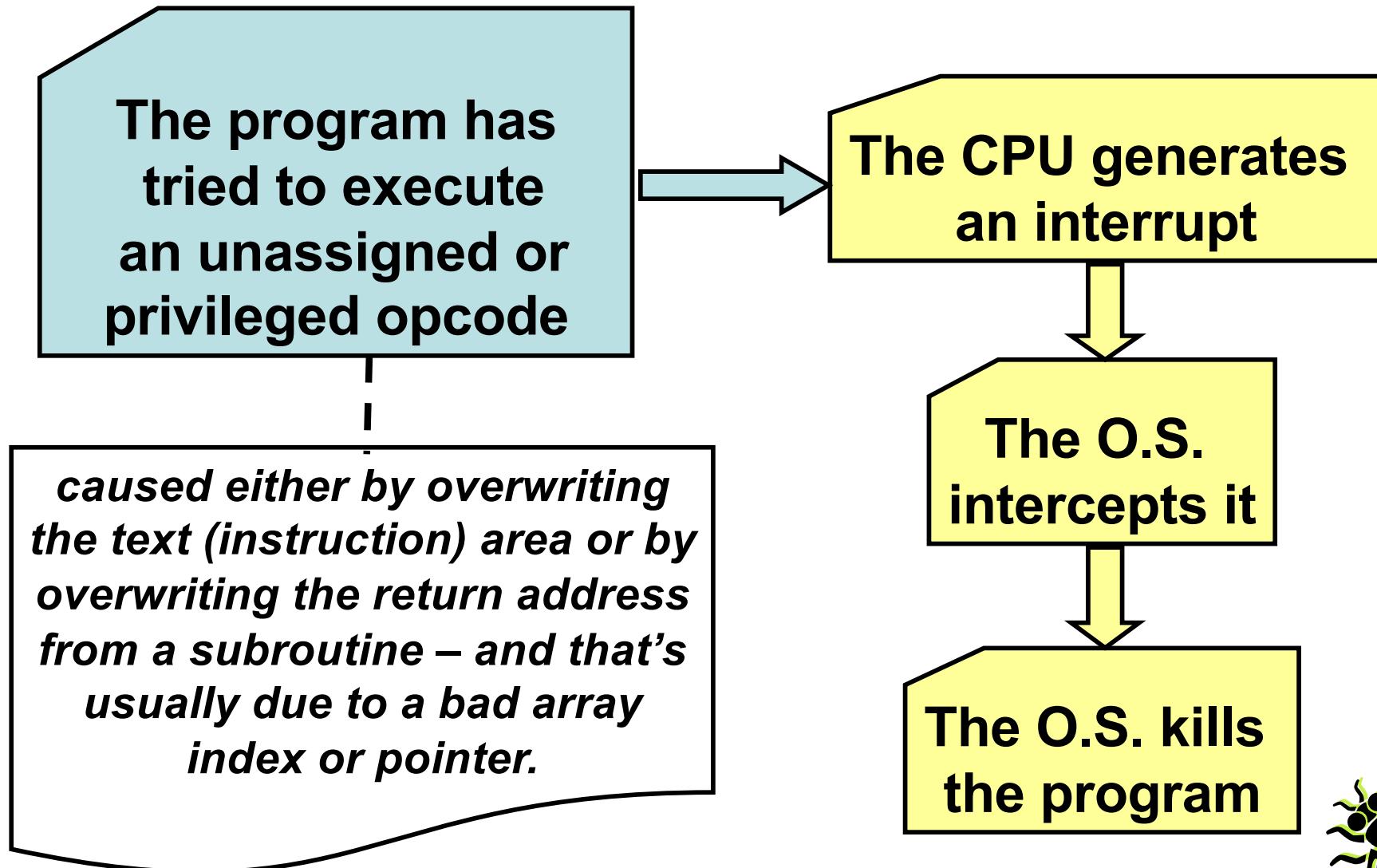
(stack frame)



Source: <http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>



How did we get to an Illegal Instruction?



Memory Leak

**The program runs,
getting slower and
slower, until some
random failure
occurs.**

**In a driver program
or similar, it's the
OS which gradually
degrades.**

```
#include <stdlib.h>

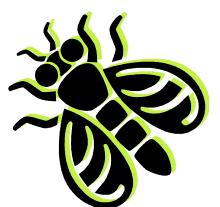
void foo() {
    int *ap;
    ap = malloc(4000);
    ...
    // return without
    // freeing the array
}

int main() {
    ...
    foo();
    ...
}
```



Avoiding Memory Leaks

- Use a programming language where garbage collection is implicit (Java, C#, Perl, Python ...)
- Otherwise, have every subroutine deallocate the objects it creates (except for objects returned as results).
- Take advantage of tools such as valgrind which probe for improper use of dynamic memory.
- And code the program very carefully!



Run-Time Failure Examples

Some Possibilities for Java (or C#)

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- ClassCastException
- OutOfMemoryException
- ... *the complete list is very long!*



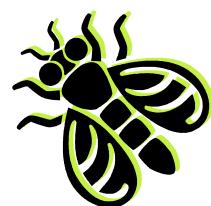
Bug Classifications

- An early and still a definitive list of bug types is due to Donald Knuth.
- He kept a logbook of 850 coding errors when developing T_EX.
- He classified the errors into 15 categories.
- The list here is a condensed version due to Adam Barr. (*Find The Bug*, Addison-Wesley, 2005.)

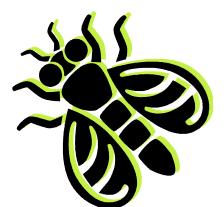
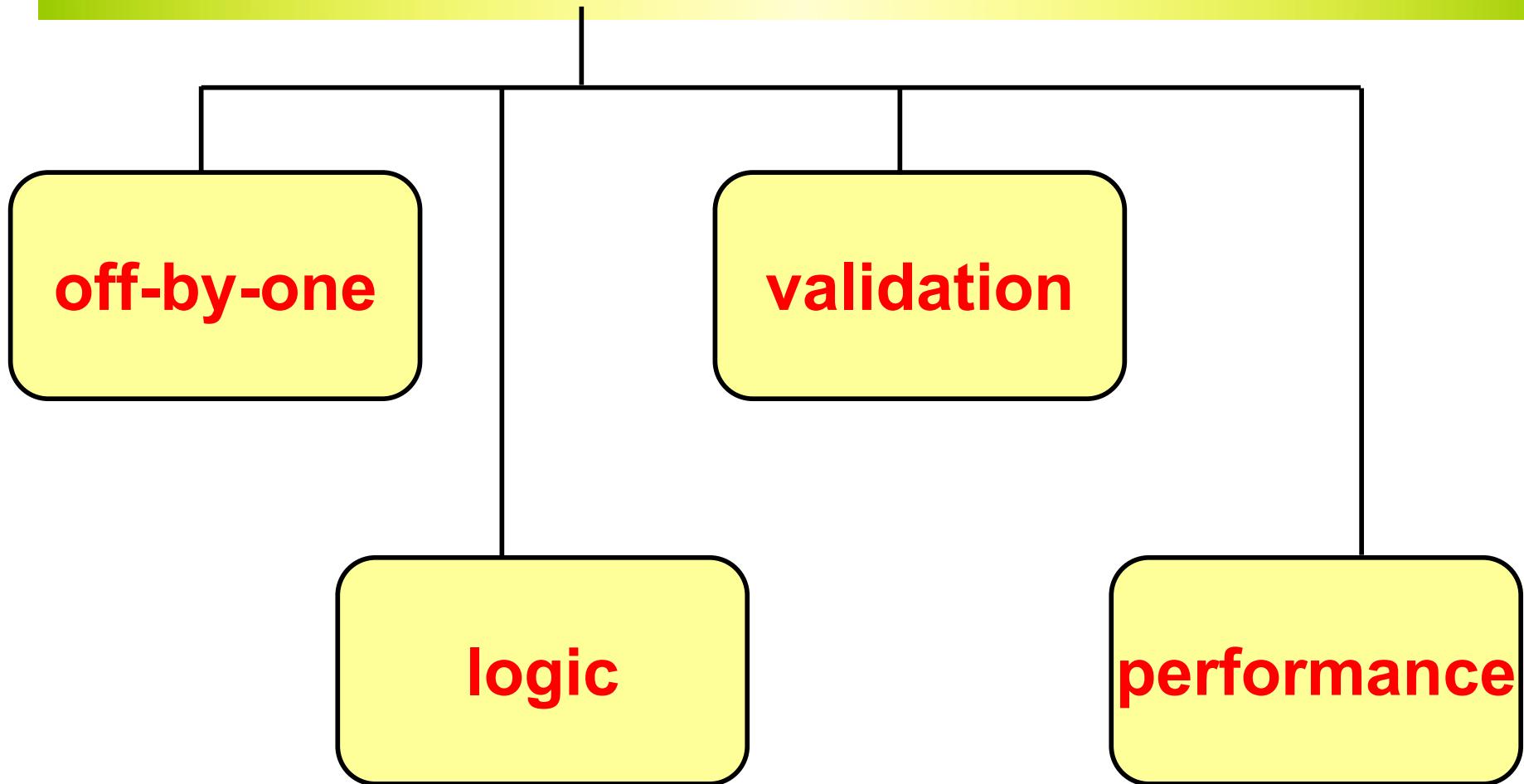


The Knuth/Barr Bug Categories

A – Algorithm	the code follows the intent of the programmer but the intent was wrong
D – Data	the code reads/writes incorrect data or accesses a wrong storage location
F – Forgotten	a missing action/statement or maybe it is in the wrong place in the program
B – Blunder	the algorithm was correct but it was implemented wrong or it was mistyped



A – Algorithm: Subcategories



A – Algorithm: Subcategories

off-by-one

Miscounting by one.

```
// count how many pages to print  
pageCount = lastPage - firstPage;  
  
// initialize the array  
for( int i=0; i<=A.length; i++ ) {  
    A[i] = 0;  
}
```



A – Algorithm: Subcategories

logic

The algorithm is simply wrong

```
// find the greatest difference  
// between any pair of elements  
biggest = 0.0;  
  
for( k = 0; k < A.length-1; k++ ) {  
    distance = abs(A[k]-A[k+1]);  
    if (distance>biggest)  
        biggest = distance;  
}
```



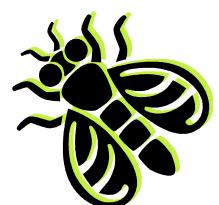
A – Algorithm: Subcategories

Failure to check
validity of variables

validation

```
// find average difference
float avgDiff( float A[], int n ) {
    int i; float sum = 0.0;
    for( i = 1; i<n; i++ )
        sum += abs(A[i]-A[i-1]);
    return sum/n;
}
```

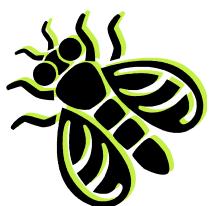
But what happens if n == 1 ?



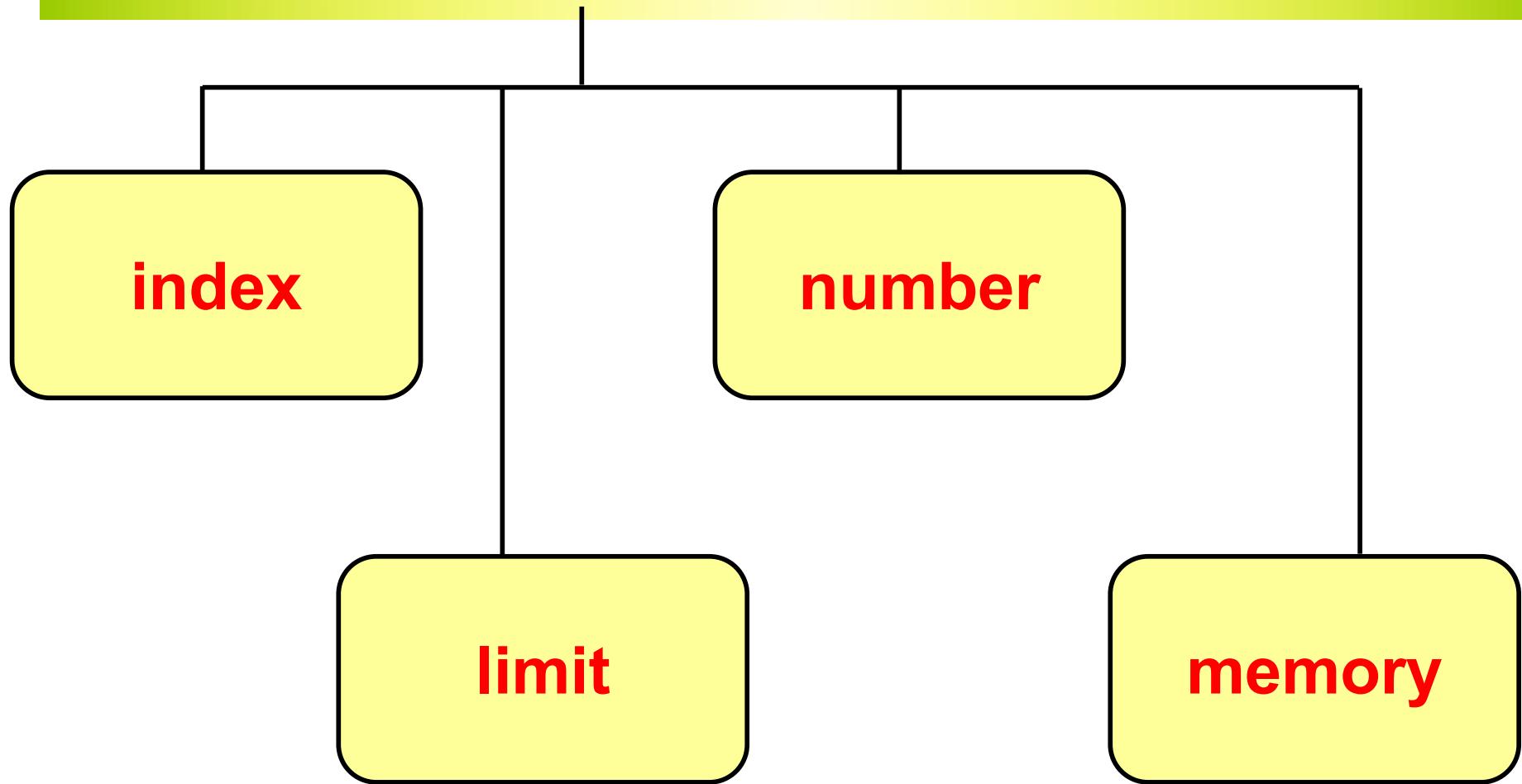
A – Algorithm: Subcategories

- an inappropriate algorithm, or
- a coding error which leads to unacceptable performance

performance



D – Data: Subcategories

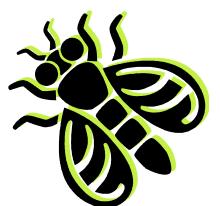


D – Data: Subcategories

index

An array index is wrong.

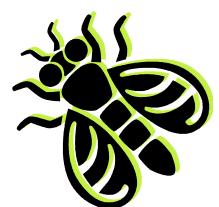
```
int search( int[] Array, int Val ) {  
    int Hi = Array.length; int Lo = 0;  
    while(Lo<=Hi) {  
        int Mid = (Hi+Lo)/2;  
        if (Array[Mid]==Val) return Mid;  
        if (Array[Mid]<Val)  
            Lo = Mid+1;  
        else  
            Hi = Mid-1;  
    }  
    return -1;  
}
```



D – Data: Subcategories

Incorrect handling for the first or last elements in a sequence.

limit

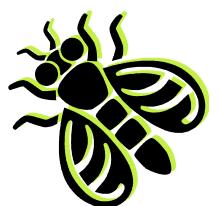


D – Data: Subcategories

Issues related to how numbers are represented in the computer

number

```
for( ; ; ) {  
    unsigned char c = getchar();  
    if (c == EOF) break;  
    ...  
}
```



D – Data: Subcategories

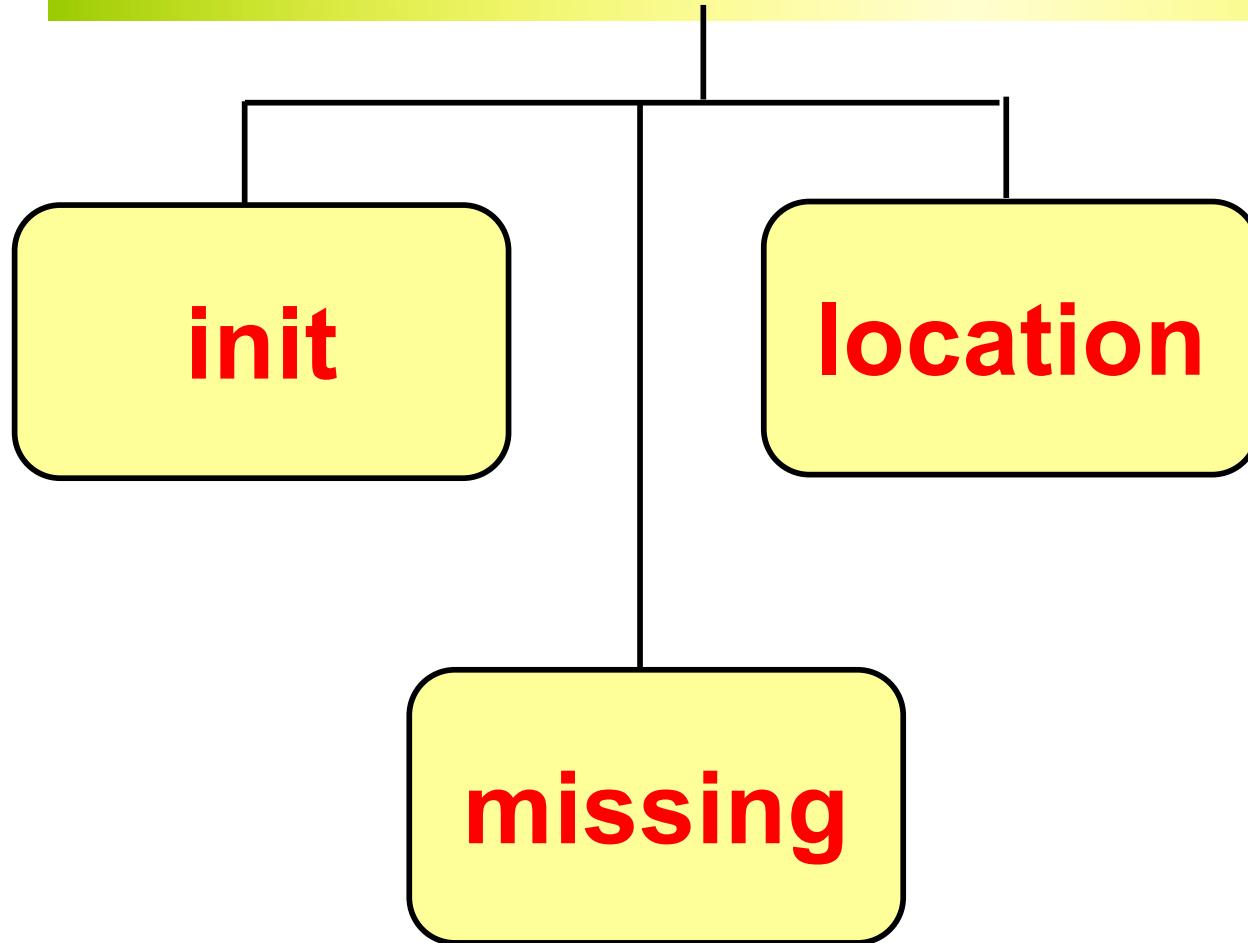
Mismanagement of memory – such as

- attempting to access an inaccessible location,
- or deallocated the same block twice, or
- failing to deallocate a block, or
- continuing to use a block after it has been deallocated.

memory



F – Forgotten: Subcategories

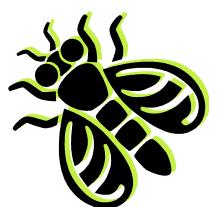


F – Forgotten: Subcategories

init

Forgetting to initialize a variable or a data structure

```
int biggest;  
int i;  
for( i=0; i<N; i++ ) {  
    if (A[i] > biggest) biggest = A[i];  
}
```

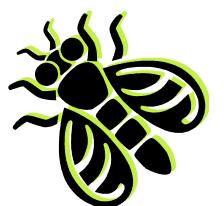


F – Forgotten: Subcategories

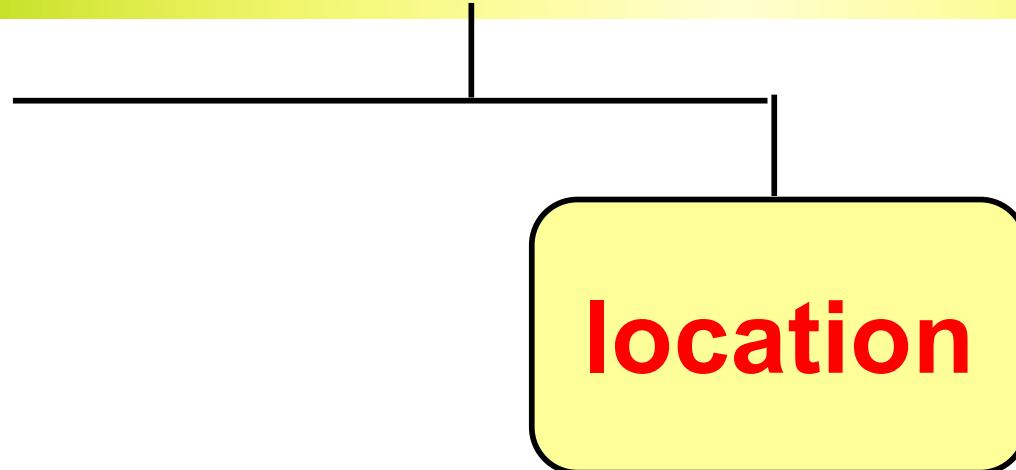
Forgetting to provide code
for some case.

missing

```
for( cnt=0; cnt<N; cnt++ ) {  
    val = ptr->v;  
    if (val < 0) continue;  
    sum += val;  
    ptr = ptr->next;  
}
```



F – Forgotten: Subcategories

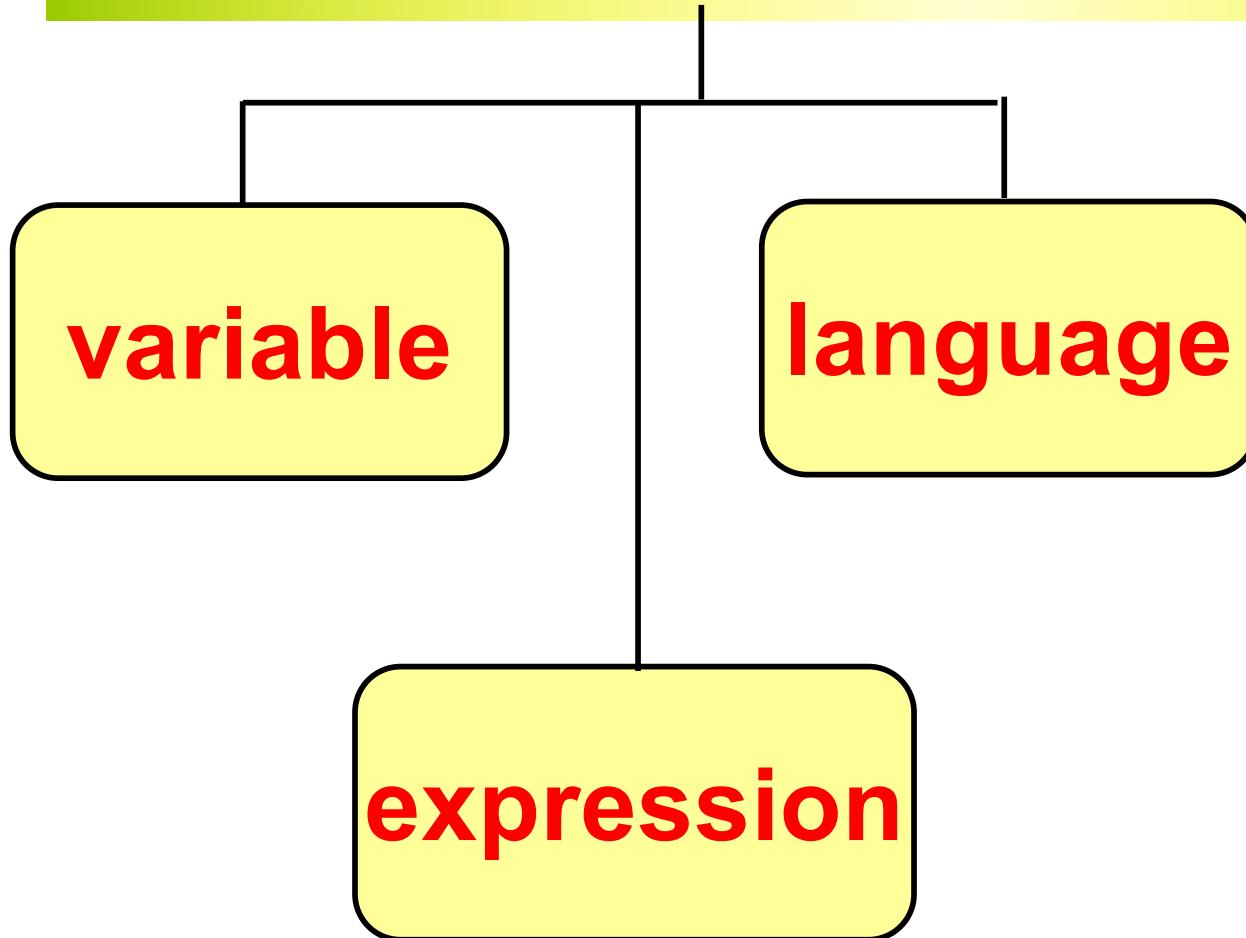


Code appears in the wrong place or is out of order.

```
array[ix] = 0;  
sum += array[ix];
```



B – Blunder: Subcategories



B – Blunder: Subcategories

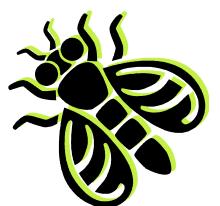
variable

Simply writing the wrong
variable name

```
x1 = transform(x1, x2, delta_x);  
y1 = transform(y1, x2, delta_y);
```



Slide 37

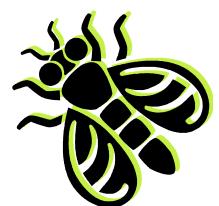


B – Blunder: Subcategories

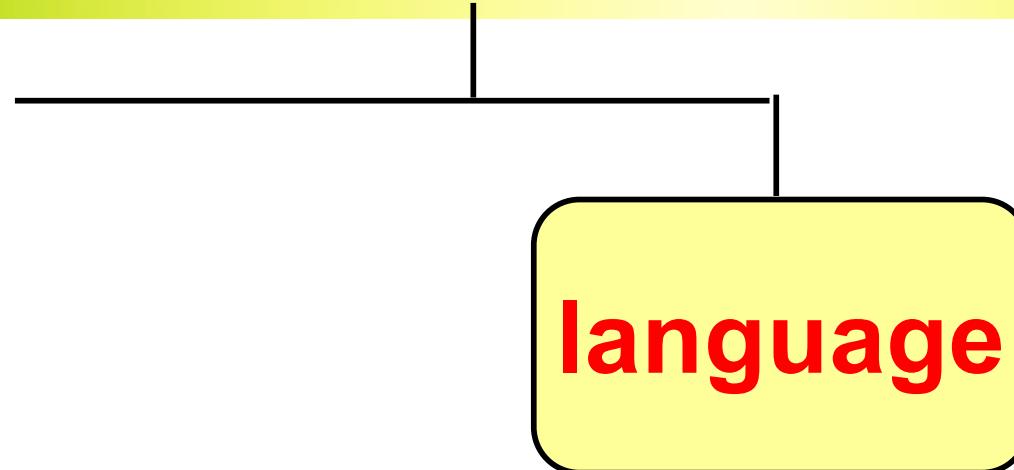
expression

An incorrectly written expression

```
if ((count < min) &&  
     (count > max)) ...  
||
```



B – Blunder: Subcategories



A mistake encouraged by
the language syntax

==
if (a ~~=~~ b) { . . . }

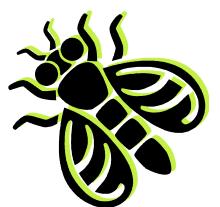


Some Bug Avoidance Strategies

Use a High-Level Language

and obtain all the advantages of

- strong type-checking at compile time,
- memory management,
- run-time checking of pointers, array indexes, arithmetic overflow, etc. (usually!).



Some Bug Avoidance Strategies

Think Before You Code

Depending on the size of the programming team and the size/scope of the project, you may want to:

- Write out and debug the specifications
- Plan the software architecture
- Design the data structures
- Develop pseudocode for the algorithms
- Prove the correctness of the algorithms



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas

Every subroutine should validate its inputs
(unless there is a clear *proof* that the inputs will always be valid)



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas

Insert assertions to verify that pointers are non-null and array indexes are in range (unless guaranteed by the programming language)



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas

Use library functions and library classes wherever possible (they have already been debugged!)



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas

Library functions often return codes to indicate exceptional conditions – be sure to check them!



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas

“Keep It Simple, Stupid!”

Avoid complicated logic in your programs; it will probably cost you more debugging time than it saves in execution time



Some Bug Avoidance Strategies

Write Robust Code – the strategies include:

validate

assertions

reuse

check results

KISS

watch out for gotchas

Many programming languages have common pitfalls, such as

`if (a = true) { ... }`

in C, C++, Java, C#.



Some Bug Avoidance Strategies

Test Your Code!

- Testing actually *finds* bugs rather than avoids bugs, but a failure which is found early (before releasing the program) and on a simple test case is much easier to analyze.
- Including test methods in classes (or test functions in modules) is a good idea.



Bug Detection – Step 1

- Study the failure symptoms to identify (if possible)
 - where in the program the fault occurred and
 - what the program state must have been.
- If necessary add an exception handler to the program (or use a debugging tool), to determine exactly where the error occurs.



Bug Detection – Step 2

- Read and re-read your code in the vicinity of the error location.
 - Better still, have *someone else* read your code.
 - Alternatively find a person who will listen and explain the program's logic to him or her.



Bug Detection – Step 3

- Experiment to find the simplest / shortest input which causes the failure to occur.
- Disable compiler optimization (if enabled).
- And try to develop theories about the program state which, if reached, would cause the observed failure.

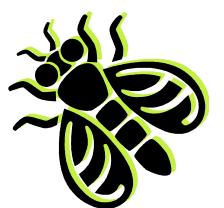
These preliminaries will make life much easier for the following steps.



Bug Detection – Step 4

We can try inserting ...

- assertions at strategic points to verify that an erroneous program state is not being entered, and/or
- output statements at strategic points to determine whether variables are taking on erroneous values.



Bug Detection

Perhaps the preceding steps have helped track down the error?

But maybe not ...

- There is too much trace output to examine?
- We didn't capture the right information?
- The bug went away when we inserted those extra statements????



The “Heisenbug”

- It is a bug which changes or goes away when you try to observe it.
- How can this happen?
- The addition of code changes the addresses of variables and subroutines, and may affect the positions of objects on the heap and times when a garbage collection occurs. An array indexing error or an invalid pointer may access a different data item and lead to different behaviour ...

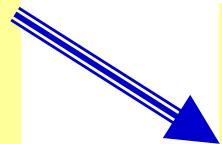


Heisenberg in 1927

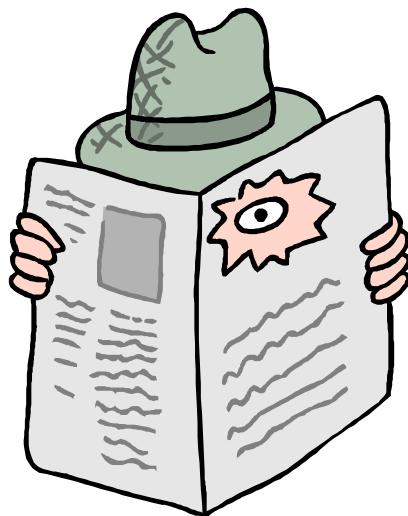


The Debugger

Sometimes we need to observe our bug in an unchanged copy of the program.



We need to have one program watch another program.



The watcher is called a *debugger* (and the program being monitored is the *debuggee*).



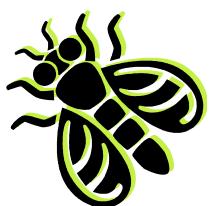
The Debugger

Two Common Tasks

analysis of
crash dumps

monitoring an
executing program to
watch it go wrong

Often, a single tool will perform either task



The Debugger – Note!

There is **no** guarantee
that the debuggee will
behave in exactly the same
way with a debugger



Perhaps it's a timing dependency?
Or perhaps the debuggee's memory
layout is different?



Some Debuggers

- Visual Studio [on Windows](#)
- dbx and gdb for debugging C/C++ ... [on Unix](#)
- eclipse for debugging [Java](#) (and other languages)
- jdb for debugging [Java](#)
- IDLE for debugging [Python](#)

- *insight* is a GUI front-end for for gdb
- *DDD* (Data Display Debugger) is a GUI front-end for gdb/dbx/jdb/...



Analysis of Crash Dumps

The debugging tool provides the ability to

discover where
the program
crashed and
why

inspect the call
stack and the
parameters

inspect the
values of
variables

Note: the first two features are provided by a stack
trace when an exception is caught in Java or C#.



From Static to Dynamic ...

Static Tools	Dynamic Tools
Source code analysis (to check compliance to coding standards, warn about possible problems)	Resource usage monitoring (CPU and memory profilers) e.g. jstat, jconsole, jprofiler, Prof-It, ...
Crash dump inspection	Debuggers



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support



Common Monitoring Facilities Provided by Debuggers

Breakpoints

```
(gdb) break main
```

Single step execution

```
(gdb) break 22
```

Inspect variables, location

Run program until a
breakpoint is reached

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

Inspect values of variables

Resume execution



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

step execute one statement and stop again

next execute one statement or a complete method/function call and stop again



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

(gdb) print sum

(gdb) where

(gdb) up



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

```
(gdb) list main  
(gdb) list encode  
(gdb) list 22
```



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

If the debugger throws an exception or performs some illegal action, the debugger must re-take control and give the user a chance to inspect the situation



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

watch sum

watches for accesses to a variable, breaks when value in variable is changed



Common Monitoring Facilities Provided by Debuggers

Breakpoints

Single step execution

Inspect variables, location

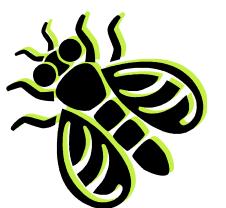
Display source code

Fault detection

Watchpoints

Multithread, multiprocess support

Ability to monitor execution of individual threads or processes in the debugger



Styles of Debuggers

Command-Line

- gdb and jdb are examples

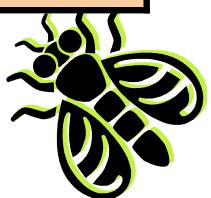
Graphical User Interface

- Visual Studio, eclipse, IDLE, gdb+DDD are examples



Epic Failures: 11 Software Bugs (ComputerWorld, 9 Sept. 2010)

1998	Mars Climate Orbiter didn't orbit
1962	Mariner 1 went off course
1996	Ariane 5 Flight 501
1994	Faulty Floating-Point Math in Intel Pentium
1990	AT&T Network Outage
2007	Windows 'Genuine Advantage' was buggy
1991	Patriot Missiles mistimed
1985-1987	Therac-25 Medical Accelerator
2001	Cobalt-60 Overdoses
2000	Osprey Airplane/Helicopter Crash
1983	Soviet Missile Defense Early-Warning System



Some Further Reading on Debuggers/Debugging

- Adam Barr. *Find The Bug*. Addison Wesley, 2005.
<http://www.findthebug.com/>
- E. Allen. *Bug Patterns in Java*. Apress, 2002.
- S.L. Bartlett, A.R. Ford, T.J. Teorey, G.S. Tyson.
Practical Debugging in Java. Pearson, 2004.
- J.B. Rosenberg. *How Debuggers Work*. Wiley, 1996.
- D.J. Agans. *Debugging*. AMA, 2002.
- M. Telles, Y. Hsieh. *The Science of Debugging*. Coriolis, 2001.

