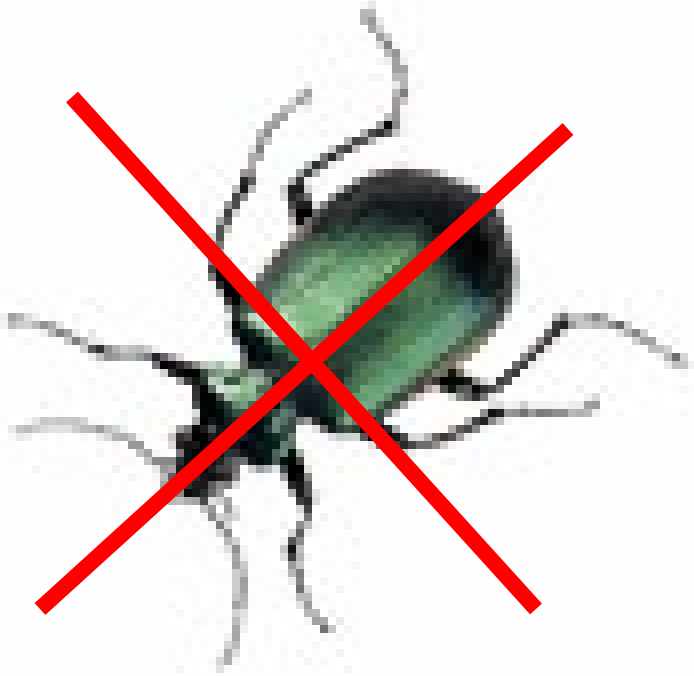


Debugging

Nigel Horspool

(with some material by Mike Zastre)

University of Victoria

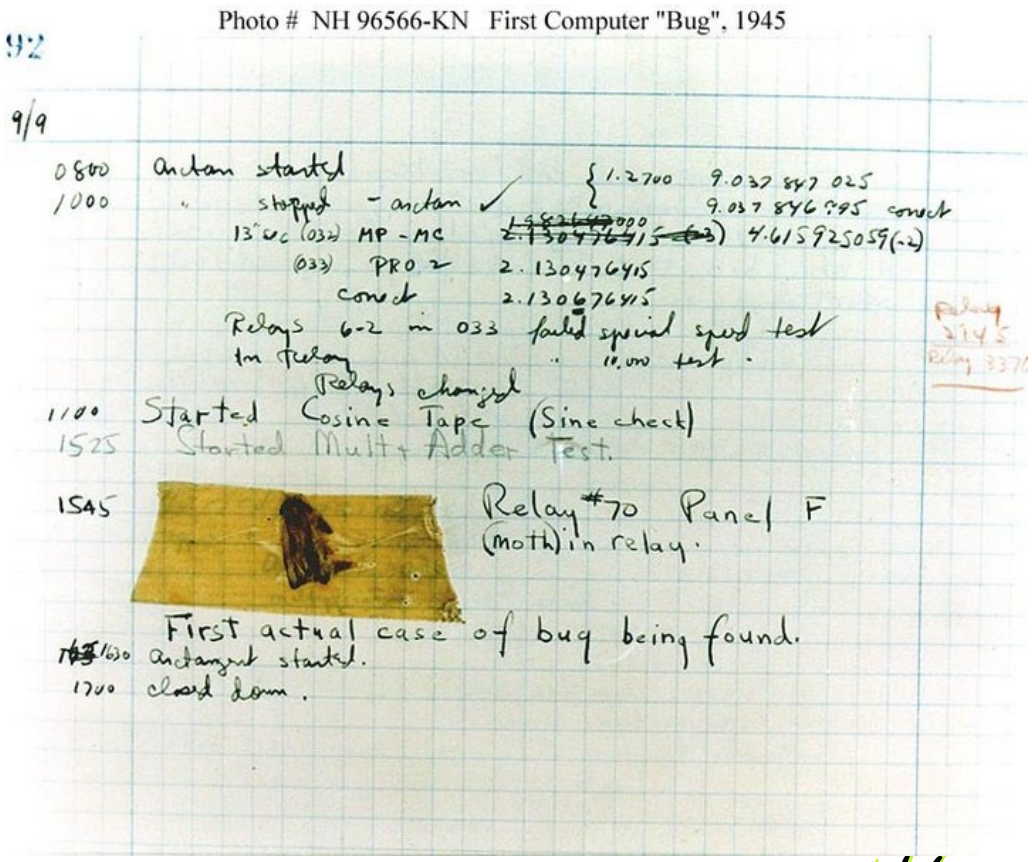
Topics

- Kinds of Programming Errors
- What is a Bug?
- Classifying Different Kinds of Bugs
- Bug Avoidance Strategies

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.)



Source: Wikipedia entry for *software 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	fault	error
something visible to the program's user	the state of the program which leads to a failure	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

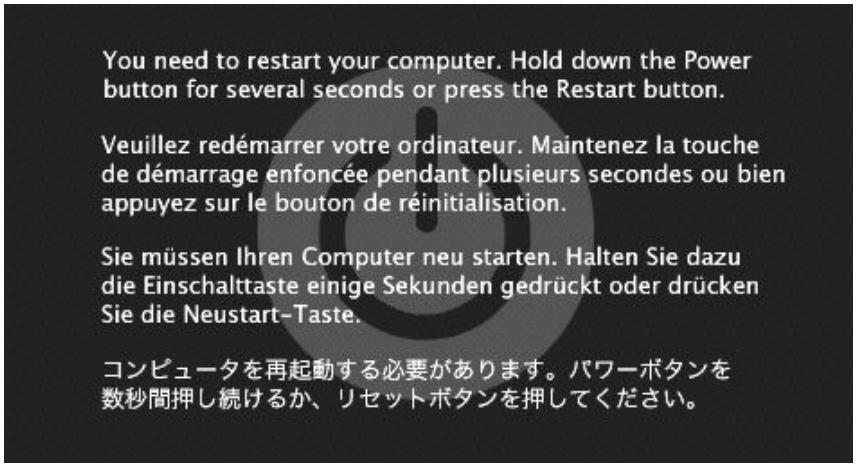
Some Possibilities for C/C++ or Assembler Code

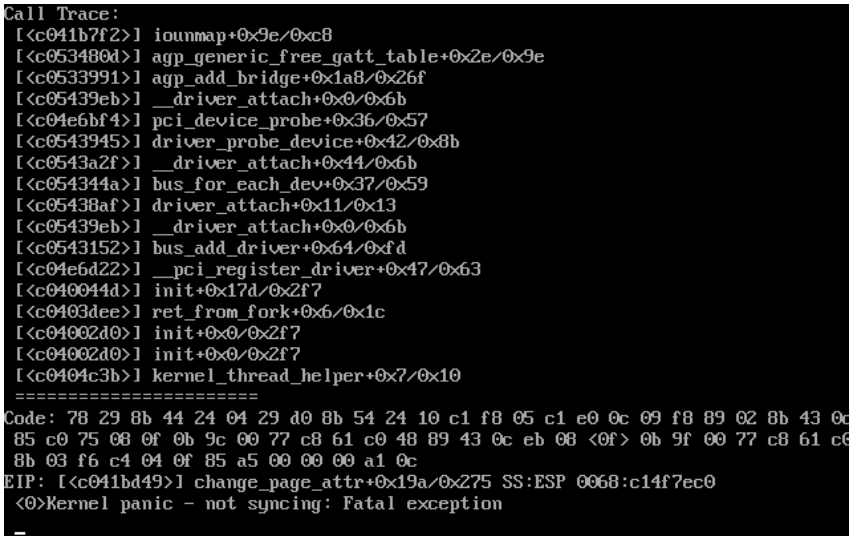
- “Blue Screen of Death”
- Segmentation fault
- Unaligned memory address
- Invalid instruction
- …

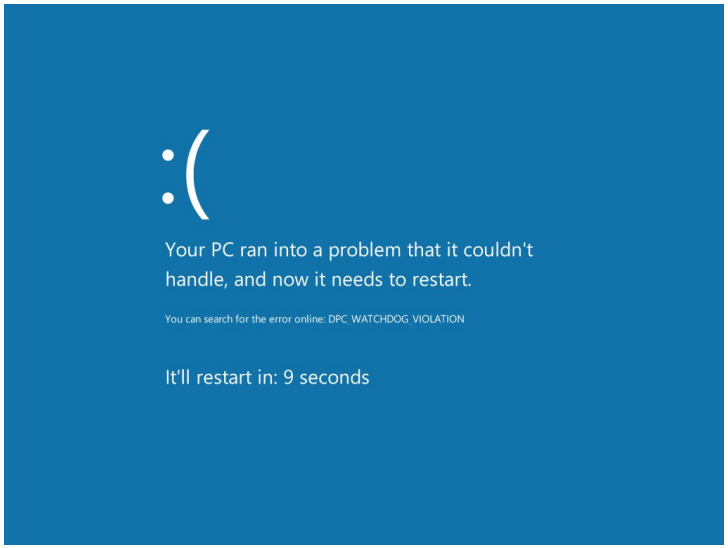
The Infamous Blue Screen of Death

Now just a memory!

But we have new BSOD flavours







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?

The program tries to access a memory location which is outside its accessible range

The CPU generates an interrupt

The O.S. intercepts it

The O.S. kills the program

Typically caused by an array index which is out of range, or by an uninitialized or invalid pointer variable

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;
}
```

How did we get to an Illegal Instruction?

The program has tried to execute an unassigned or privileged opcode

caused either by overwriting the text (instruction) area or by overwriting the return address from a subroutine – and that's usually due to a bad array index or pointer.

The CPU generates an interrupt

The O.S. intercepts it

The O.S. kills the program

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 <stlib.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!*

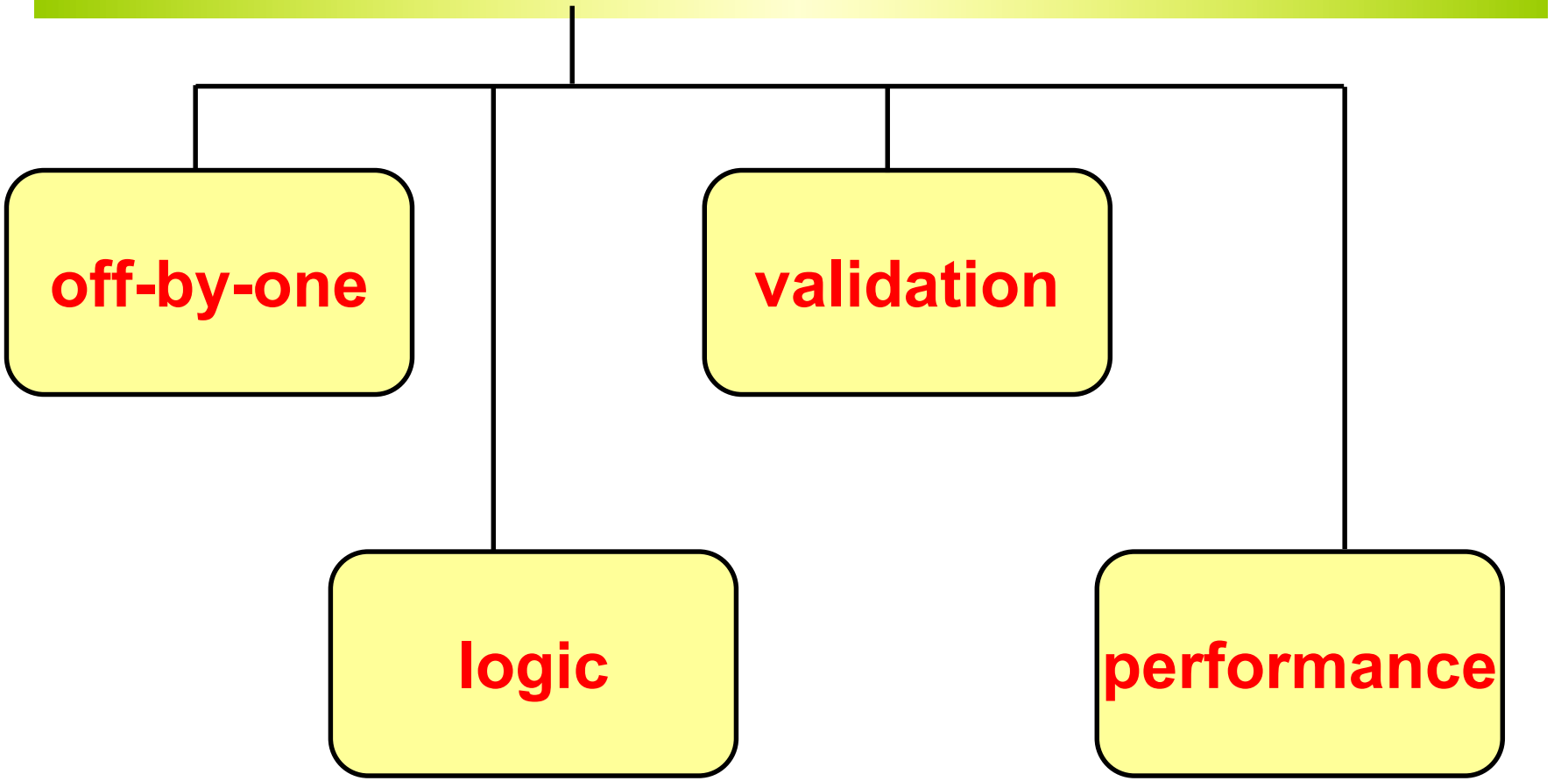
Bug Classifications

- 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;
}
```

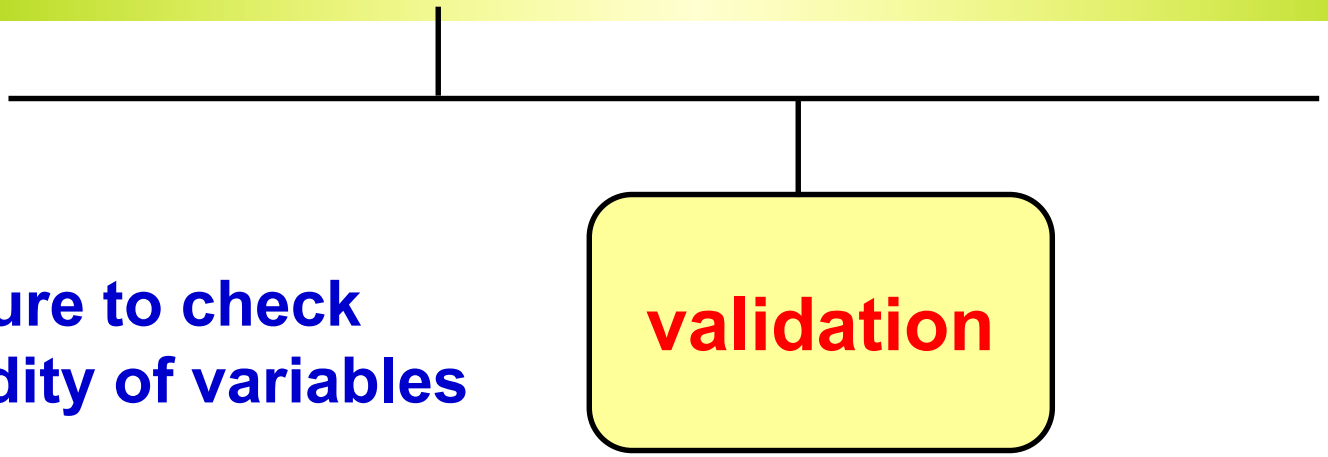
A – Algorithm: Subcategories

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

```
// 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;
}
```

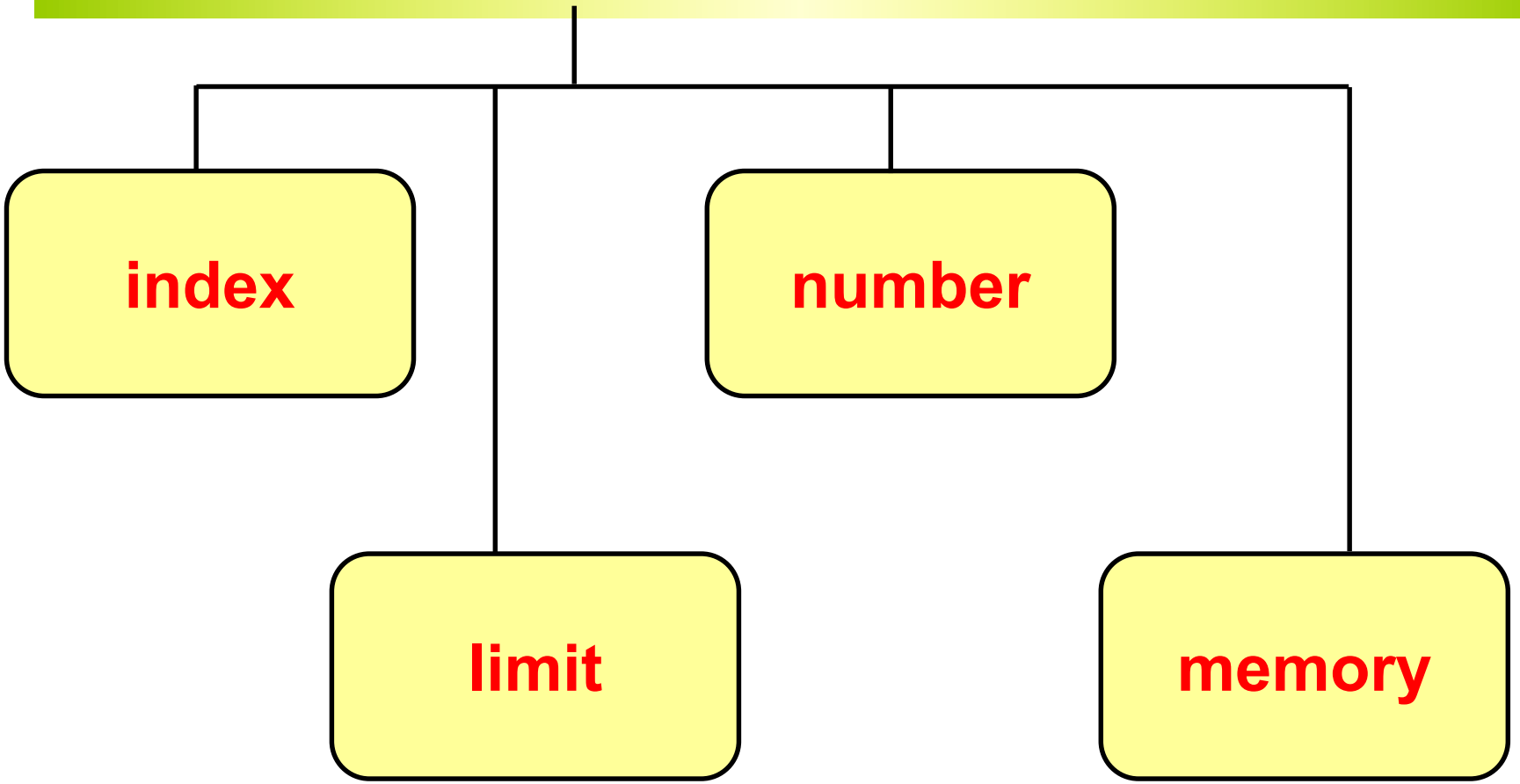
But what happens if n == 1 ?

A – Algorithm: Subcategories

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

performance

D – Data: Subcategories



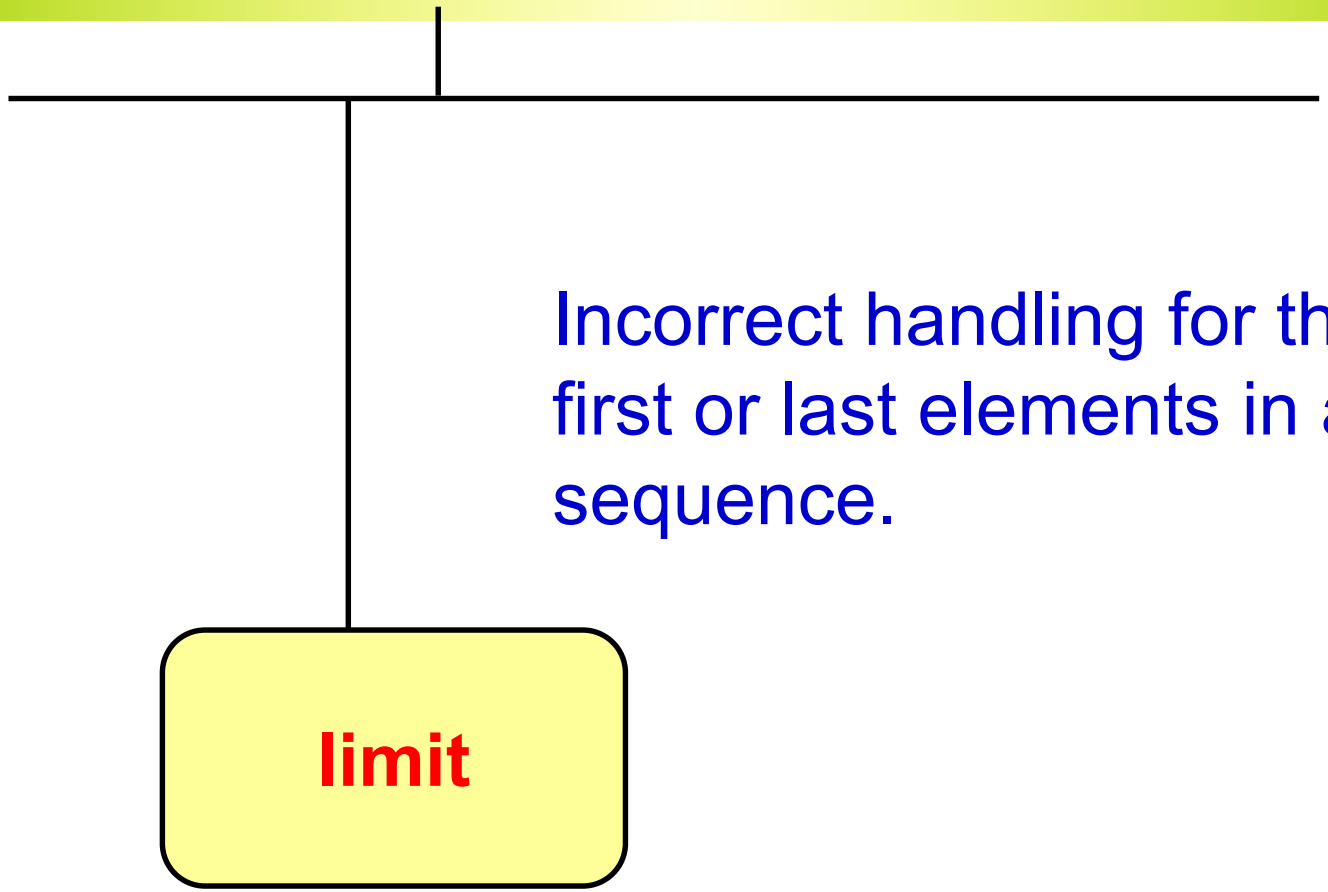
D – Data: Subcategories

index

An array index is wrong.

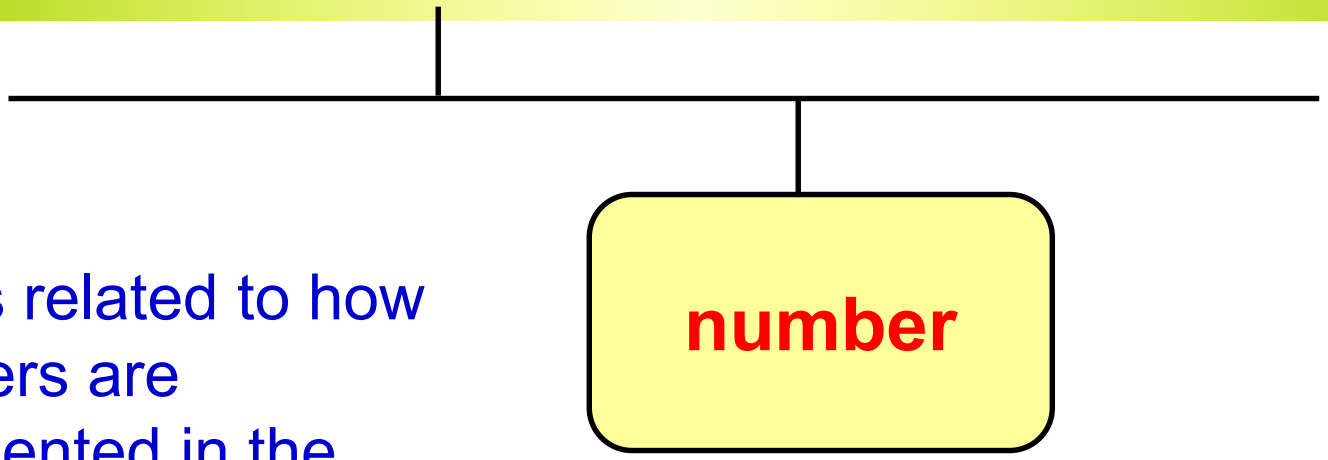
```
int search( int[] Array, int Val ) {
    int Hi = Array.length; int Lo = 0;
    while(Lo<=Hi) {
        int Mid = (Hi+Lo)/2;
        if (Array[Mid]==Val) return Mid;
        if (Array[Mid]<Val)
            Lo = Mid+1;
        else
            Hi = Mid-1;
    }
    return -1;
}
```

D – Data: Subcategories



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

D – Data: Subcategories



Issues related to how numbers are represented in the computer

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

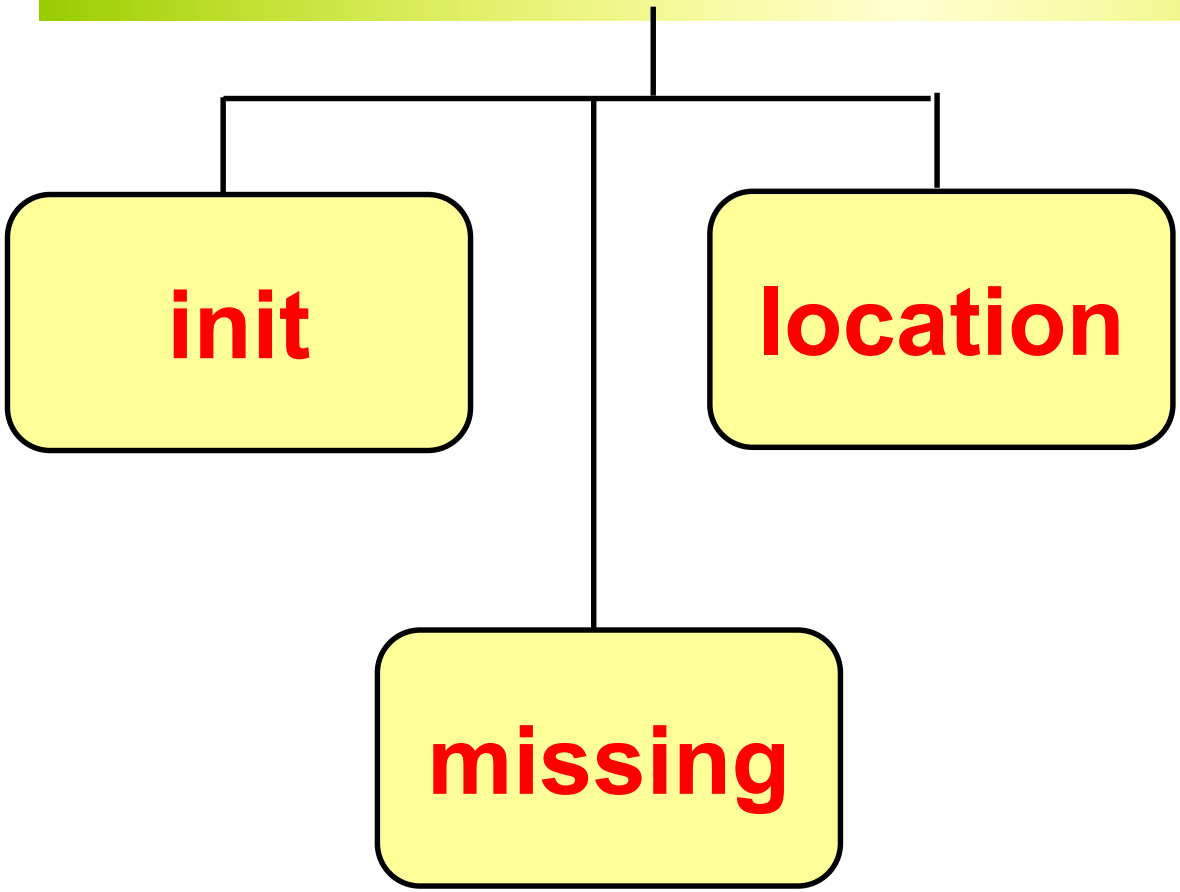
D – Data: Subcategories

Mismanagement of memory – such as

- attempting to access an inaccessible location,
- or deallocating 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



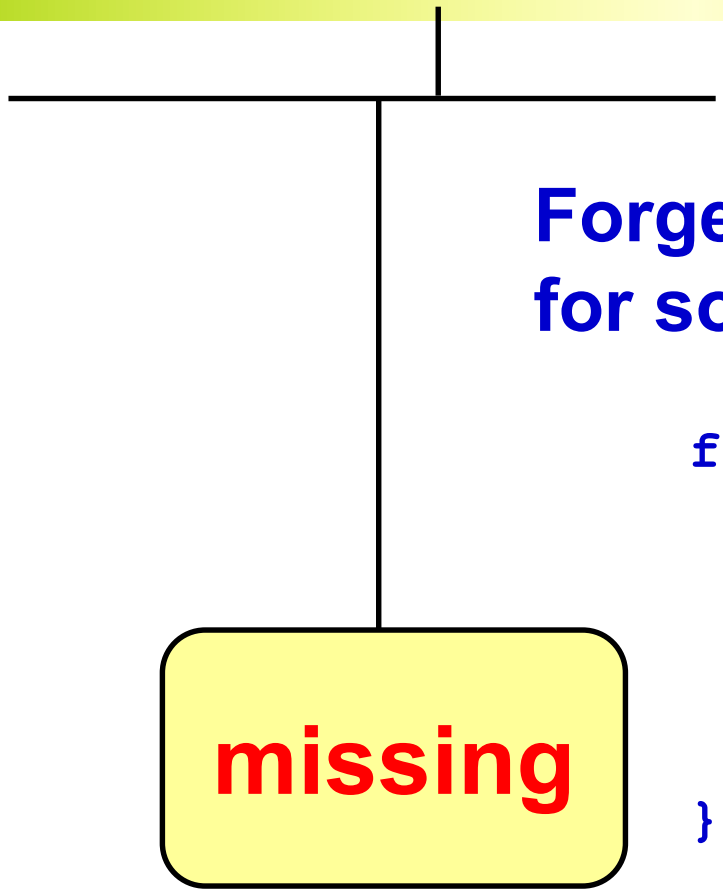
F – Forgotten: Subcategories

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];
}
```

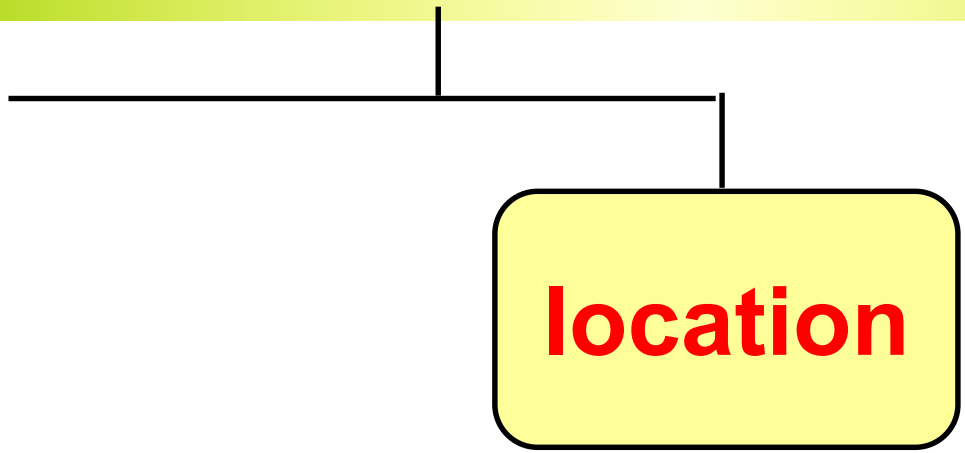
F – Forgotten: Subcategories



Forgetting to provide code for some case.

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

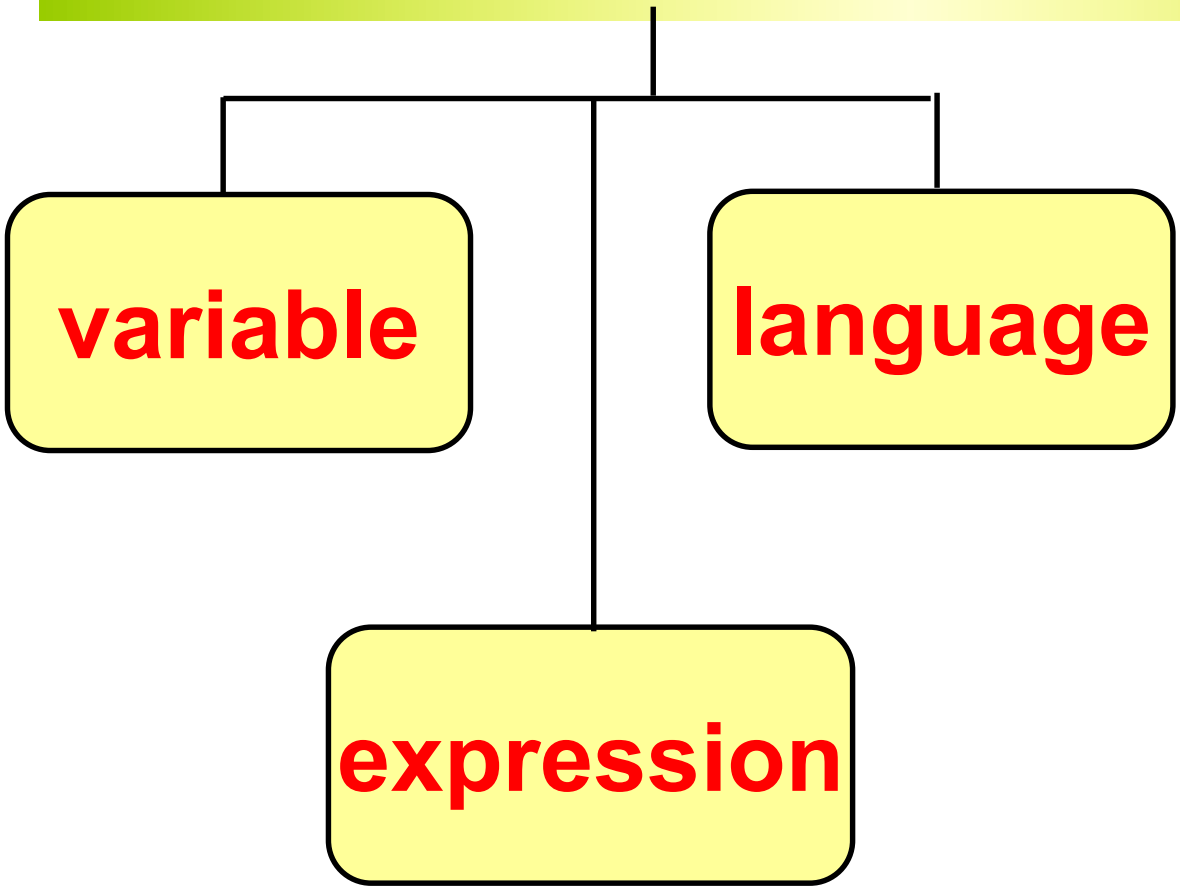
F – Forgotten: Subcategories



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

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

B – Blunder: Subcategories



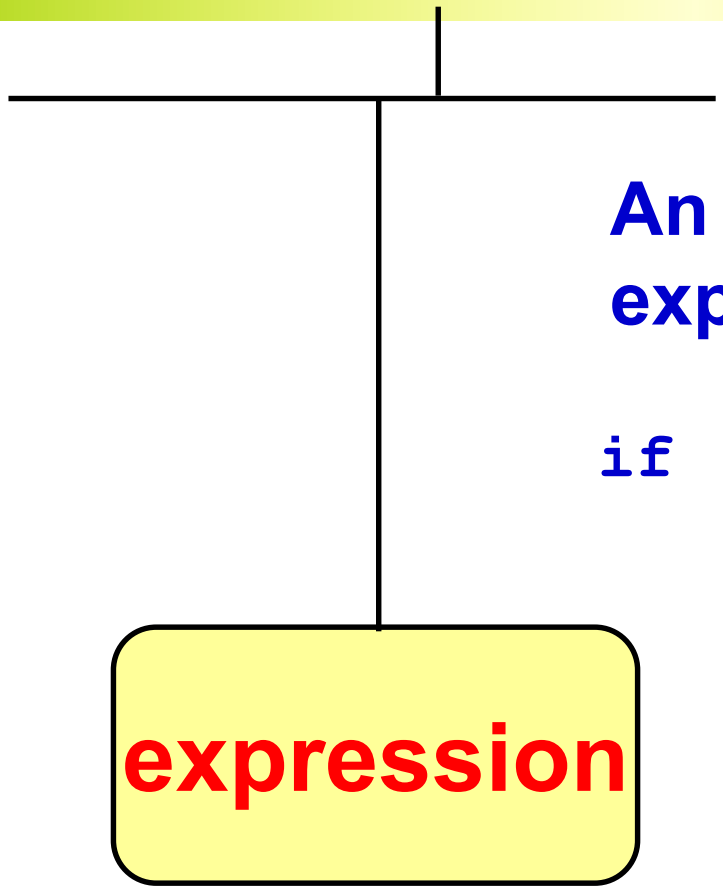
B – Blunder: Subcategories

variable

Simply writing the wrong variable name

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

B – Blunder: Subcategories

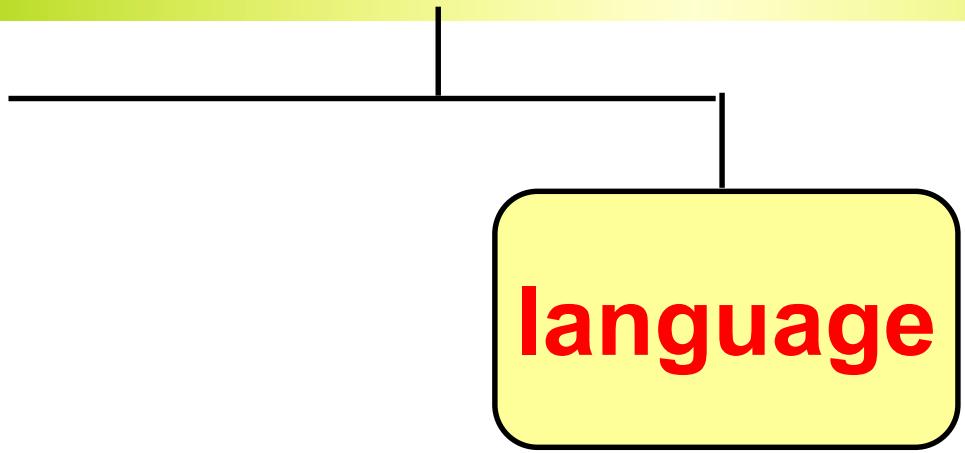


An incorrectly written expression

```
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.