Outline

- Monday: design, interfaces, representation of information
- Tuesday: testing, debugging, mechanization
- Thursday: programming style

Testing and debugging

- testing
  - orderly, systematic, exhaustive checks on whether a program is working properly
    - test as you write the code
    - test systematically
    - automate testing
  - testing process begins when program is designed
    - continues as program evolves
    - never ends
- debugging
  - efficiently figuring out what's wrong
    - never ends either
- mechanization
  - enlisting the aid of the computer

Test as you write the code

- many errors can be eliminated before they happen by careful coding

- check boundary conditions
- look for off by one errors
- state pre- and post-conditions and invariants
- add assertions
- defensive programming
- check error returns
- turn on all compiler checks

Boundary condition testing

```
    FUNCTION BINSEARCH(TABLE, N)
    INTEGER TABLE(10), LOW, HIGH, MID
    LOW = 1
    HIGH = N
10  MID = (LOW + HIGH) / 2
    IF (HIGH <= LOW) THEN
        BINSEARCH = 0
    ELSE
        IF (KEY == TABLE(MID)) THEN
            BINSEARCH = MID
        ELSE
            IF (KEY > TABLE(MID)) THEN
                LOW = MID + 1
            ELSE
                HIGH = MID - 1
            END IF
            GO TO 10
        END IF
    END IF
    END FUNCTION
```

Boundary condition testing

```
INTEGER FUNCTION BINSEARCH(KEY, TABLE, N)
    INTEGER KEY, TABLE(N), LOW, HIGH, MID
    LOW = 1
    HIGH = N
    BINSEARCH = 0
    DO WHILE (LOW <= HIGH)
        MID = (LOW + HIGH) / 2
        IF (KEY == TABLE(MID)) THEN
            BINSEARCH = MID
            EXIT
        ELSE IF (KEY > TABLE(MID)) THEN
            LOW = MID + 1
        ELSE
            HIGH = MID - 1
        END IF
    END DO
END FUNCTION
```

Lessons

- boundaries to check
  - no input (a form of defensive programming)
  - single element in the array
    - key low, high and equal
  - two elements in the array
- all paths through the code produce a value

- invariants
  - at the top of the loop,
    - limits are ok, key is within if anywhere
    - LOW <= MID <= HIGH
  - after testing KEY vs TABLE(MID), it matched or
  - if KEY > middle element
    - LOW is middle + 1 so search resumes in upper half
  - else
    - HIGH is middle - 1 so search resumes in lower half
  - the selected half does not include the former middle element
    - so the former middle element is not examined again

- compiler didn't check consistency of arguments or types!
  - my original revision was full of unnoticed errors

"Off-by-one" errors

- many errors are caused by being off by one:
  - too high or too low by a single position
  - relational test is wrong
  - code is in the wrong place
- result is often walking off the end of an array

```
int *array;

array = calloc(nmemb, sizeof(int));
for (loop = 0; loop <= nmemb; loop++)
    array[loop] = loop;
```

More "off-by-one" errors ...

- putting code in the wrong place:

```
      SUM = 0.0
      COUNT = 0.0
C ADVANCE COUNTER
    4 COUNT = COUNT + 1.0
      I = COUNT
      READ (5, 100) X(I)
C CHECK FOR END OF FILE
      IF (X(I) .EQ. TEST) GO TO 9
      SUM = SUM + X(I)
      GO TO 4
C COMPUTE THE MEAN
    9 AVG = SUM / COUNT
```

Defensive programming

- assume the worst about input, and defend against it

```
get_num() {
   char s[80], *temp = s;
   do {
       *temp=getchar() /* read a digit */
       if (isdigit(*temp)) temp++;
   } while (*(temp-1)!='\r'); /* until return */
   *temp='\0'; /* null terminate */
   return(atoi(s));
}
```

- be able to handle careless or malicious input
- never generate careless output

Defensive programming

```
get_num() {
    char s[80];
    int i, c;

    for (i = 0; i < sizeof(s)-1; ) {
        c = getchar();
        if (c == '\n' || c == '\r' || c == EOF)
            break;
        if (isdigit(c))
            s[i++] = c;
    }
    s[i] = 0;
    return atoi(s);
}
```

Other checks

- pre- and post-conditions, and invariants
  - what is assumed on entry?
  - what is true on exit?
  - what remains unchanged?
  - does the code do the right thing?
- assertions
  - state what has to be true at specific point

```
assert(low <= mid && mid <= high)
```

- check error status when a subroutine or function is called
  - file close will return an error if there was any write error
    - detects disk full or over quota
  - always check returns from storage allocation
- turn on all compiler warnings
  - though not all compilers do a good job
- do code reviews
  - read someone else's code to look for potential problems

Systematic testing

- test incrementally
  - test each new feature as implemented
  - test each new code as written
- test simple parts first
  - basic features before fancy ones
- know what output to expect
  - easiest if test inputs and outputs are in files and can be run automatically
- verify conservation properties
  - preserves number of items, sizes of data structures, order relationships, etc.
- compare independent implementations
  - e.g., base 64 converter vs openssl
  - e.g., awk vs gawk
  - e.g., different compilers
- measure test coverage
  - use compiler to generate coverage information
  - add test cases to go through each part of code
  - it's hard to achieve high coverage

Sanity testing of hash table

```
int Array::sanity(char *msg)
{
   int n, i;
   Aelem *ep;

   for (i = n = 0; i < _size; i++) {
      for (ep = _base[i]; ep != 0; ep = ep->anext) {
         n++;
         if (!member(ep->aname)) {
            fprintf(stderr, "sanity failure: %s\n",
               ep->aname.c_str());
            return 0;
         }
      }
   }
   if (n != _count) {
      fprintf(stderr,"sanity fail %s: size %d %d\n",
         msg, n, _count);
      return 0;
   }
   return 1;
}
```

Mechanization: let the computer do the work

- write code by programs
  - compilers, compiler compilers, interface builders, wizards
  - specialized languages: regular expressions, ...
- use executable specifications
  - avoid writing the same information multiple times
  - capture the varying part in specification files
- do clerical jobs by programs
  - build, test, system administration, ...
- test programs by programs
  - hand testing is too slow and prone to error

- when the job is mechanical, it's time to mechanize

Test automation

- test scaffolds
  - shell scripts, programs, makefiles, etc., that run tests and compare results automatically
- automated regression testing
  - does the new version get the same answer as the old version?
  - if it doesn't, explain why
- self-contained tests
  - compare to known output or independently created outputs
  - interoperability
- stress tests
  - large inputs
  - random inputs
  - perverse inputs

- testing base64 encode / decode programs
- testing hash tables
- testing AWK

Base 64 encode/decode testing

- encoder: takes arbitrary input into ASCII in 6-bit char set
  - each 6 bits in becomes 8 bits out
- decoder: reverses the encoder
- official standard (RFC 2045) for how to do it
  - your code has to interoperate with other people's code

- boundaries:
  - input lengths 0, 1, 2 mod 3
  - inputs with high order bits turned on
    - because there's a lot of shifting and masking in the code
    - and sometimes people only test on character input
- test files of length 0, 1, ... 9 are probably enough
  - but check a handful of very long files too

```
for i in *     # test decoder
do
      echo $i:
      openssl enc -e -base64 <$i >foo.$i
      java decode <foo.$i >glop.$i
      if cmp -s $i glop.$i
      then
           rm foo.$i glop.$i
      else
           echo BAD: $i
      fi
done
```

Hash table tests

Test your code as thoroughly and as mechanically as you can. My approach was to write a test driver that accepts a sequence of command lines like this, one for each member function:

```
n size          new array A of size size
p name val      A[name] = val
g name          print A[name]
m name          print 1 or 0 if A[name] exists or
c               print count of elems
s               print size of the array (hash tab
d name          delete A[name]
D               delete the array
G               double the table size
```

I used this to do basic checking by hand. Then I wrote an Awk program (naturally) to generate random sequences of such commands. Other Awk programs generated more systematic command sequences. I ran all those commands to make sure that internal checking in my C++ code didn't run into sanity errors. Finally, I wrote another Awk program to read the same input, do the same operations, and produce what ought to be the same output (modulo inescapable differences like the lack of **grow** and the random order of hash table outputs); a shell script ran the two versions and reported on unexpected differences in the outputs. In this way, I could do volume tests.

AWK

- a language for pattern scanning and processing
  - Al Aho, Brian Kernighan, Peter Weinberger
  - Bell Labs, ~1977
- intended for simple data processing
- selection, validation:
  - "Print all lines longer than 80 characters"

```
length > 80
```

- transforming, rearranging:
  - "Replace the 2nd field by its logarithm"

```
{ $2 = log($2); print }
```

- report generation:
  - "Add the numbers in the first field, print the sum and average"

```
{ sum += $1 }
END  { print sum, sum/NR }
```

Basic AWK programs:

```
{ print NR, $0 }                        precede each line by line number
{ $1 = NR; print }                      replace first field by line number
{ print $2, $1 }                        print field 2, then field 1
{ temp = $1; $1 = $2; $2 = temp; print }        flip first two fields
{ $2 = ""; print }                      zap field 2
{ print $NF }                           print last field

NF > 0                                  print non-empty lines
NF > 4                                  print if more than 4 fields
$NF > 4                                 print if last field greater than 4

NF > 0      {print $1, $2}              print two fields of non-empty lines
/regexpr/                               print matching lines (egrep)
$1 ~ /regexpr/                          print lines where first field matches

END { print NR }                        print line count

      { nc += length($0) + 1; nw += NF }        wc command
END { print NR, "lines", nw, "words", nc, "characters" }

$1 > max { max = $1; maxline = $0 }             print longest line
END      { print max, maxline }
```

AWK test strategy

- regression tests that compare previous version to current version
- absolute tests against known answers or independently computed results
- stand-alone tests for
  - language features in isolation, representative small programs, bigger complete programs
  - specific language areas: expression evaluation, regular expressions, ...
  - test cases for identified bugs
  - new tests for new features
  - stress tests, boundary tests, coverage tests, error messages
  - timing of basic operations for performance monitoring
- test data:
  - realistic data
  - boundary conditions
  - random input
  - high volume input
  - illegal inputs

AWK testing strategy

- ~1000 tests in test suite
- running them all takes a single command
  - output is just a progress report unless something breaks

- never ignore a bug
- maintain a record of all bugs and fixes
- add a new test for each bug found
- never delete a test

- re-examine the tests occasionally
  - you may not be testing what you think you are

Record-keeping

- record of all bug fixes since August 1987
  - Nov 22, 2003: fixed a bug in regular expressions that dates (so help me) from 1977; it's been there from the beginning. an anchored longest match that was longer than the number of states triggered a failure to initialize the machine properly. many thanks to moinak ghosh for not only finding this one but for providing a fix, in some of the most mysterious code known to man.

  - fixed a storage leak in call() that appears to have been there since 1983 or so -- a function without an explicit return that assigns a string to a parameter leaked a Cell. thanks to moinak ghosh for spotting this very subtle one.

- and some not yet fixed:

  "Consider the awk program:

  ```
  awk '{print $40000000000000}'
  ```

  which exhausts memory on the system. This actually occurred in the program:

  ```
  awk '{i += $2}
  END {print $i}'
  ```

  where the simple typing error crashed the system."

Using AWK for testing RE code

- regular expression tests are described in a very small specialized language:

```
^a.$ ~   ax
         aa
     !~  xa
         aaa
         axy
```

- each test is converted into a command that exercises the new version:

```
echo 'ax' | a.out '!/^a.$'/ {print "bad"}'
```

- illustrates
  - little languages
  - programs that write programs
  - mechanization

Basic debugging

- debugger or print statements?

- look for familiar patterns
  - pointer scanf arguments
  - wrong I/O format conversions
  - mismatched / out of order arguments
  - inconsistent types
  - getchar into a char
  - order of evaluation and side effects
  - faulty initialization
  - pointer dereferencing
  - dangling pointers
  - references: assignment is not copying
  - 0 origin or 1 origin arrays?
  - ...

familiar patterns ...

```
   #include <stdio.h>

   int main() {
        int n;
        scanf("%d", n);
        printf("%d\n", n);
   }
```

```
% cc bad.c
% a.out
123
Segmentation fault (core dumped)
```

```
% gcc bad.c
% a.out
123
-4196932
1
Bus error (core dumped)
```

familiar patterns (2) ...

```
   function foo(n)
      foo = n
   end function foo

   program bad
      print "(i5, 1x, i5)", 5, foo(5)
      stop
   end program bad
```

```
% f90 -free bad.f
% a.out
    5 *****
```

familiar patterns (3) ...

```
factorial(int n) {
    int result;
    result = 1;
    while (n--)
        result *= n;
    return result;
}
```

```
n = 0;
while (i < n)
    a[i] = b[i++];
```

```
char *f() {
    char buf[100];
    ...
    return buf;
}
```

Tactics

- look at the most recent change first
  - "I only changed one thing"
- fix all similar places
- debug it now: fix all known errors
  - turn on all compiler warnings
- use the stack trace
- read the code carefully
  - on paper, not on the screen
- explain the code to someone else

Harder cases

- make the bug reproducible
- divide and conquer: make small cases that illustrate the problem
- numerology
- print “got here” to see where flow of control goes
- write self-checking code
  - assertions, consistency checks, conservation
- produce log files with debugging output
- draw a picture
- use tools
- keep records

Practice...

- symptom: infinite loop, infinite output

```
nullFirst = ((temp = bis.read()) == -1);
while (!readAllInput && !nullFirst) {
   for (int i = 0; i < 4; i++) {
      if (Character.isWhitespace((char) temp))
         continue;
      temp = translate(temp, charMap);
      if (temp == -1)
         return;
      in[i] = temp;
      if ((temp = bis.read()) == -1)
         readAllInput = true;
   }
   // irrelevant part omitted
   for (int i = 0; i < 3; i++) {
      if (out[i] != -1)
         System.out.print(
            Character.toString((char)out[i]));
   }
}
```

Hardest cases

- memory allocation problems
  - fences, checkers, audit trails, ...
    `0xDEADBEEF`
- non-reproducible bugs
  - Heisenbugs: trying to look at them makes them go away
    `assert(*p++ == NULL);`
- mental model is wrong
  - blind spots
  - language misunderstanding
  - violation of abstractions: something using a back door
- compiler or library errors / language problems
  - flush in Java
  - signed vs unsigned
  - Borland isprint bug

How to report a bug

- make sure you have the latest version
- make sure you're using it correctly
- find the smallest case that fails
- send all the information
  - version
  - compiler, operating system, ...
  - small inputs that fail
  - corresponding outputs

A good bug report

On Tue, 17 Feb 2004, Brian Tsang wrote:

> In case you're still maintaining awk, this program seems to freeze the program even after awk prints out the details of the syntax error. I compiled using the Bell Labs distribution on hats and on my own computer (cygwin). Minimal excerpt from test/switch2.awk in the gawk3.1.3 distribution:

```
BEGIN {
    switch (substr("x",1,1)) {
    case /ask.com/:
      break
    case "google":
      break
    }
}
```

- a really minimal case:

```
{
    s {
    c /./
    }
}
```

Conclusions

- "Program testing can be used to show the presence of bugs, but never to show their absence!"
  - Edsger Dijkstra