Defensive Programming

CIS*2750

Advanced Programming Concepts

Series on Testing

1. Defensive programming
   - *Detect problems as early as possible*
2. Intro. to testing
   - *Classic testing methods for “finished” code*
3. Debugging
   - *Fixing defects uncovered by testing*

1. Defensive Programming

It's like "Defensive Driving"

- You *expect* "unexpected" problems to crop up
  - her turn is signal on, but car doesn't turn
  - you brake, but he doesn't slow down behind
  - it's "deer season," so you scan the roadside forest

- Some areas for "suspicious" programming:
  - input data, not as specified
    - function arguments, file contents, human input
  - module behaviour, not as specified

Need Error Handling Strategy

- Defensive programming uncovers “errors”
  - **Now, what are you going to do about them?**

```
if (error happened) {
      what code goes here? }
```

- Best to have thought-out **strategy** re errors
    - Don’t leave up to individual coder’s judgment
    - Don’t handle on ad hoc basis
  - Otherwise, code quality (re error checking & handling) will vary widely across system

See “Good Practices” on Assignments Webpage

1. If the spec calls for some particular treatment…
2. If you run into the problem while trying to complete the processing…
3. If you complete the function's processing without finding a problem…

- This is example of “error handling strategy” suitable for student assignments

Striking a Balance

- *One extreme:*
  - Check for every conceivable (and inconceivable) error condition
  - Downside:
    - Costs time/money to write that code
    - Error checking/handling code must be tested, too!
    - Takes up memory with code never likely to run
- *Other extreme:* “nothing will go wrong!”
    - Fragile system, late discovery of hard-to-locate bugs

Analyzing Possible Errors

- Errors are not all the same
  - They have different characteristics → “kinds” of errors
  - They have different “severities” → affects whether continuation is smart or even possible
- Treatment may depend on phase of SW lifecycle
  - Are you debugging the program?
  - Is it running in production?

Recognize *Three Kinds* of Errors

1. Problems with **external** data/conditions
   – Outside the control of your program, not your fault
   – User/operator should be informed; bad data should never crash the program! (seg fault)

2. Erroneous **internal** (to your SW) usage, and
3. **rare** conditions (lumping these together)
   – *These could be your SW's fault!*
   – Module B detects that module A called it with bad args
   – Out of memory/disk space
   – Unexpected error return/result from library func call

Recognize *Two Severities*

1. "Fatal" errors:
   - Meaningless for program to continue execution
     - e.g., out of memory, nonsensical value of critical variable
   - Best to abort, and inform if possible
     - Otherwise, how will anyone diagnose the problem?
2. "Nonfatal" errors (recovery may be possible):
   - ***Testing/debug phase:***
     - Let programmers know before **aborting** (e.g., `abort()`)
   - ***Production phase:***
     - Keep running, recover gracefully if possible

- Some add "warning" or "informative" severities

After Detection, Who Handles?

- General principle:
  - **Handle** errors in context, in the same place where you **detected** them

- You're aware an error may occur, might as well write the code to handle it *while you know what the problem is*

Benefits of Local Error Handling

- Avoids letting invalid state **propagate** elsewhere through other modules
- Self-documenting:
  - Clear idea what the (usually complex) error handling code is supposed to be there for
- Keeps the complex code for particular error **contained and localized** instead of smeared throughout the system

But there are exceptions…

When to Reflect Errors Upward

- **Utility** packages are exception! (calutil)
  - Can *detect* errors, but may not know how to *handle* in way acceptable to application
    - E.g., utility printing error message usually not appropriate
    - App. may have special use of stdout/stderr streams, may be running in GUI
- Best: reflect error status up to caller
  - *Caller* applies "handle error in context" principle to suit nature of application

Tools for Reporting Errors

Unconditional report, allow recovery/continuation

- fprintf(stderr, …);
  - nonbuffered stream, crash won't prevent output
- syslog: uses OS event logging facility
  - #include <syslog.h>; "man openlog" for help
  - see Linux Programming, pp. 148 (2/E); 162 (3/E)

Conditional report: print & abort or continue quietly

- assert(*expression*);
  - #include <assert.h>; "man assert" for help

Using Assertions

assert(*expression*);

- *expression*==0 (meaning “false”) → assertion fails, prints message on **stdout** and calls **abort()**
- Good way to handle *improbable* conditions likely arising from *bugs* where you just want to abort
  - verifying **preconditions** in functions (misuse by caller)
  - verifying malloc not NULL (allocation gone wild)
    - We’re using for malloc/calloc/realloc, strdup, etc.
  - DON’T use for testing input data
  - see Linux Programming, pp. 340-341 (2/E); 439 (3/E)

Turning Assertions On/Off

- Turned **on** by default
  - helpful printout: source file, line no., function name
  - aborts program
- To turn **off** at compile time:
  - insert line: #define NDEBUG
  - makefile: gcc … -DNDEBUG
- On/off is a matter of *policy* in context of application-wide error handling *strategy*
- Since could be either on/off, assertion must not have side effects! =any code that must be executed

When will this code fail?

Utility

```
double avg( double a[], int n) {
  int i;
  double sum = 0.0;
  for ( i=0; i < n; i++ )
    sum = sum + a[i];
  return ( sum/(double)n );
}
```

```
#include <stdio.h>
#include <stdlib.h>
double avg ( double[], int );
int main ( int argc, char *argv[] ) {
  int count = argc - 1;
  double array[100];
  int i;
  for ( i=1; i <= count; i++ )
    array[i-1] = atof(argv[i]);
  printf("Avg=%f\n",avg(array,count));
}
```

When will this code fail?

> testzero 1 2 3.4 45.6 78.9

Avg=26.180000

> testzero

Avg=NaN

So “n>0” was an unstated precondition of the avg() function!

Using Assertions

- In testzero.c, avg function:

```
#include <assert.h>
  assert( n>0 );
```

- Running with no arguments:

```
testzero
testzero.c:15: avg: Assertion ‘n>0’ failed.
```

- Is this a good use of assert?
  - *Yes:* verifies precondition of avg
  - Catches **bug** in main(): main *should* check for 0 args
  - Production phase: could choose to turn off assertion

Bad Use of Assertion

- Can main() check for 0 args with assert()?
  - *No:* bad use!
    - 0 args is due to external input
    - Contrast violation of internal utility's precondition
  - "Assert" not substitute for input validation code
  - *Best:* print message telling user to type at least one number

Smart/dumb use of assert

SMART

```
int *arr;
arr = malloc(20*sizeof(int));
assert( arr != NULL );      // states what condition should be true
assert( arr );              // same meaning, NULL ptr. compares equal to 0 (false)
```

DUMB

```
int *arr;
assert( arr = malloc(20*sizeof(int)) );        // saved a line of code!
```

- If assertions are disabled (-DNDEBUG), the <u>second malloc</u> will disappear!

What to Check For

- External inputs:
  - Check for cases that “can't occur” just in case!
  - Check for extreme values -- negative values, huge values, etc.

```
if ( grade < 0 || grade > 100 ) /* impossible grades! */
    letter = '?';
else if ( grade >= 80 )
    letter = 'A';
else   ...
```

- Internal: Check for NULL pointers, out of range subscripts, default switch case, divide by zero

What to Check For

- Check pre- and post-conditions
  - Before and after a critical piece of code, check to see if the necessary pre- and post-conditions (value of variables, etc.) hold true
- Famous divide by zero error:

  On the USS Yorktown (guided-missile cruiser), a crew member entered a zero, which resulted in a divide by zero, the error cascaded and shut down the propulsion system and the ship drifted for a couple of hours.

What to Check For

- Check error returns
  - Don’t take for granted that status is OK
  - Always look at the values returned from library functions and system calls (see man pages)
  - Remember to check for output errors and failures as well as input failures - fprintf, fwrite
- Bad status: print text message, *not* error no.
  - “man” strerror function, errno global variable

On to Testing!

- Defensive programming = some degree of self-testing code
  - Tests itself at development time
  - Continues testing in production
- Still need to methodically apply tests to modules, system as whole
  - Practices very well developed in SW engr.
  - Can only introduce in 2750, more in 3750/3760
  - We have a 4th year testing course