Defensive Programming
How to Finish Your Project before 2109

Robert McLay

The Texas Advanced Computing Center

August 24, 2009
Outline

- Introduction
- Defensive Programming
- Language Specific Defenses
- Design with Testing in Mind
- Conclusions
Why?

• Programming is complicated.
• It’s impossible to hand check trillions of calculations. $10^6$ secs $= 11.5$ days, $10^9$ secs $= 31.7$ years, $10^{12}$ secs $= 316.9$ centuries, $10^{15}$ secs $> 31$ million years.
• You’d like to have time to enjoy other things besides working at the computer.
More on Why?

- Answers Matter.
- NOAA Researchers used Ranger to predict hurricane Ike’s landfall.
- Other Researchers predict 100 yr flood plain using computer models.
More on Why (II) ?

- This talk is mainly geared toward developer-users of computing.
- For users: it is what to ask from your developers.
- In Texas: Traffic Violation → Defensive Driving Class.
Conservation of Answer

- Physics: Conservation of Energy, Momentum.
- Computation Science: Conservation of Answer.
- $3 \times 2 = 2 \times 3 = 2 + 2 + 2 = 6$
- Use this fact: We expect to get the same answer today that we got yesterday, last week.
• Introduction

• Defensive Programming

• Language Specific Defenses

• Design with Testing in Mind

• Conclusions
Defensive Programming

• Save time by using Good Practices.
• Use tools. Do not repeat every mistake that others have already made.
• Save time to make your own unique one.
Foundation of Defensive Programming

- Build system: Make, Scons, Bjam.
- Source Code Management: CVS, SVN, Git.
- Design with Testing in Mind.
- DRY: Do not Repeat Yourself
Build Systems

- Use a build system like: Make, Scons, Bjam.
- If you use Make then only use Gnu Make.
- It is portable (Unix and Windows).
- It is Powerful.
- Good Documentation is available.
Your build system should:

- Do more than just build the program executable.
- It can run your tests.
- Easily build Optimize/Debug/Max Debug version of your code.
- Help with Source Code Management.
- What other things can it do to win friends and influence people?
Optimize/Debug executables

• Have your build system use an env. var. to control Opt/Dbg/Mdbg
• Opt: Optimal performance (safety locks off)
• Dbg: Debug compilation (-g -O0)
• Mdbg: (F90) out-of-bound array, uninitialized scalars, floating point exceptions
• Mdbg: (C++:g++,intel): -D_GLIBCXX_DEBUG
  -D_GLIBCXX_DEBUG_PEDANTIC: STL array bounds checking
• Remember to use “opt” for production runs.
QA: Record Version, Build time in Executable

- Have the build system generate a file that contains:
  - Opt/Dbg/Mdbg
  - Program Version number
  - Build date

- Can use strings on executable.
- Write to log file.
- Store in solution files.
Make Example

VDATE := $(shell date +'%F %H:%M')
F90_VERSION := $(shell ./report_compiler_version $(FC))
build.F90:
  @echo "! -*- f90 -*-" > $@
  echo "module build" > $@
  echo ' character(80), parameter :: OPT_LEVEL = "'$(OPTNAME)'"' >> $@
  echo ' character(80), parameter :: BUILD_DATE = "'$(VDATE)'"' >> $@
  echo ' character(80), parameter :: F90_VERSION = "'$(F90_VERSION)'"' >> $@
  echo 'end module build'

• Source in ~train00/defensive_progs.tar.gz
• Complete example in defensive_progs/VersionInfo
Source Control Management

- Do not leave home without it!
- Track down where your code broke.
- Make a Branch for testing ideas.
- Merge back when ready.
- Store your regression test “gold” files.
Out of Bounds Errors

- Array out of bounds errors are the bane of our profession.
- Fortran Compilers support array bounds checking.
- C/C++: valgrind, purify, splint(C), ElectricFence, DMALLOC, ...
- C++ has some support for range checking.
Non-normalized Floating point numbers

- Normalized Float Range: $10^{\pm38}$.
- Normalized Double Range: $10^{\pm308}$.
- What happens when you see $4.94 \times 10^{-324}$?
- An array/pointer out of bounds error.
Fortran Example: Non-Normalized Double

```fortran
program out_of_bounds
    implicit none

    integer    :: i
    real(8)    :: a(2)
    integer(8) :: k(2) = (/ 1, 2 /)

    equivalence (a(1), k(1))

    print *, "a: ", (a(i), i = 1,2)
    print *, "k: ", (k(i), i = 1,2)

end program out_of_bounds
```

$ ./out
a:  4.940656458412465E-324  9.881312916824931E-324
k:  1  2
C Example: Non-Normalized Double

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    typedef union {
        long i; double x;
    } Foo;
    Foo foo;
    foo.i = 1;
    printf("%22.17g\n",foo.x);
    return 0;
}

$ ./out
4.9406564584124654e-324
```
Out of Bounds Errors

- $k(1) = 1$
- $a(1) = 4.94065e-324$
- Non-Normalized $\Rightarrow$ Out of Bounds.
- Complete example in `defensive_progs/out_of_bounds`
My first exposure to Testing

- Perl Installation and Testing:
  
  ```
  lib/Math/BigInt/t/mbi.................ok
  lib/Math/BigInt/t/mif..................ok
  lib/Math/BigInt/t/bigfltpm............ok
  lib/Math/BigInt/t/bigintc.............ok
  ```

- Wouldn’t something like that be great for my codes?
Testing Philosophies

- Unit Testing: Test each routine independently.
- Integration Testing: Test sub-systems.
- System Testing: Test the whole program.
- Test Driven Design.
Unit Testing

• Much of the testing literature ⇒ Unit Testing.
• JUnit, CUnit, CppUnit, xUnit, ...
• What do most numerical codes do?
• Solve $Ax = b$ or $x_{i+1} = Ax_i + b$.
• Hard to unit test every entry in $A$. 
What do you know?

• In general you know the solution but not the intermediate steps.
• You know $x$ and $b$.
• Might know some analytical solutions.
• Can manufacture solutions.
• Might have experimental data.
• Might have really good eyeballs/intuition.
• $\Rightarrow$ System Testing.
Method of Manufactured Solutions

• Suppose you have $\nabla \cdot k \nabla T = q$
• Define $k(T) = ...$ and $T = ...$
• Mathematically compute $q = ...$
• Voilà: one test case.
CoA: Implement Algorithm in different language

- Your program is written in Fortran, C and/or C++ ⇒ Speed.
- For testing use Matlab or a scripting language such as Python, Perl, Lua, ...
- Use your scripting language code to verify answers.
- Scripting languages typically handle memory management.
- Scripting: Simple Implementation, High level operation (Matvec, Matrix Mult, ...)
Testing Criteria and Tests

- Must have some way to test the solution.
- We don’t expect exactly the same solution bit for bit.
- Different Machines, Opt Level, Compiler Version.
- Serial vs. Parallel: order of operation differences.
- We need some kind of Norm.
What is close enough?

- diff or cmp
- Relative or absolute $L_2$ norm

$$L_2^R = \sqrt{\frac{\sum (u_j - u_j^*)^2}{\sum (u_j^*)^2}}$$  \hspace{1cm} (1)

$$L_2^A = \sqrt{\sum (u_j - u_j^*)^2}$$  \hspace{1cm} (2)

- Max difference at one point.
- Atmospheric results (31 level, Value decreases with increasing height).
- Typically use $1 \times 10^{-6}$ for both norms in serial or parallel.
Program Development Style

• Refactor: Make the smallest change and test and repeat.
• Sometimes I’ve made too big a change.
• Use SCM to checkout old version and redo changes.
• Always Refactor, Restructure then Test before adding New Features.
What kind of Tests (I) ?

- Test every important feature of the code.
- Rapid turn around.
- Not all tests must be fast but long ones get run much less.
- Use night or weekends to run long tests.
What kind of Tests (II) ?

• Make most tests small.
• Small Grids, Run 2 to 5 iterations, Restart Solution
• Run on your personal computer.
• Run Mpi locally, 2,4,8 processes, even on a laptop.
• I run 32 tests (some parallel) < 30 secs on a laptop.
Automate Testing

• Need a norm testing program: close enough to “gold”.
• Some script to report: passed, failed, diff for each test.
• Software package: buildbot.
• `make small_test; make medium_test; make large_test`
• Some scripting tool.
Wanted Features in Test Manager

- Test Manager normally doesn’t build the executable(s).
- Automate everything else:
  - Run program or Submit to batch queue
  - Run Comparison tool
  - Report Passed, Failed or Diff
Power features

- Easy to run just one test.
- Easy to select a subset of test matching some criteria: Short, long, serial, parallel, some feature, number of processors.
- Easy to rerun the tests that fail/diff (i.e. wrong).
- Easy to “cd” to cases that are “wrong”.
- Easy to cleanup old tests.
- Support running both batch and interactive modes.
- Easy to “re” analyze results.
Magic through Regression Testing

- Makes refactoring possible.
- MGFLO/MGF story:
  - C/Fortran $\Rightarrow$ C $\Rightarrow$ C++ $\Rightarrow$ Real C++ $\Rightarrow$
    Dial-an-operator Version $\Rightarrow$ Solver Control Version
  - Tracked down an array that was $a(j,i)$ needed to be $a(i,j)$.
  - Would have been very difficult to find otherwise.
- Using Version control on tests has been a life saver.
  - Can use “binary search” to track down program changes.
• Introduction

• Defensive Programming

• Language Specific Defenses

• Design with Testing in Mind

• Conclusions
• implicit none
• Otherwise: a world of hurt.
• One typo can be impossible to find: 1, 1 and 0, 0
• NASA and do 100 j = 1.4
What will be printed?

program do_100_j
integer j
j = 0

    do 100 j = 1.4
        print *,j
    100 continue

end program do_100_j

• 1, 2, 3, 4?
• 1.4?
• 0?
• defensive_progs/do_100_j
Fortran 90 (II)

- Use different compilers: g95, gfortran, ifort, pgf90.
- Consider placing routines inside module.
  - This checks argument type and count.
  - Automatically generate dependencies (fdepend).
- C-preprocessor for conditional compilation.
Fortran 90 Memory Management

- Can only leak memory by use of pointers, not allocatable.
- Memory lifetime is still an issue.
- Limited tools for F90 memory issues: valgrind?
Fortran 90 assert

#if (defined(_GFORTRAN__) || defined(_G95__) || defined(__PGI__))
#   define MKSTR(x) "x"
#else
#   define MKSTR(x) #x
#endif
#ifndef NDEBUG
#   define ASSERT(x, msg) if (.not. (x)) call assert(_FILE_,_LINE_,MKSTR(x),msg)
#else
#   define ASSERT(x, msg)
#endif

subroutine assert(file, ln, testStr, msgIn)
   implicit none
   character(*) :: file, testStr, msgIn
   integer :: ln

   print *, "Assert: ",trim(testStr)," Failed at ",trim(file),":",ln
   print *, "Msg:", trim(msgIn)
   stop
end subroutine assert

subroutine build_edges()
   ...
   ASSERT(ibdry_edge == Num_Edges, "Miss count in boundary edges")
end
Flush output

- call flush(io_unit)
- Intel ifort, PGI pgf90 supports “flush io_unit”
  - A Fortran 2003 feature.
C/C++ Memory Management

- Memory Management Errors are the biggest problem.
- Leaking memory, Out of bounds errors.
- If your program messes up the heap, it will abort in an MPI routine.
- It is unlikely to be an MPI problem.
C/C++ Memory Management (II)

- Check status of malloc/calloc/realloc everytime.
- C++: register new_handler to detect out_of_memory problems.
Using compiler to track down problems

- Different compilers report different errors/warnings
- GCC/Intel: -Wall -Wextra
- PGI: -Minform
- You can turn off certain warnings:
  - GCC: -Wno-unused-functions -Wno-unused-parameters
  - Intel: -diag-disable:1419,869
• Introduction

• Defensive Programming

• Language Specific Defenses

• Design with Testing in Mind

• Conclusions
Design with Testing in Mind

• How can we test these new routine(s)?
• One number theory.
• EventStep Trick.
One Number Theory

• If you have a sea of numbers that are wrong:
• Just trace one number.
• Errors are likely to be:
  – The initial data is wrong.
  – Failed to scatter or bcast correctly.
  – Some wrong number at the boundary.
  – Error in original code.
EventStep Trick

• While refactoring, your code no-longer produces the “correct answers”

• What techniques can you use:
  – Random Print Statements
  – Debugger like ddt, totalview, idb, gdb, ...
  – It would be great to isolate the problem.
do i = 1,n

    !Some Compute Step
    call compute(a, sz, i)

    !Another Compute Step
    ! A: FACTOR(i): 2*i  B: FACTOR(i) : i+1
    call compute(b, sz, FACTOR(i))

end do
Adding EventStep

do i = 1,n

!Some Compute Step
call compute(a, sz, i)

! Print intermediate results
call dbg_prtVec("a", a, sz, eventStep)
call dbg_fini(FILE, LINE, eventStep)

!Another Compute Step
! A: FACTOR(i): 2*i  B: FACTOR(i) : i+1
call compute(b, sz, FACTOR(i))

! Print intermediate results
call dbg_prtVec("b", b, sz, eventStep)
call dbg_fini(FILE, LINE, eventStep)

end do

• Complete Example at defensive_progs/pairTest
Using EventStep Idea

- Place “good” version in directory “a”.
- Place “broken” version in directory “b”.
- Create “pairTest.sh” in directory above.
- Compile code with “-g -00”.

EventStep
#!/bin/bash
runProg()
{
  DIR=$1; CMD=$2; WD='pwd'
  cd $DIR; eval "$CMD"; cd $WD
}
MAXSTEP=10
for i in $(seq 1 $MAXSTEP); do
  export EVENTSTEP=$i
  runProg b "rm -f foo.*; b"
  runProg a "rm -f foo.*; a"
  cmp a/foo.vec b/foo.vec
  status=$?
  if [ $status -eq 0 ]; then
    echo " EventStep: $EVENTSTEP Passed"
  else
    echo " EventStep: $EVENTSTEP Diff"
    break;
  fi
done
Results

EventStep: 1 Passed
EventStep: 2 Passed
EventStep: 3 Passed
a/foo.vec b/foo.vec differ: byte 35, line 2
EventStep: 4 Diff
• Introduction

• Defensive Programming

• Language Specific Defenses

• Design with Testing in Mind

• Conclusions
Tools for Defensive Programmers

- Know your build system: Gnu Make, bjam.
- Use a SCM: svn, git.
- Design w/ Testing in Mind.
- Know Bash Scripting.
- Know Another Scripting language: Matlab, Python, Perl.
Conclusions

• Yes you can clean a pool with a toothbrush but a power washer is much faster.
• In Real Estate: Location, Location, Location.
• In Computational Science: Conservation of Answer.
• Which means: Test, Test, Test.
• Example code at: ~/train00/defensive_progs.tar.gz on Ranger
References

• Gnu Make Manual: Stallman, McGrath, Gnu.
• Managing Projects with GNU Make: Mechlenburg, O’Reilly, 2004.
• Programming in Lua, Ierusalimschy, Lua.org, 2006.
• Version Control with Git, Loeliger, O’Reilly, 2009.