



Debugging Techniques and Tools

V. Ruggiero (v.ruggiero@ Cineca.it)
Roma, 14 July 2017

SuperComputing Applications and Innovation Department

Outline

Introduction

Static analysis

Run-time analysis

Debugging

Conclusions

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.!

Maurice Wilkes discovers debugging, 1949.

Testing-Debugging

- ▶ **TESTING**: finds errors.
- ▶ **DEBUGGING**: localizes and repairs them.

TESTING DEBUGGING CYCLE:
we test, then debug, then repeat.

Testing-Debugging

- ▶ **TESTING**: finds errors.
- ▶ **DEBUGGING**: localizes and repairs them.

TESTING DEBUGGING CYCLE:

we test, then debug, then repeat.

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

Edsger Dijkstra



What is a bug?

- ▶ **Defect:** An incorrect program code



What is a bug?

- ▶ **Defect:** An incorrect program code \implies a bug in the code.

What is a bug?

- ▶ **Defect:** An incorrect program code \implies a bug in the code.
- ▶ **Infection:** An incorrect program state

What is a bug?

- ▶ **Defect:** An incorrect program code \implies a bug in the code.
- ▶ **Infection:** An incorrect program state \implies a bug in the state.

What is a bug?

- ▶ **Defect:** An incorrect program code \implies a bug in the code.
- ▶ **Infection:** An incorrect program state \implies a bug in the state.
- ▶ **Failure:** An observable incorrect program behaviour

What is a bug?

- ▶ **Defect**: An incorrect program code \implies a bug in the code.
- ▶ **Infection**: An incorrect program state \implies a bug in the state.
- ▶ **Failure**: An observable incorrect program behaviour \implies a bug in the behaviour.

Infection chain

Defect

- ▶ The programmer creates a **defect** in the program code (also known as bug or fault).

Infection chain

Defect \implies Infection

- ▶ The programmer creates a **defect** in the program code (also known as bug or fault).
- ▶ The **defect** causes an **infection** in the program state.



Infection chain

Defect \implies Infection \implies Failure

- ▶ The programmer creates a **defect** in the program code (also known as bug or fault).
- ▶ The **defect** causes an **infection** in the program state.
- ▶ The **infection** creates a **failure** - an externally observable error.

Tracking down defect

Failure

- ▶ A **Failure** is visible to the end user of a program. For example, the program prints an incorrect output.



Tracking down defect

Infection \leftarrow Failure

- ▶ A **Failure** is visible to the end user of a program. For example, the program prints an incorrect output.
- ▶ **Infection** is the underlying state of the program at runtime that leads to a **Failure**. For example, the program might display the incorrect output because the wrong value is stored in a variable.



Tracking down defect

Defect \leftarrow Infection \leftarrow Failure

- ▶ A **Failure** is visible to the end user of a program. For example, the program prints an incorrect output.
- ▶ **Infection** is the underlying state of the program at runtime that leads to a **Failure**. For example, the program might display the incorrect output because the wrong value is stored in a variable.
- ▶ A **Defect** is the actual incorrect fragment of code that the programmer wrote; this is what must be changed to fix the problem.



First of all..

THE BEST WAY TO DEBUG A PROGRAM IS
TO MAKE NO MISTAKES

First of all..

THE BEST WAY TO DEBUG A PROGRAM IS
TO MAKE NO MISTAKES

- ▶ Preventing bugs.
- ▶ Detecting/locating bugs.

First of all..

THE BEST WAY TO DEBUG A PROGRAM IS TO MAKE NO MISTAKES

- ▶ Preventing bugs.
- ▶ Detecting/locating bugs.

Ca. 80 percent of software development costs spent on identifying and correcting defects.

It is much more expensive (in terms of time and effort) to detect/locate existing bugs, than prevent them in the first place.

The Fundamental question

How can I prevent Bugs?

The Fundamental question

How can I prevent Bugs?

- ▶ Design.
- ▶ Good writing.
- ▶ Self-checking code.
- ▶ Test scaffolding.

Testing: a simple program

Input

Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Testing: a simple program

Input

Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Output

Tell whether the triangle is:

Scalene
Isosceles
Equilateral.

Testing: a simple program

Input

Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Output

Tell whether the triangle is:

Scalene
Isosceles
Equilateral.

Create a set of test cases for this program.

Testing: a simple program

Input

Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Output

Tell whether the triangle is:

Scalene
Isosceles
Equilateral.

Create a set of test cases for this program.
("The art of software testing" G.J. Myers)

Testing: a simple program

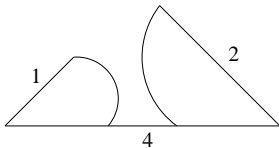
Q:1 Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third ?

Testing: a simple program

Q:1 Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third ?

(4,1,2) is an invalide triangle.

(a,b,c) with $a > b+c$



Define valide triangles $a < b + c$



Testing: a simple program

Q:2 Do you have a test case with some permutations of previous test?



Testing: a simple program

Q:2 Do you have a test case with some permutations of previous test?

(1,4,2) (4,1,2)

Fulfill above definition, but are still invalid.

Patch definition of valid triangles:

$$a < b + c$$

$$b < a + c$$

$$c < a + b$$

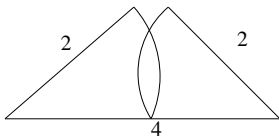
Testing: a simple program

Q:3 Do you have a test case with three integers greater than zero such that the sum of two numbers is equal to the third?

Testing: a simple program

Q:3 Do you have a test case with three integers greater than zero such that the sum of two numbers is equal to the third?

(4,2,2) is invalid triangle with equal sum.



Fulfill above definition, but is invalid:

$$a < b+c$$

$$b < a+c$$

$$c < a+b$$



Testing: a simple program

Do you have a test case:

4. with some permutations of previous test? (2,4,2) (2,2,4)
5. that represents a valid scalene triangle? (3,4,5)
6. that represents a valid equilateral triangle? (3,3,3)
7. that represents a valid isosceles triangle? (4,3,3)
8. with some permutations of previous test? (3,4,3) (3,3,4)
9. in which one side has a zero value? (0,4,3)
10. in which one side has a negative value? (-1,4,3)
11. in which all sides are zero? (0,0,0)
12. specifying at least one noninteger value? (2,2.5,4)
13. specifying the wrong number of values? (2,3) or (2,3,5,4)
14. For each test case did you specify the expected output from the program in addition to the input values?

About the example

- ▶ A set of test case that satisfies these conditions does not guarantee that all possible errors would be found.
- ▶ An adequate test of this program should expose at least these errors.
- ▶ Highly qualified professional programmers score, on the average, **7.8** out of a possible 14.

Outline

Introduction

Static analysis

Run-time analysis

Debugging

Conclusions

Definitions

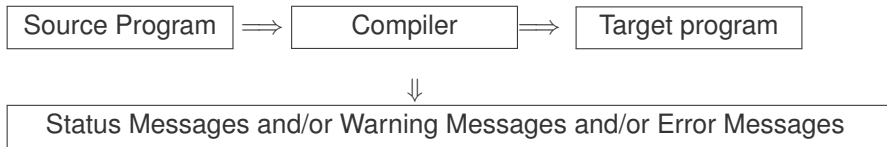
Static analysis refers to a method of examining software that allows developers to discover dangerous programming practices or potential errors in source code, without actually run the code.

Definitions

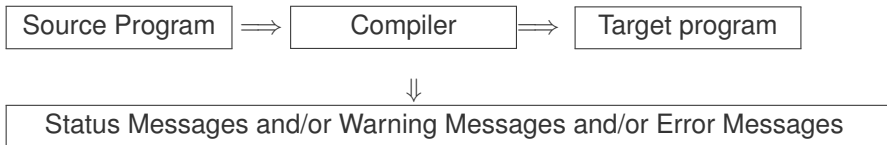
Static analysis refers to a method of examining software that allows developers to discover dangerous programming practices or potential errors in source code, without actually run the code.

- ▶ Using compiler options.
- ▶ Using static analyzer.

Compiler and errors



Compiler and errors



Compiler checks:

- ▶ Syntax.
- ▶ Semantic.

Compilers

- ▶ Not all compilers find the same defects.
- ▶ The more information a compilers has, the more defects it can find.
- ▶ Some compilers operate in "forgiving" mode but have "strict" or "pedantic" mode, if you request it.



Static analyzer for C

- ▶ `splint` [-option -option ...] filename [filename ...]
- ▶ Based on Lint
 - ▶ Unused declarations.
 - ▶ Type inconsistencies.
 - ▶ Variables used before being assigned.
 - ▶ Function return values that are ignored.
 - ▶ Execution paths with no return.
 - ▶ Apparent infinite loops.
 - ▶ Dereferencing pointers with possible null values
 - ▶ Problematic uses of macros.
 - ▶ Memory leaks
 - ▶ ...

Static analyzer for Fortran

- ▶ [ftnchek](#) Fortran 77 support. Free.
- ▶ [Forcheck](#) Full Fortran 2008 syntax support and verification of standard conformance.
- ▶ [Cleanscape FortranLint](#) OpenMP support.
- ▶ [plusFORT](#) is a multi-purpose suite of tools for analyzing and improving Fortran programs.
- ▶ ...



Static analyzer: errors and warnings

- ▶ 40% false positive reports of correct code.
- ▶ 40% multiple occurrence of same problem.
- ▶ 10% minor or cosmetic problems.
- ▶ 10% serious bugs, very hard to find by other methods.

From "**The Developer's Guide to Debugging**" T. Grotker, U. Holtmann, H. Keding, M. Wloka

Static analyzer: Lessons learned

- ▶ Do not ignore compiler warnings, even if they appear to be harmless.
- ▶ Use multiple compilers to check the code.
- ▶ Familiarize yourself with a static checker.
- ▶ Reduce static checker errors to (almost) zero.
- ▶ Rerun all test cases after a code cleanup.
- ▶ Doing regular sweeps of the source code will pay off in long term.

From "**The Developer's Guide to Debugging**" T. Grotker, U. Holtmann, H. Keding, M. Wloka

Outline

Introduction

Static analysis

Run-time analysis
Memory checker

Debugging

Conclusions

Runtime signals

- ▶ When a job terminates abnormally, it usually tries to send a signal (exit code) indicating what went wrong.
- ▶ The exit code from a job is a standard OS termination status.
- ▶ Typically, exit code 0 (zero) means successful completion.
- ▶ Your job itself calling `exit()` with a non-zero value to terminate itself and indicate an error.
- ▶ The specific meaning of the signal numbers is **platform-dependent**.



Runtime signals

You can find out why the job was killed using:

```
[ruggiero@matrix1 ~]$ kill -l
```

```
1) SIGHUP  2) SIGINT  3) SIGQUIT  4) SIGILL
5) SIGTRAP 6) SIGABRT 7) SIGBUS  8) SIGFPE
9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT
17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU
25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH
29) SIGIO 30) SIGPWR 31) SIGSYS 34) SIGRTMIN
....
```



Runtime signals

To find out what all the "kill -l" words mean:

```
[ruggiero@matrix1 ~]$ man 7 signal
```

```
.....  
Signal      Value      Action      Comment  
-----  
SIGHUP      1           Term        Hangup detected on controlling terminal  
or death of controlling process  
SIGINT      2           Term        Interrupt from keyboard  
SIGQUIT     3           Core        Quit from keyboard  
SIGILL      4           Core        Illegal Instruction  
SIGABRT     6           Core        Abort signal from abort(3)  
.....
```


Action description

Term	Default action is to terminate the process.
Ign	Default action is to ignore the signal.
Core	Default action is to terminate the process and dump the core.
Stop	Default action is to stop the process.
Cont	Default action is to continue the process if is currently stopped.



Common runtime signals

Signal name	OS signal name	Description
Floating point exception	SIGFPE	The program attempted an arithmetic operation with values that do not make sense
Segmentation fault	SIGSEGV	The program accessed memory incorrectly
Aborted	SIGABRT	Generated by the runtime library of the program or a library it uses, after having detecting a failure condition.

FPE example

```
1 main()
2 {
3     int a = 1.;
4     int b = 0.;
5     int c = a/b;
6 }
```



FPE example

```
1 main()
2 {
3     int a = 1.;
4     int b = 0.;
5     int c = a/b;
6 }
```

```
[ruggiero@matrix1 ~]$ gcc fpe_example.c
```

```
[ruggiero@matrix1 ~]$ ./a.out
```

Floating exception



SEGV example

```
1 main()
2 {
3   int array[5]; int i;
4     for(i = 0; i < 255; i++) {
5       array[i] = 10;}
6   return 0;
7 }
```

SEGV example

```
1 main()
2 {
3   int array[5]; int i;
4     for(i = 0; i < 255; i++) {
5       array[i] = 10;}
6   return 0;
7 }
```

```
[ruggiero@matrix1 ~]$ gcc segv_example.c
```

```
[ruggiero@matrix1 ~]$ ./a.out
```

Segmentation fault



ABORT example

```
1 #include <assert.h>
2 main()
3 {
4     int i=0;
5     assert(i!=0);
6 }
```

ABORT example

```
1 #include <assert.h>
2 main()
3 {
4     int i=0;
5     assert(i!=0);
6 }
```

```
[ruggiero@matrix1 ~]$ gcc abort_example.c
```

```
[ruggiero@matrix1 ~]$ ./a.out
```

```
a.out: abort_example.c:5: main: Assertion 'i!=0' failed.
Abort
```


Common runtime errors

- ▶ Allocation Deallocation errors (AD).
- ▶ Array conformance errors (AC).
- ▶ Array Index out of Bound (AIOB).
- ▶ Language specific errors (LS).
- ▶ Floating Point errors (FP).
- ▶ Input Output errors (IO).
- ▶ Memory leaks (ML).
- ▶ Pointer errors (PE).
- ▶ String errors (SE).
- ▶ Subprogram call errors (SCE).
- ▶ Uninitialized Variables (UV).

Useful link

- ▶ Iowa State University's High Performance Computing Group
- ▶ Run Time Error Detection Test Suites for Fortran, C, and C++
- ▶ <http://rted.public.iastate.edu>



Grading Methodology: score

- ▶ **0.0:** is given when the error was not detected.
- ▶ **1.0:** is given for error messages with the correct error name.
- ▶ **2.0:** is given for error messages with the correct error name and line number where the error occurred but not the file name where the error occurred.
- ▶ **3.0:** is given for error messages with the correct error name, line number and the name of the file where the error occurred.
- ▶ **4.0:** is given for error messages which contain the information for a score of 3.0 but less information than needed for a score of 5.0 .
- ▶ **5.0:** is given in all cases when the error message contains all the information needed for the quick fixing of the error.



Grading Methodology : an example

```

!*****
!  copyright (c) 2005 Iowa State University, Glenn Luecke, James Coyle,
!  James Hoekstra, Marina Kraeva, Olga Taborskaia, Andre Wehe, Ying Xu,
!  and Ziyu Zhang, All rights reserved.
!  Licensed under the Educational Community License version 1.0.
!  See the full agreement at http://rted.public.iastate.edu/ .
!*****
!*****
!
!   Name of the test:  F_H_1_1_b.f90
!
!   Summary:          allocation/deallocation error
!
!   Test description: deallocation twice
!                   for allocatable array in a subroutine
!                   contains in a main program
!
!   Support files:    Not needed
!
!   Env. requirements: Not needed
!
!   Keywords:         deallocation error
!                   subroutine contains in a main program
!

```



Grading Methodology: an example

```

!
!   Last modified:      1/17/2005
!
!   Programmer:        Ying Xu, Iowa State Univ.
!*****
program tests
  implicit none
  integer :: n=10, m=20
  double precision :: var

  call sub(n,m,var)
  print *,var
contains
  subroutine sub(n,m,var)
    integer, intent(in) :: n,m
    double precision, intent(inout) :: var
    double precision, allocatable :: arr(:, :) ! DECLARE

```



Grading Methodology: an example

```
integer :: i,j
allocate (arr(1:n,1:m))
do i=1,n
  do j=1,m
    arr(i,j) = dble(i*j)
  enddo
end do
var = arr(n,m)
deallocate(arr)
deallocate(arr) ! deallocate second time here. ERROR
return
end subroutine sub
end program tests
```

Grading Methodology: an example

Real message (grade 1.0)

```
Fortran runtime error: Internal: Attempt to DEALLOCATE  
unallocated memory.
```

Ideal message (grade 5.0)

```
ERROR: unallocated array  
At line 52 column 17 of subprogram 'sub'  
in file 'F_H_1_1_b.f90', the argument  
'arr' in the DEALLOCATE statement is an  
unallocated array. The variable is declared  
in line 41 in subprogram 'sub' in file 'F_H_1_1_b.f90'.
```

Fortran Results

Compiler	AC	A D	AIOB	LS	FP	IO
gcc-4.3.2	1	0.981481	3.40025	2.88235	0	2.33333
gcc-4.3.2	1	1.38889	3.40025	2.88235	0	2.33333
gcc-4.4.3	1	1.38889	3.40025	2.88235	0	2.33333
gcc-4.6.3	1	1.27778	0.969504	0.94117	0	2.33333
gcc-4.7.0	1	1.38889	0.969504	0.94117	0.714286	2.33333
gcc-4.8.2	1	1.38889	0.969504	0.94117	0.142857	2.33333
gcc-4.9.2	1	1.42593	0.969504	0.94117	0.142857	2.33333
g95.4.0.3	0.421053	1.22222	3.60864	3.82353	0.571428	2.66667
intel-10.1.021	0.421053	1.42593	3.45362	2.82353	0.571428	2.11111
intel-11.0.074	0.421053	1.68519	3.446	2.82353	0.571428	2.11111
intel-11.1	1	1.90741	3.47649	2.88235	1.42857	2.33333
intel-12.0.2	0.421053	1.62963	3.44727	2.82353	0.571428	2.11111
intel-14.0.1	0.421053	1.62963	3.44854	2.82353	0.571428	2.11111
intel-15.0.2	0.421053	1.62963	3.44854	2.82353	2.85714	2.11111
open64.4.2.3	3	0.888889	2.63405	3	0	1
pgi-7.2-5	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-8.0-1	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-10.3	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-11.8	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-12.8	1	1	3.8831	3.82353	1	2.61111
pgi-13.10	1	1	3.8831	3.82353	1	2.72222
pgi-14.1.0	0.421053	0.5	3.8526	3.82353	0	2.61111
pgi-14.10.0	0.421053	0.5	3.8526	3.82353	0	2.61111
pgi-16.3	0.421053	0.5	3.8526	3.82353	0	2.61111
sun.12.1	3	2.77778	3.00381	3	2	2.16667
sun.12.1+bcheck	3	2.77778	3.03431	3	0.285714	2.16667

Fortran Results

Compiler	ML	PE	SE	SCE	UV
gcc-4.3.2	0	3.49609	3.25	0	0.0159236
gcc-4.3.2	0	3.49609	3.25	0	0.0286624
gcc-4.4.3	0	3.49609	3.25	0	0.0286624
gcc-4.6.3	0	1	3.25	0.166667	0.22293
gcc-4.7.0	0	1	3.25	0.166667	0.130573
gcc-4.8.2	0	1	3.25	0.166667	0.0955414
gcc-4.9.2	0	1	3.25	0.166667	0.0955414
g95.4.0.3	1	3	3.43333	0	0.0159236
intel-10.1.021	0	3.5625	0	0.166667	0.286624
intel-11.0.074	0	3.55469	0	0.166667	0.299363
intel-11.1	1	3.55469	1	1	1.07643
intel-12.0.2	0	3.55469	0	0.166667	0.292994
intel-14.0.1	0	3.55469	0	0.166667	0.292994
intel-15.0.2	0	3.55469	0	0.166667	0.286624
open64.4.2.3	0	3.3625	0	0.0833333	0.286624
pgi-7.2-5	0	4	0	0	0.022293
pgi-8.0-1	0	4	0	0	0.022293
pgi-10.3	0	4	0	0	0.0254777
pgi-11.8	0	4	0	0	0.0127389
pgi-12.8	1	4	1	1	1
pgi-13.10	1	4	1	1	1
pgi-14.1.0	0	4	0	0	0.143312
pgi-14.10.0	0	4	0	0	0.0127389
pgi-16.3	0	4	0	0	0.136943
sun.12.1	0	3.03125	3	0	0.022293
sun.12.1+bcheck	1.25	3.03125	3	1	0.640127

C Results

Compiler	AD	AloB	LS	FP	IO
gcc-4.3.2	0.44	0.00925926	0.0416667	0.2	0.0666667
gcc-4.4.3	0.44	0.00925926	0.0416667	0.2	0.0666667
gcc-4.6.3	0.44	0.00925926	0.0416667	0.2	0.2
gcc-4.7.0	0.44	0.00925926	0.0416667	0.2	0.2
gcc-4.8.2	0.4	0.00925926	0.0416667	0	0.2
gcc-4.9.2	0.4	0.00925926	0.0416667	0	0.2
intel-10.1.021	0.52	0.00925926	0.0416667	0	0.2
intel-11.0.074	0.52	0.00925926	0.0416667	0	0.2
intel-11.1	0.68	1	1	1	1
intel-12.0.2	0.44	0.00925926	0.0416667	0	0.2
intel-14.0.1	0.4	0.00925926	0.0416667	0	0.2
intel-15.0.2	0.4	0.00925926	0.0416667	0	0.2
open64-4.2.3	0.52	0.00925926	0.0416667	0	0.2
pgi-7.2-5	0.44	0.00925926	0.0416667	0	0.2
pgi-8.0-1	0.44	0.00925926	0.0416667	0	0.2
pgi-10.3	0.44	0.00925926	0.0416667	0	0.2
pgi-11.8	0.44	0.00925926	0.0416667	0	0.2
pgi-12.8	0.68	2.40741	2.16667	0	1
pgi-13.10	0.68	2.40741	2.16667	0	1
pgi-14.1-0	0.48	1.93519	1.625	0	0.2
pgi-14.10-0	0.44	1.93519	1.625	0	0.3
pgi-16.3	0.44	0.00925926	0.0416667	0	0.2
sun-12.1	0.44	0.00925926	0	0	0.2
sun-12.1+bcheck	0.16	0	0	0	0

C Results



Compiler	ML	PE	SE	SCE	UV
gcc-4.3.2	0.0166667	0.0166667	0.05	0	0
gcc-4.4.3	0.0166667	0.0166667	0.05	0	0
gcc-4.6.3	0.0166667	0.0166667	0.05	0	0
gcc-4.7.0	0.0166667	0.0166667	0.05	0	0
gcc-4.8.2	0.0166667	0.0166667	0.075	0	0
gcc-4.9.2	0.0166667	0.0166667	0.075	0	0
intel-10.1.021	0.0666667	0.0166667	0.05	0	0
intel-11.0.074	0.0666667	0.0166667	0.05	0	0
intel-11.1	1	1	1	1	1
intel-12.0.2	0.0666667	0.0166667	0.05	0	0
intel-14.0.1	0.133333	0.0166667	0.05	0	0
intel-15.0.2	0.0166667	0.0166667	0.075	0	0
open64-4.2.3	0.0666667	0.0166667	0.05	0	0
pgi-7.2.5	0.0666667	0.0166667	0.05	0	0
pgi-8.0-1	0.0666667	0.0166667	0.05	0	0
pgi-10.3	0.0666667	0.0166667	0.05	0	0
pgi-11.8	0.0666667	0.0166667	0.05	0	0
pgi-12.8	1	1	1	1	1
pgi-13.10	1	1	1	1	1
pgi-14.1-0	0.133333	0.0166667	0.05	0	0
pgi-14.10-0	0.0666667	0.0166667	0.05	0	0
pgi-16.3	0.0666667	0.0166667	0.075	0	0
sun-12.1	0.0666667	0.0166667	0.05	0	0
sun-12.1+bcheck	1.13333	0.025	0	0	0

C++ Results

Compiler	AD	AloB	FP	IO	ML	PE	SE	UV
gcc-4.3.2	0.44	0.00925926	0	0	0.0666667	0.0166667	0.05	0
gcc-4.4.3	0.44	0.00925926	0	0	0.0666667	0.0166667	0.05	0
gcc-4.6.3	0.44	0.00925926	0	0.2	0.0666667	0.0166667	0.05	0
gcc-4.7.0	0.44	0.00925926	0	0.2	0.0666667	0.0166667	0.05	0
gcc-4.8.2	0.4	0.00925926	0	0.2	0.0666667	0.0166667	0.075	0
gcc-4.9.2	0.4	0.00925926	0	0.2	0.0666667	0.0166667	0.075	0
intel-10.1.021	0.330275	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
intel-11.0.074	0.330275	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
intel-11.1	0.926606	1	1	1	1	1	1	1
intel-12.0.2	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
intel-14.1.0	0.4	0.00925926	0	0.2	0.133333	0.0166667	0.05	0
intel-15.0.2	0.4	0.00925926	0	0.2	0.0666667	0.0166667	0.075	0
open64-4.2.3	0.330275	0.00903614	0	0	0.047619	0.0254777	0.05	0
pgi-7.2.5	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-8.0.1	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-10.3	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-11.8	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-12.8	0.926606	2.23795	1	1	1	1	1	1
pgi-13.1	0.926606	2.23494	1	1	1	1	1	1
pgi-14.1-0	0.321101	1.69277	0	0.0714286	0.0714285	0.0254777	0.05	0
pgi-14.10-0	0.321101	1.69277	0	0.107143	0.047619	0.0254777	0.05	0
pgi-16.3	0.44	0.00925926	0	0.2	0.0666667	0.0166667	0.075	0
sun-12.1	0.311927	0.00903614	0	0	0.047619	0.0254777	0.05	0
sun-12.1+bcheck	0.247706	0.0150602	0	0	1.16667	0.0191083	0	0



Grading Methodology: Used options

Fortran

gcc	-frange-check -O0 -fbounds-check -g -ffpe-trap=invalid,zero,overflow -fdiagnostics-show-location=every-line
g95	-O0 -fbounds-check -g -ftrace=full
intel	-O0 -C -g -traceback -ftrapuv -check
open64	-C -g -O0
pgi	-C -g -Mchkptr -O0
sun	-g -C -O0 -xs -ftrap=%all -fnonstd -xcheck=%all

C

gcc	-O0 -g -fbounds-check -ftrapv
intel	-O0 -C -g -traceback
open64	-g -C -O0
pgi	-g -C -Mchkptr -O0
sun	-g -C -O0 -xs -ftrap=%all -fnonstd -xcheck=%all

C++

gcc	-O0 -g -fbounds-check -ftrapv
intel	-O0 -C -g -traceback
open64	-g -C -O0
pgi	-g -C -Mchkptr -O0
sun	-g -C -O0 -xs -ftrap=%all -fnonstd -xcheck=%all

Outline

Introduction

Static analysis

Run-time analysis
Memory checker

Debugging

Conclusions



Fixing memory problems

- ▶ **Memory leaks** are data structures that are allocated at runtime, but not deallocated once they are no longer needed in the program.
- ▶ **Incorrect use of the memory management** is associated with incorrect calls to the memory management: freeing a block of memory more than once, accessing memory after freeing...
- ▶ **Buffer overruns** are bugs where memory outside of the allocated boundaries is overwritten, or corrupted.
- ▶ **Uninitialized memory bugs**: reading uninitialized memory.

Valgrind

- ▶ Open Source Software, available on Linux for x86 and PowerPc processors.
- ▶ Interprets the object code, not needed to modify object files or executable, non require special compiler flags, recompiling, or relinking the program.
- ▶ Command is simply added at the shell command line.
- ▶ No program source is required (black-box analysis).

www.valgrind.org

Valgrind:tools

- ▶ Memcheck: a memory checker.
- ▶ Callgrind: a runtime profiler.
- ▶ Cachegrind: a cache profiler.
- ▶ Helgrind: find race conditions.
- ▶ Massif: a memory profiler.

Why should I use Valgrind?

- ▶ Valgrind will tell you about tough to find bugs.
- ▶ Valgrind is very thorough.
- ▶ You may be tempted to think that Valgrind is too picky, since your program may seem to work even when valgrind complains. It is users' experience that fixing ALL Valgrind complaints will save you time in the long run.

But...

Why should I use Valgrind?

- ▶ Valgrind will tell you about tough to find bugs.
- ▶ Valgrind is very thorough.
- ▶ You may be tempted to think that Valgrind is too picky, since your program may seem to work even when valgrind complains. It is users' experience that fixing ALL Valgrind complaints will save you time in the long run.

But...

Valgrind is kind-of like a virtual x86 interpreter.

So your program will run 10 to 30 times slower than normal.

Valgrind won't check static arrays.



Use of uninitialized memory:test1.c

- ▶ Local Variables that have not been initialized.
- ▶ The contents of malloc's blocks, before writing there.



Use of uninitialized memory:test1.c

- ▶ Local Variables that have not been initialized.
- ▶ The contents of malloc's blocks, before writing there.

```
1 #include <stdlib.h>
2 int main()
3 {
4     int p,t,b[10];
5     if (p==5)
6         t=p+1;
7         b[p]=100;
8     return 0;
9 }
```



Use of uninitialized memory:test1.c

- ▶ Local Variables that have not been initialized.
- ▶ The contents of malloc's blocks, before writing there.

```
1 #include <stdlib.h>
2 int main()
3 {
4     int p,t,b[10];
5     if (p==5)      ERROR
6         t=p+1;
7         b[p]=100;  ERROR
8     return 0;
9 }
```

Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t1
```

Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t1
```

```
==7879== Memcheck, a memory error detector.  
      ....  
==7879== Conditional jump or move depends on uninitialised value(s)  
==7879==    at 0x8048399: main (test1.c:5)  
==7879==  
==7879== Use of uninitialised value of size 4  
==7879==    at 0x80483A7: main (test1.c:7)  
==7879==  
==7879== Invalid write of size 4  
==7879==    at 0x80483A7: main (test1.c:7)  
==7879==    Address 0xCEF8FE44 is not stack'd, malloc'd or (recently) free'd  
==7879==  
==7879== Process terminating with default action of signal 11 (SIGSEGV)  
==7879==    Access not within mapped region at address 0xCEF8FE44  
==7879==    at 0x80483A7: main (test1.c:7)  
==7879==  
==7879== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 3 from 1)  
==7879== malloc/free: in use at exit: 0 bytes in 0 blocks.  
==7879== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.  
==7879== For counts of detected errors, rerun with: -v  
==7879== All heap blocks were freed -- no leaks are possible.  
Segmentation fault
```




Illegal read/write test2.c

```
1  #include <stdlib.h>
2  int main()
3  {
4      int *p, i, a;
5      p=malloc(10*sizeof(int));
6      p[11]=1;
7      a=p[11];
8      free(p);
9      return 0;
10 }
```



Illegal read/write test2.c

```
1 #include <stdlib.h>
2 int main()
3 {
4     int *p, i, a;
5     p=malloc(10*sizeof(int));
6     p[11]=1; ERROR
7     a=p[11]; ERROR
8     free(p);
9     return 0;
10 }
```



Illegal read/write: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t2
```

```
.....
==8081== Invalid write of size 4
==8081==    at 0x804840A: main (test2.c:6)
==8081==   Address 0x417B054 is 4 bytes after a block of size 40 alloc'd
==8081==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8081==   by 0x8048400: main (test2.c:5)
==8081==
==8081== Invalid read of size 4
==8081==    at 0x8048416: main (test2.c:7)
==8081==   Address 0x417B054 is 4 bytes after a block of size 40 alloc'd
==8081==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8081==   by 0x8048400: main (test2.c:5)
==8081==
==8081== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 1)
==8081== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8081== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==8081== For counts of detected errors, rerun with: -v
==8081== All heap blocks were freed -- no leaks are possible.
```



Invalid free:test3.c

```
1 #include <stdlib.h>
2 int main()
3 {
4     int *p,i;
5     p=malloc(10*sizeof(int));
6     for(i=0;i<10;i++)
7         p[i]=i;
8     free(p);
9     free(p);
10    return 0;
11 }
```

Invalid free:test3.c

```
1 #include <stdlib.h>
2 int main()
3 {
4     int *p,i;
5     p=malloc(10*sizeof(int));
6     for(i=0;i<10;i++)
7         p[i]=i;
8     free(p);
9     free(p);    ERROR
10    return 0;
11 }
```



Invalid free: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t3
```

```
.....  
==8208== Invalid free() / delete / delete[]  
==8208==    at 0x40231CF: free (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==8208==    by 0x804843C: main (test3.c:9)  
==8208==    Address 0x417B028 is 0 bytes inside a block of size 40 free'd  
==8208==    at 0x40231CF: free (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==8208==    by 0x8048431: main (test3.c:8)  
==8208==  
==8208== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)  
==8208== malloc/free: in use at exit: 0 bytes in 0 blocks.  
==8208== malloc/free: 1 allocs, 2 frees, 40 bytes allocated.  
==8208== For counts of detected errors, rerun with: -v  
==8208== All heap blocks were freed -- no leaks are possible.
```

Mismatched use of functions: test4.cpp

- ▶ If allocated with **malloc**, **calloc**, **realloc**, **valloc** or **memalign**, you must deallocate with **free**.
- ▶ If allocated with **new[]**, you must deallocate with **delete[]**.
- ▶ If allocated with **new**, you must deallocate with **delete**.



Mismatched use of functions:test4.cpp

- ▶ If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- ▶ If allocated with **new[]**, you must deallocate with **delete[]**.
- ▶ If allocated with **new**, you must deallocate with **delete**.

```
1 #include <stdlib.h>
2 int main()
3 {
4     int *p,i;
5     p=(int*)malloc(10*sizeof(int));
6     for(i=0;i<10;i++)
7         p[i]=i;
8     delete(p);
9     return 0;
10 }
```




Mismatched use of functions:test4.cpp

- ▶ If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- ▶ If allocated with **new[]**, you must deallocate with **delete[]**.
- ▶ If allocated with **new**, you must deallocate with **delete**.

```
1 #include <stdlib.h>
2 int main()
3 {
4     int *p,i;
5     p=(int*)malloc(10*sizeof(int));
6     for(i=0;i<10;i++)
7         p[i]=i;
8     delete(p);    ERROR
9     return 0;
10 }
```



Mismatched use of functions: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t4
```

```
.....  
==8330== Mismatched free() / delete / delete []  
==8330== at 0x4022EE6: operator delete(void*) (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==8330==by 0x80484F1: main (test4.c:8)  
==8330==Address 0x4292028 is 0 bytes inside a block of size 40 alloc'd  
==8330==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==8330==by 0x80484C0: main (test4.c:5)  
==8330==  
==8330==ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)  
==8330==malloc/free: in use at exit: 0 bytes in 0 blocks.  
==8330==malloc/free: 1 allocs, 1 frees, 40 bytes allocated.  
==8330==For counts of detected errors, rerun with: -v  
==8330==All heap blocks were freed -- no leaks are possible.
```



Invalid system call parameter: test5.c

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 int main()
4 {
5     int *p;
6     p=malloc(10);
7     read(0,p,100);
8     free(p);
9     return 0;
10 }
```



Invalid system call parameter:test5.c

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 int main()
4 {
5     int *p;
6     p=malloc(10);
7     read(0,p,100); ERROR
8     free(p);
9     return 0;
10 }
```



Invalid system call parameter: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t5
```

```
...  
==18007== Syscall param read(buf) points to unaddressable byte(s)  
==18007==   at 0x4EEC240: __read_nocancel (in /lib64/libc-2.5.so)  
==18007==   by 0x40056F: main (test5.c:7)  
==18007== Address 0x517d04a is 0 bytes after a block of size 10 alloc'd  
==18007==   at 0x4C21168: malloc (vg_replace_malloc.c:236)  
==18007==   by 0x400555: main (test5.c:6)  
...
```



Memory leak detection:test6.c

```
1 #include <stdlib.h>
2     int main()
3     {
4         int *p,i;
5         p=malloc(5*sizeof(int));
6         for(i=0; i<5;i++)
7             p[i]=i;
8
9             return 0;
10    }
```



Memory leak detection:test6.c

```
1 #include <stdlib.h>
2     int main()
3     {
4         int *p,i;
5         p=malloc(5*sizeof(int));
6         for(i=0; i<5;i++)
7             p[i]=i;
8         free(p);
9         return 0;
10    }
```



Memory leak detection: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t6
```

```
.....  
==8237== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 3 from 1)  
==8237== malloc/free: in use at exit: 20 bytes in 1 blocks.  
==8237== malloc/free: 1 allocs, 0 frees, 20 bytes allocated.  
==8237== For counts of detected errors, rerun with: -v  
==8237== searching for pointers to 1 not-freed blocks.  
==8237== checked 65,900 bytes.  
==8237==  
==8237== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1  
==8237==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==8237==    by 0x80483D0: main (test6.c:5)  
==8237==  
==8237== LEAK SUMMARY:  
==8237==    definitely lost: 20 bytes in 1 blocks.  
==8237==    possibly lost: 0 bytes in 0 blocks.  
==8237==    still reachable: 0 bytes in 0 blocks.  
==8237==    suppressed: 0 bytes in 0 blocks.
```


What won't Valgrind find?

```
1      int main()  
2      {  
3          char x[10];  
4          x[11]='a';  
5      }
```

What won't Valgrind find?

```
1      int main()  
2      {  
3          char x[10];  
4          x[11]='a';  
5      }
```

- ▶ Valgrind doesn't perform bound checking on static arrays (allocated on stack).
- ▶ Solution for testing purposes is simply to change static arrays into dynamically allocated memory taken from the heap, where you will get bounds-checking, though this could be a message of unfreed memory.

sum.c: source

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (int argc, char* argv[]) {
4     const int size=10;
5     int n, sum=0;
6     int* A = (int*)malloc( sizeof(int)*size);
7
8     for(n=size; n>0; n--)
9         A[n] = n;
10    for(n=0; n<size; n++)
11        sum+=A[n];
12    printf("sum=%d\n", sum);
13    return 0;
14 }
```

sum.c: compilation and run

```
ruggiero@shiva:~> gcc -O0 -g -fbounds-check -ftrapv sum.c
```

```
ruggiero@shiva:~> ./a.out
```

```
sum=45
```

Valgrind:example

```
ruggiero@shiva:~> valgrind --leak-check=full --tool=memcheck ./a.out
```

```
==21579== Memcheck, a memory error detector.  
...  
==21791==Invalid write of size 4  
==21791==at 0x804842A: main (sum.c:9)  
==21791==Address 0x417B050 is 0 bytes after a block of size 40 alloc'd  
==21791==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==21791==by 0x8048410: main (sum.c:6)  
==21791==Use of uninitialised value of size 4  
==21791== at 0x408685B: _itoa_word (in /lib/libc-2.5.so)  
==21791==by 0x408A581: fprintf (in /lib/libc-2.5.so)  
==21791==by 0x4090572: printf (in /lib/libc-2.5.so)  
==21791==by 0x804846B: main (sum.c:12)  
==21791==  
==21791==Conditional jump or move depends on uninitialised value(s)  
==21791==at 0x4086863: _itoa_word (in /lib/libc-2.5.so)  
==21791==by 0x408A581: fprintf (in /lib/libc-2.5.so)  
==21791==by 0x4090572: printf (in /lib/libc-2.5.so)  
==21791==by 0x804846B: main (sum.c:12)  
==21791==40 bytes in 1 blocks are definitely lost in loss record 1 of 1  
==21791==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==21791==by 0x8048410: main (sum.c:6)  
==21791==
```

outbc.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3     int main (void)
4     {
5         int i;
6         int *a = (int*) malloc( 9*sizeof(int));
7
8         for ( i=0; i<=9; ++i){
9             a[i] = i;
10            printf ("%d\n ", a[i]);
11        }
12
13        free(a);
14        return 0;
15    }
```



outbc.c: compilation and run

```
ruggiero@shiva:~> icc -C -g outbc.c
```

```
ruggiero@shiva:~> ./a.out
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```



outbc.c: compilation and run

```
ruggiero@shiva:~> pgcc -C -g outbc.c
```

```
ruggiero@shiva:~> ./a.out
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```


Electric Fence

- ▶ Electric Fence (efence) stops your program on the exact instruction that overruns (or underruns) a malloc() memory buffer.
- ▶ GDB will then display the source-code line that causes the bug.
- ▶ It works by using the virtual-memory hardware to create a red-zone at the border of each buffer - touch that, and your program stops.
- ▶ Catch all of those formerly impossible-to-catch overrun bugs that have been bothering you for years.



Electric Fence

```
ruggiero@shiva:~> icc -g outbc.c libefence.a -o outbc -lpthread
```

```
ruggiero@shiva:~> ./outbc
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
Segmentation fault
```



Outline

Introduction

Static analysis

Run-time analysis

Debugging

gdb

Totalview

Conclusions



My program fails!

- ▶ Erroneous program results.
- ▶ Execution deadlock.
- ▶ Run-time crashes.

The ideal debugging process

- ▶ Find origins.
 - ▶ Identify test case(s) that reliably show existence of fault (when possible). Duplicate the bug.
- ▶ Isolate the origins of infection.
 - ▶ Correlate incorrect behaviour with program logic/code error.
- ▶ Correct.
 - ▶ Fixing the error, not just a symptom of it.
- ▶ Verify.
 - ▶ Where there is one bug, there is likely to be another.
 - ▶ The probability of the fix being correct is not 100 percent.
 - ▶ The probability of the fix being correct drops as the size of the program increases.
 - ▶ Beware of the possibility that an error correction creates a new error.

Bugs that can't be duplicated

- ▶ Dangling pointers.
- ▶ Initializations errors.
- ▶ Poorly synchronized threads.
- ▶ Broken hardware.

Isolate the origins of infection

- ▶ Divide and conqueror.
- ▶ Change one thing at time.
- ▶ Determine what you changed since the last time it worked.
- ▶ Write down what you did, in what order, and what happened as a result.
- ▶ Correlate the events.

Why is debugging so difficult?

- ▶ The symptom and the cause may be **geographically** remote.
- ▶ The symptom may **disappear (temporarily)** when another error is corrected.
- ▶ The symptom may actually be caused by **nonerrors** (e.g., round-off inaccuracies).
- ▶ It may be difficult to accurately reproduce input conditions (e.g, a **real time** application in which input ordering is indeterminate).
- ▶ The symptom may be due to causes that are **distributed** across a number of tasks running on different processors.

Why use a Debugger?

- ▶ No need for precognition of what the error might be.
- ▶ Flexible.
 - ▶ Allows for "live" error checking (no need to re-write and re-compile when you realize a certain type of error may be occurring).
- ▶ Dynamic.
 - ▶ **Execution Control** Stop execution on specified conditions: **breakpoints**
 - ▶ **Interpretation** **Step-wise** execution code
 - ▶ **State Inspection** **Observe** value of variables and stack
 - ▶ **State Change** **Change** the state of the stopped program.



Why people don't use debuggers?

- ▶ With simple errors, may not want to bother with starting up the debugger environment.
 - ▶ Obvious error.
 - ▶ Simple to check using prints/asserts.
- ▶ Hard-to-use debugger environment.
- ▶ Error occurs in optimized code.
- ▶ Changes execution of program (error doesn't occur while running debugger).

Why don't use print?

- ▶ Cluttered code.
- ▶ Cluttered output.
- ▶ Slowdown.
- ▶ Time consuming.
- ▶ And can be misleading.
 - ▶ Moves things around in memory, changes execution timing, etc.
 - ▶ Common for bugs to hide when print statements are added, and reappear when they're removed.



Outline

Introduction

Static analysis

Run-time analysis

Debugging

gdb

Totalview

Conclusions

What is gdb?

- ▶ The GNU Project debugger, is an open-source debugger.
- ▶ Protected by GNU General Public License (GPL).
- ▶ Runs on many Unix-like systems.
- ▶ Was first written by Richard Stallmann in 1986 as part of his GNU System.
- ▶ Is an Workstation Application Code extremely powerful all-purpose debugger.
- ▶ Its text-based user interface can be used to debug programs written in C, C++, Pascal, Fortran, and several other languages, including the assembly language for every micro-processor that GNU supports.
- ▶ www.gnu.org/software/gdb

Two levels of control

- ▶ Basic:
 - ▶ Run the code and wait for it crash
 - ▶ Identify line where it crashes
 - ▶ With luck the problem is obvious
- ▶ Advanced:
 - ▶ Set breakpoints
 - ▶ Analyze data at breakpoints
 - ▶ Watch specific variables



prime-numbers finding program

- ▶ Print a list of all primes which are less than or equal to the user-supplied upper bound *UpperBound*.
- ▶ See if J divides $K \leq UpperBound$, for all values J which are
 - ▶ themselves prime (no need to try J if it is nonprime)
 - ▶ less than or equal to $\text{sqrt}(K)$ (if K has a divisor larger than this square root, it must also have a smaller one, so no need to check for larger ones).
- ▶ *Prime[l]* will be 1 if l is prime, 0 otherwise.



Main.c

```
1 #include <stdio.h>
2 #define MaxPrimes 50
3 int Prime[MaxPrimes], UpperBound;
4 int main()
5 {
6     int N;
7     printf("enter upper bound\n");
8     scanf("%d", UpperBound);
9     Prime[2] = 1;
10    for (N = 3; N <= UpperBound; N += 2)
11        CheckPrime(N);
12    if (Prime[N]) printf("%d is a prime\n",N);
13    return 0;
14 }
```




CheckPrime.c

```
1 #define MaxPrimes 50
2 extern int Prime[MaxPrimes];
3 void CheckPrime(int K)
4 {
5     int J;
6     for (J = 2; J*J <=K; J++)
7         if (Prime[J] == 1)
8             if (K % J == 0) {
9                 Prime[K] = 0;
10                return ;
11            }
12     Prime[K] = 1;
13 }
```



Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o check_prime
```

Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o check_prime
```

```
<ruggiero@matrix2 ~> ./check_prime
```

Compilation and run

```
<ruiggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o check_prime
```

```
<ruiggiero@matrix2 ~> ./check_prime
```

enter upper bound



Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o check_prime
```

```
<ruggiero@matrix2 ~> ./check_prime
```

enter upper bound

20



Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o check_prime
```

```
<ruggiero@matrix2 ~> ./check_prime
```

```
enter upper bound
```

```
20
```

```
Segmentation fault
```

Compilation options for gdb

- ▶ You will need to compile your program with the appropriate flag to enable generation of symbolic debug information, the `-g` option is used for this.
- ▶ Don't compile your program with optimization flags while you are debugging it.

Compiler optimizations can "rewrite" your program and produce machine code that doesn't necessarily match your source code.

Compiler optimizations may lead to:

- ▶ Misleading debugger behaviour.
 - ▶ Some variables you declared may not exist at all
 - ▶ some statements may execute in different places because they were moved out of loops
- ▶ Obscure the bug.

Lower optimization level

- ▶ When your program has crashed, disable or lower optimization to see if the bug disappears.
(optimization levels are not comparable between compilers, not even -O0).
- ▶ If the bug persists \implies you can be quite sure there's something wrong in your application.
- ▶ If the bug disappears, without a serious performance penalty \implies send the bug to your computing center and continue your simulations.



Lower optimization level

- ▶ When your program has crashed, disable or lower optimization to see if the bug disappears.
(optimization levels are not comparable between compilers, not even -O0).
- ▶ If the bug persists \implies you can be quite sure there's something wrong in your application.
- ▶ If the bug disappears, without a serious performance penalty \implies send the bug to your computing center and continue your simulations.
- ▶ **But your program may still contain a bug that simply doesn't show up at lower optimization \implies have some checks to verify the correctness of your code.**



Lower optimization level

- ▶ When your program has crashed, disable or lower optimization to see if the bug disappears.
(optimization levels are not comparable between compilers, not even -O0).
- ▶ If the bug persists \implies you can be quite sure there's something wrong in your application.
- ▶ If the bug disappears, without a serious performance penalty \implies send the bug to your computing center and continue your simulations.
- ▶ **But your program may still contain a bug that simply doesn't show up at lower optimization \implies have some checks to verify the correctness of your code.**

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o check_prime
```



Running under debugger

- ▶ Option 1: Run gdb with the program passed in as parameter



Running under debugger

- ▶ Option 1: Run gdb with the program passed in as parameter

```
<ruggiero@matrix2 ~>gdb check_prime
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb) r
```

Running under debugger

- ▶ Option 2: Run gdb first and then load and execute the program from the debugger command line:



Running under debugger

- ▶ Option 2: Run gdb first and then load and execute the program from the debugger command line:

```
<ruggiero@matrix2 ~>gdb
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb) exec-file check_prime
```



Running under debugger

- ▶ Option 2: Run gdb first and then load and execute the program from the debugger command line:

```
<ruggiero@matrix2 ~>gdb
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb) exec-file check_prime
```

```
(gdb) r
```



Running under debugger

- ▶ Option 3: Attach the debugger to a running program

Running under debugger

- ▶ Option 3: Attach the debugger to a running program

```
<ruggiero@matrix2 ~> ps -u |grep check_prime
```

Running under debugger

- ▶ Option 3: Attach the debugger to a running program

```
<ruggiero@matrix2 ~> ps -u |grep check_prime
```

```
ruggiero 25934 0.0 0.0 4168 356 pts/80 T 13:55 0:00 ./check_prime
```



Running under debugger

- ▶ Option 3: Attach the debugger to a running program

```
<ruggiero@matrix2 ~> ps -u |grep check_prime
```

```
ruggiero 25934 0.0 0.0 4168 356 pts/80 T 13:55 0:00 ./check_prime
```

```
(gdb) attach 25934
```

Using gdb

- ▶ When you run the gdb command
 - ▶ The **-d** option is useful when source and executable reside in different directories
 - ▶ Use **-q** to skip the licensing message.
- ▶ In the gdb environment
 - ▶ Type **help** at any time to see a list of the debugger options and commands.
 - ▶ Type **she** execute the rest of the line as a shell command.

Starting gdb

```
<ruggiero@matrix2 ~>gdb check_prime
```



Starting gdb

```
<ruggiero@matrix2 ~>gdb check_prime
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb)
```

gdb: Basic commands

command	shorthand	argument	description
run/kill	r/k	NA	start/end program being debugged
continue	c	NA	continue running program from last breakpoint
step	s	NA	take a single step in the program from last position
next	n	NA	Step program, proceeding through subroutine calls
where/backtrace	NA/bt	NA negative value	Print backtrace of all stack frames, or innermost COUNT frames print outermost -COUNT frames
print	p	variableName	show value of a variable
list	l	SrcFile:lineNumber	show the specified source code line
break	b	SrcFile:lineNumber functionName	set breakpoint at specified line set breakpoint at function name
watch	NA	variableName	stops when the variable changes value
display	NA	variableName	print value of variable name each time the program stops.
quit	q	NA	exit gdb



prime-number finding program

```
(gdb) r
```




prime-number finding program

```
(gdb) r
```

```
Starting program: check_prime  
enter upper bound
```



prime-number finding program

```
(gdb) r
```

```
Starting program: check_prime  
enter upper bound
```

```
20
```



prime-number finding program

```
(gdb) r
```

```
Starting program: check_prime  
enter upper bound
```

```
20
```

```
Program received signal SIGSEGV, Segmentation fault.  
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```



prime-number finding program

```
(gdb) r
```

```
Starting program: check_prime  
enter upper bound
```

```
20
```

```
Program received signal SIGSEGV, Segmentation fault.  
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6  
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6  
#2 0x000000000040054f in main () at Main.c:8
```



prime-number finding program

```
(gdb) list Main.c:8
```

prime-number finding program



```
(gdb) list Main.c:8
```

```
#0 0x00000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6  
#1 0x00000003faa25ee7c in scanf () from /lib64/libc.so.6  
#2 0x000000000040054f in main () at Main.c:8
```

```
3  int Prime[MaxPrimes],UpperBound;  
5  main()  
6  {  int N;  
7      printf("enter upper bound\n");  
8      scanf("%d",&UpperBound);  
9      Prime[2] = 1;  
10     for (N = 3; N <= UpperBound; N += 2)  
11         CheckPrime(N);  
12         if (Prime[N]) printf("%d is a prime\n",N);
```



Main.c :new version

```
1 #include <stdio.h>
2 #define MaxPrimes 50
3 int Prime[MaxPrimes],UpperBound;
4 int main()
5 {
6     int N;
7     printf("enter upper bound\n");
8     scanf("%d", UpperBound);
9     Prime[2] = 1;
10    for (N = 3; N <= UpperBound; N += 2)
11        CheckPrime(N);
12        if (Prime[N]) printf("%d is a prime\n",N);
13    return 0;
14 }
```



Main.c :new version

```
1 #include <stdio.h>
2 #define MaxPrimes 50
3 int Prime[MaxPrimes],UpperBound;
4 int main()
5 {
6     int N;
7     printf("enter upper bound\n");
8     scanf("%d", &UpperBound);
9     Prime[2] = 1;
10    for (N = 3; N <= UpperBound; N += 2)
11        CheckPrime(N);
12        if (Prime[N]) printf("%d is a prime\n",N);
13    return 0;
14 }
```


Main.c :new version

```
1 #include <stdio.h>
2 #define MaxPrimes 50
3 int Prime[MaxPrimes],UpperBound;
4 int main()
5 {
6     int N;
7     printf("enter upper bound\n");
8     scanf("%d", &UpperBound);
9     Prime[2] = 1;
10    for (N = 3; N <= UpperBound; N += 2)
11        CheckPrime(N);
12        if (Prime[N]) printf("%d is a prime\n",N);
13    return 0;
14 }
```

In other shell COMPILATION



prime-number finding program

(gdb)

Kill the program being debugged? (y or n)



prime-number finding program

```
(gdb)
```

```
Kill the program being debugged? (y or n) y
```

```
(gdb) run
```

```
Starting program: check_prime  
enter upper bound
```



prime-number finding program

```
(gdb)
```

```
Kill the program being debugged? (y or n) y
```

```
(gdb) run
```

```
Starting program: check_prime  
enter upper bound
```

```
20
```



prime-number finding program

```
(gdb)
```

```
Kill the program being debugged? (y or n) y
```

```
(gdb) run
```

```
Starting program: check_prime  
enter upper bound
```

```
20
```

```
Program exited normally.
```

gdb: Breakpoint control

- ▶ Stop the execution of the program
- ▶ Allow you to examine the execution state in detail
- ▶ Can be assigned to a line or function
- ▶ Can be set conditionally

command	argument	description
info	breakpoints/b/br	Prints to screen all breakpoints
breakpoint	srcFile:lineNumber a<b	Conditional insertion of breakpoint
enable/disable	breakpointNumber	Enable/disable a breakpoint
delete	breakpointNumber	Delete a breakpoint
clear	srcFile:lineNumber functionName	Clear breakpoints at given line or function



gdb: Examining data

C	Fortran	Result
(gdb) p x	(gdb) p x	Print scalar data x value
(gdb) p V	(gdb) p V	Print all V vector components
(gdb) p V[i]	(gdb) p V(i)	Print element i of V vector
(gdb) p V[i]@n	(gdb) p V(i)@n	Print n consecutive elements starting with V_i
(gdb) p M	(gdb) p M	Print all matrix M elements
(gdb) p M[i]	Not Available	Print row i of matrix M
(gdb) p M[i]@n	Not Available	Print n consecutive rows starting with i row
(gdb) p M[i][j]	(gdb) p M(i,j)	Print matrix element Mij
(gdb) p M[i][j]@n	(gdb) p M(i,j)@n	Print n consecutive elements starting with Mij

gdb commands

```
(gdb) break Main.c:1
```


gdb commands

```
(gdb) break Main.c:1
```

```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```



prime-number finding program

```
(gdb) next
```

prime-number finding program

```
(gdb) next
```

```
main () at Main.c:7  
7         printf("enter upper bound\n");
```



prime-number finding program

```
(gdb) next
```



prime-number finding program

```
(gdb) next
```

```
8         scanf ("%d", &UpperBound);
```

```
(gdb) next
```



prime-number finding program

```
(gdb) next
```

```
8         scanf ("%d", &UpperBound);
```

```
(gdb) next
```

```
20
```



prime-number finding program

```
(gdb) next
```

```
8         scanf("%d", &UpperBound);
```

```
(gdb) next
```

```
20
```

```
9         Prime[2] = 1;
```

```
(gdb) next
```



prime-number finding program

```
(gdb) next
```

```
8         scanf("%d", &UpperBound);
```

```
(gdb) next
```

```
20
```

```
9         Prime[2] = 1;
```

```
(gdb) next
```

```
10        for (N = 3; N <= UpperBound; N += 2)
```

```
(gdb) next
```


prime-number finding program

```
(gdb) next
```

```
8         scanf("%d", &UpperBound);
```

```
(gdb) next
```

```
20
```

```
9         Prime[2] = 1;
```

```
(gdb) next
```

```
10        for (N = 3; N <= UpperBound; N += 2)
```

```
(gdb) next
```

```
        CheckPrime(N);
```



prime-number finding program

```
(gdb) display N
```

prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```



prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6  
6         for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6  
6           for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

```
12           Prime[K] = 1;
```

```
(gdb) next
```

```
}
```



prime-number finding program

```
(gdb) n
```



prime-number finding program

```
(gdb) n
```

```
10         for (N = 3; N <= UpperBound; N += 2)
11: N = 3
12     }
```

```
(gdb) n
```



prime-number finding program

```
(gdb) n
```

```
10         for (N = 3; N <= UpperBound; N += 2)
11: N = 3
12: }
13: }
```

```
(gdb) n
```

```
11         CheckPrime(N);
12: N = 5
13: }
```

```
(gdb) n
```




prime-number finding program

```
(gdb) n
```

```
10         for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11         CheckPrime(N);
1: N = 5
```

```
(gdb) n
```

```
10         for (N = 3; N <= UpperBound; N += 2)
1: N = 5
```

```
(gdb) n
```

prime-number finding program

```
(gdb) n
```

```
10         for (N = 3; N <= UpperBound; N += 2)
11: N = 3
12     }
```

```
(gdb) n
```

```
11         CheckPrime(N);
12: N = 5
```

```
(gdb) n
```

```
10         for (N = 3; N <= UpperBound; N += 2)
11: N = 5
```

```
(gdb) n
```

```
11         CheckPrime(N);
12: N = 7
```



prime-number finding program

```
(gdb) 1 Main.c:10
```

prime-number finding program

```
(gdb) 1 Main.c:10
```

```
5     main()
6     {   int N;
7         printf("enter upper bound\n");
8         scanf("%d", &UpperBound);
9         Prime[2] = 1;
10        for (N = 3; N <= UpperBound; N += 2)
11            CheckPrime(N);
12        if (Prime[N]) printf("%d is a prime\n",N);
13        return 0;
14    }
```



Main.c :new version

```
1 #include <stdio.h>
2 #define MaxPrimes 50
3 int Prime[MaxPrimes],
4   UpperBound;
5   main()
6   { int N;
7     printf("enter upper bound\n");
8     scanf("%d",&UpperBound);
9     Prime[2] = 1;
10    for (N = 3; N <= UpperBound; N += 2)
11        CheckPrime(N);
12        if (Prime[N]) printf("%d is a prime\n",N);
13
14    return 0;
15 }
```

Main.c :new version

```
1 #include <stdio.h>
2 #define MaxPrimes 50
3 int Prime[MaxPrimes],
4   UpperBound;
5   main()
6   { int N;
7     printf("enter upper bound\n");
8     scanf("%d",&UpperBound);
9     Prime[2] = 1;
10    for (N = 3; N <= UpperBound; N += 2) {
11        CheckPrime(N);
12        if (Prime[N]) printf("%d is a prime\n",N);
13    }
14    return 0;
15 }
```



prime-number finding program

In other shell COMPILATION



prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n)

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) **y**

```
(gdb) d
```



prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) **y**

```
(gdb) d
```

Delete all breakpoints? (y or n)



prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) **y**

```
(gdb) d
```

Delete all breakpoints? (y or n) **y**



prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) **y**

```
(gdb) d
```

Delete all breakpoints? (y or n) **y**

```
(gdb) r
```

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) **y**

```
(gdb) d
```

Delete all breakpoints? (y or n) **y**

```
(gdb) r
```

```
Starting program: check_prime  
enter upper bound
```

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) **y**

```
(gdb) d
```

Delete all breakpoints? (y or n) **y**

```
(gdb) r
```

```
Starting program: check_prime  
enter upper bound
```

20



prime-number finding program

```
3 is a prime  
5 is a prime  
7 is a prime  
11 is a prime  
13 is a prime  
17 is a prime  
19 is a prime
```

```
Program exited normally.
```



prime-number finding program

```
(gdb) list Main.c:6
```


prime-number finding program

```
(gdb) list Main.c:6
```

```
1      #include <stdio.h>
2      #define MaxPrimes 50
3      int Prime[MaxPrimes],
4      UpperBound;
5      main()
6      { int N;
7        printf("enter upper bound\n");
8        scanf("%d",&UpperBound);
9        Prime[2] = 1;
10       for (N = 3; N <= UpperBound; N += 2){
```



prime-number finding program

```
(gdb) break Main.c:8
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb) run
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb) run
```

```
Starting program: check_prime  
enter upper bound  
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8  
8      scanf("%d",&UpperBound);
```

```
(gdb) next
```



prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb) run
```

```
Starting program: check_prime  
enter upper bound  
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8  
8      scanf("%d",&UpperBound);
```

```
(gdb) next
```

```
20
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb) run
```

```
Starting program: check_prime  
enter upper bound  
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8  
8         scanf("%d",&UpperBound);
```

```
(gdb) next
```

```
20
```

```
9         Prime[2] = 1;
```



prime-number finding program

```
(gdb) set UpperBound=40  
(gdb) continue
```



prime-number finding program

```
(gdb) set UpperBound=40  
(gdb) continue
```

Continuing.

```
3 is a prime  
5 is a prime  
7 is a prime  
11 is a prime  
13 is a prime  
17 is a prime  
19 is a prime  
23 is a prime  
29 is a prime  
31 is a prime  
37 is a prime
```

Program exited normally.

Debugging post mortem

- ▶ When a program exits abnormally the operating system can write out `core` file, which contains the memory state of the program at the time it crashed.
- ▶ Combined with information from the symbol table produced by `-g` the core file can be used to find the line where program stopped, and the values of its variables at that point.
- ▶ Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- ▶ In the GNU Bash shell the command `ulimit -c` control the maximum size of the core files. If the size limit is set to zero, no core files are produced.



Debugging post mortem

- ▶ When a program exits abnormally the operating system can write out **core** file, which contains the memory state of the program at the time it crashed.
- ▶ Combined with information from the symbol table produced by **-g** the core file can be used to find the line where program stopped, and the values of its variables at that point.
- ▶ Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- ▶ In the GNU Bash shell the command **ulimit -c** control the maximum size of the core files. If the size limit is set to zero, no core files are produced.

```
ulimit -c unlimited  
gdb exe_file core
```

Graphical Debuggers

- ▶ `gdb -tui` or `gdbtui`
- ▶ `ddd` (data display debugger) is a graphical front-end for command-line debuggers.
- ▶ `ddt` (Distributed Debugging Tool) is a comprehensive graphical debugger for scalar, multi-threaded and large-scale parallel applications that are written in C, C++ and Fortran.
- ▶ Etc.

Why don't optimize?

```
1  int main(void)
2  {
3      float q;
4      q=3.;
5      return 0;
6  }
```



Why don't optimize?

```
1 int main(void)
2 {
3     float q;
4     q=3.;
5     return 0;
6 }
```

```
<ruggiero@shiva ~/CODICI>gcc opt.c -g -O0 -o opt
<ruggiero@matrix2 ~>gdb opt
(gdb) b main
```

Breakpoint 1 at 0x8048395: file opt.c, line 4.

Why don't optimize?

```

1  int main(void)
2  {
3      float q;
4      q=3.;
5      return 0;
6  }
```

```

<ruggiero@shiva ~/CODICI>gcc  opt.c -g -O0 -o opt
<ruggiero@matrix2 ~>gdb opt
(gdb) b main
```

Breakpoint 1 at 0x8048395: file opt.c, line 4.

```

<ruggiero@shiva ~/CODICI>gcc  opt.c -g -O3 -o opt
<ruggiero@matrix2 ~>gdb opt
(gdb) b main
```

Breakpoint 1 at 0x8048395: file opt.c, line 6.



Outline

Introduction

Static analysis

Run-time analysis

Debugging

gdb

Totalview

Conclusions



Totalview (www.totalviewtech.com)

- ▶ Used for debugging and analyzing both serial and parallel programs.
- ▶ Supported languages include the usual HPC application languages:
 - ▶ C,C++,Fortran
 - ▶ Mixed C/C++ and Fortran
 - ▶ Assembler
- ▶ Supported many commercial and Open Source Compilers.
- ▶ Designed to handle most types of HPC parallel coding (multi-process and/or multi-threaded applications).
- ▶ Supported on most HPC platforms.
- ▶ Provides both a GUI and command line interface.
- ▶ Can be used to debug programs, running processes, and core files.
- ▶ Provides graphical visualization of array data.
- ▶ Includes a comprehensive built-in help system.
- ▶ And more...



Compilation options for Totalview

- ▶ You will need to compile your program with the appropriate flag to enable generation of symbolic debug information. For most compilers, the **-g** option is used for this.
- ▶ It is recommended to compile your program **without optimization flags** while you are debugging it.
- ▶ TotalView will allow you to debug executables which were not compiled with the -g option. However, only the assembler code can be viewed.
- ▶ Some compilers may require additional compilation flags. See the *TotalView User's Guide* for details.

```
ifort [option] -O0 -g file_source.f -o filename
```



Starting Totalview

Command	Action
totalview	Starts the debugger. You can then load a program or corefile, or else attach to a running process.
totalview <i>filename</i>	Starts the debugger and loads the program specified by <i>filename</i> .
totalview <i>filename corefile</i>	Starts the debugger and loads the program specified by <i>filename</i> and its core file specified by <i>corefile</i> .
totalview <i>filename -a args</i>	Starts the debugger and passes all subsequent arguments (specified by <i>args</i>) to the program specified by <i>filename</i> . The <i>-a</i> option must appear after all other TotalView options on the command line.

Totalview:panel

1. Stack Trace

- ▶ Call sequence

2. Stack Frame

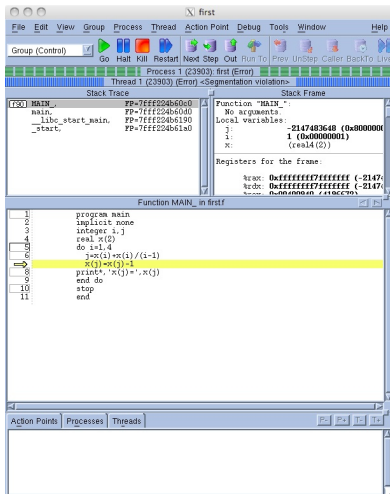
- ▶ Local variables and their values

3. Source Window

- ▶ Indicates presently executed statement
- ▶ Last statement executed if program crashed

4. Info tabs

- ▶ Informations about processes and action points.



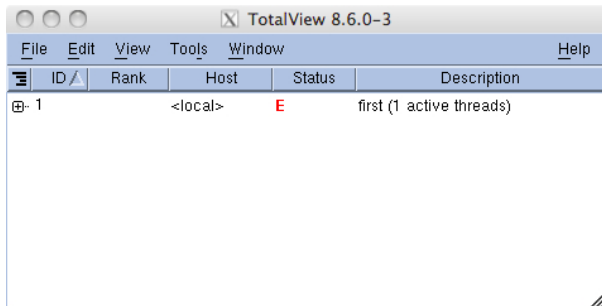
Totalview: Action points

- ▶ **Breakpoint** stops the execution of the process and threads that reach it.
 - ▶ Unconditional
 - ▶ Conditional: stop only if the condition is satisfied.
 - ▶ Evaluation: stop and execute a code fragment when reached.
- ▶ **Process barrier point** synchronizes a set of processes or threads.
- ▶ **Watchpoint** monitors a location in memory and stop execution when its value changes.

Totalview: Setting Action points

- ▶ **Breakpoint**
 - ▶ Right click on a source line → Set breakpoint
 - ▶ Click on the line number
- ▶ **Watchpoint**
 - ▶ Right click on a variable → Create watchpoint
- ▶ **Barrier point**
 - ▶ Right click on a source line → Set barrier
- ▶ **Edit action point property**
 - ▶ Right click on a action point in the Action Points tab → Properties.

Totalview:Status

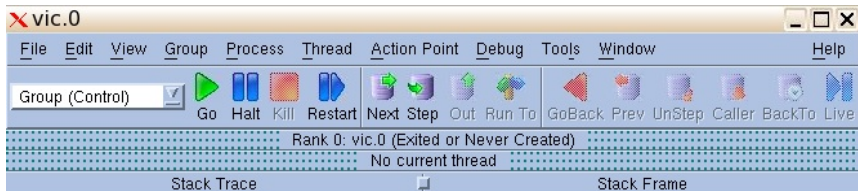


The screenshot shows a window titled "TotalView 8.6.0-3" with a menu bar (File, Edit, View, Tools, Window, Help) and a table of thread information. The table has columns for ID, Rank, Host, Status, and Description. One thread is listed with ID 1, Rank <local>, Status E, and Description "first (1 active threads)".

ID	Rank	Host	Status	Description
1	<local>		E	first (1 active threads)

Status Code	Description
T	Thread is stopped
B	Stopped at a breakpoint
E	Stopped because of a error
W	At a watchpoint
H	In a Hold state
M	Mixed - some threads in a process are running and some not
R	Running

Totalview: Execution control commands



Command	Description
Go	Start/resume execution
Halt	Stop execution
Kill	Terminate the job
Restart	Restarts a running program, or one that has stopped without exiting
Next	Run to next source line or instruction. If the next line/instruction calls a function the entire function will be executed and control will return to the next source line or instruction.
Step	Run to next source line or instruction. If the next line/instruction calls a function, execution will stop within function.
Out	Execute to the completion of a function. Returns to the instruction after one which called the function.
Run to	Allows you to arbitrarily click on any source line and then run to that point.

Totalview: Mouse buttons

Mouse Button	Purpose	Description	Examples
Left	Select	Clicking on object causes it to be selected and/ or to perform its action	Clicking a line number sets a breakpoint. Clicking on a process/thread name in the root window will cause its source code to appear in the Process Window's source frame.
Middle	Dive	Shows additional information about the object - usually by popping open a new window.	Clicking on an array object in the source frame will cause a new window to pop open, showing the array's values.
Right	Menu	Pressing and holding this button a window/frame will cause its associated menu to pop open.	Holding this but ton while the mouse pointer is in the Root Window will cause the Root Window menu to appear. A menu selection can then be made by dragging the mouse pointer while continuing to press the middle button down.

I've checked everything!

what else could it be?

- ▶ Full file system.
- ▶ Disk quota exceeded.
- ▶ File protection.
- ▶ Maximum number of processes exceeded.
- ▶ Are all object file are up do date? Use Makefiles to build your projects
- ▶ What did I change since last version of my code?
Use a version control system: CVS,RCS,...
- ▶ Does any environment variable affect the behaviour of my program?

Outline

Introduction

Static analysis

Run-time analysis

Debugging

Conclusions



Last resorts

- ▶ Ask for help.
- ▶ Explain your code to somebody.
- ▶ Go for a walk, to the movies, leave it to tomorrow.

Errare humanum est ...

- ▶ Where was error made?
- ▶ Who made error?
- ▶ What was done incorrectly?
- ▶ How could the error have been prevented?
- ▶ Why wasn't the error detected earlier?
- ▶ How could the error have been detected earlier?

Bibliography

- ▶ **Why program fail. A guide to systematic debugging.** A. Zeller *Morgan Kaufmann Publishers* 2005.
- ▶ **Expert C Programming deep C secrets** P. Van der Linden *Prentice Hall PTR* 1994.
- ▶ **How debuggers works Algorithms, data,structure, and Architecture** J. B. Rosemberg *John Wiley & Sons* 1996.
- ▶ **Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code** R. B. Blunden *Apress* 2003.
- ▶ **Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems** D.J. Agans *American Management Association* 2002.
- ▶ **The Art of Debugging** N. Matloff, P. J. Salzman *No starch press* 2008.

Bibliography

- ▶ **Debugging With GDB: The Gnu Source-Level Debugger** R.M. Stallmann, R.H. Pesch, S. Shebs *Free Software Foundation* 2002.
- ▶ **The Practice of Programming** B.W. Kernighan, R. Pike *Addison-Wesley* 1999.
- ▶ **Code Complete** S. McConnell *Microsoft Press* 2004.
- ▶ **Software Testing Technique** B. Beizer *The Coriolis Group* 1990.
- ▶ **The Elements of Programming Style** B. W. Kernighan P.J. Plauger *Computing Mcgraw-Hill* 1978.
- ▶ **The Art of Software testing** K.J. Myers *Kindle Edition* 1979.
- ▶ **The Developer's Guide to Debugging** H. Grotker, U. Holtmann, H. Keding, M.Wloka *Kindle Edition* 2008.
- ▶ **The Science of DEBUGGING** M. Telles Y. Hsieh *The Coriolis Group* 2001.