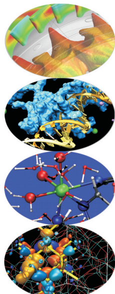
SCAI SuperComputing Applications and Innovation — CINECA

SAPIENZA Università di Roma

Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)



Roma, Marzo 2017

Outline

- Bugs and Prevention
- Testing
- Static analysis
- Run-time analysis
- Debugging
- Parallel debugging

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 $\Longrightarrow$ a bug in the code.

CINECA SCAI SuperComputing Applications and Innovation

What is a bug?

SAPIENZA UNIVERSITÀ DI ROMA

- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state

CINECA

What is a bug?

- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state $\Longrightarrow$ a bug in the state.

What is a bug?

- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state $\Longrightarrow$ a bug in the state.
- Failure: An osservable incorrect program behaviour

What is a bug?

- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state $\Longrightarrow$ a bug in the state.
- Failure: An osservable incorrect program behaviour $\Longrightarrow$ 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 $\Longrightarrow$ Infection

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

Infection chain

Defect $\Longrightarrow$ Infection $\Longrightarrow$ Failure

- The programmer creates a defect in the program code (also known as bug or fault).
- The defect causes an in 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 $\Longleftarrow$ 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 avariable.

Tracking down defect

Defect $\Longleftarrow$ Infection $\Longleftarrow$ 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 avariable.
- 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 exisisting bugs, than prevent them in the first place.

www.ifsq.org

www.ifsq.org

- We have an agreed common goal: to raise the standard of software (and software development) around the world by promoting Code Inspection as a prerequisite to Software Testing in the production and delivery cycle.
- Our strong hope is that eventually the idea of inspecting code before testing will be as self-evident as testing software before using it.

The Fundamental question

How can I prevent Bugs?

The Fundamental question

How can I prevent Bugs?

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

Preventing bugs via design

Programming is a design activity.
It’s a creative act, not mechanical code generation.

Preventing bugs via design

Programming is a design activity.
It's a creative act, not mechanical code generation.

- Good modularization.
- Strong encapsulation/information hiding.
- Clear,simple pre- and post-processing.
- Requirements of operations should testable.

What is a module?

- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.

What is a module?

- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.
  39% of all errors are caused by internal
  interface errors / errors in communication between routines.

What is a module?

- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.
  39% of all errors are caused by internal
  interface errors / errors in communication between routines.
- Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand.
- The art of software engineering is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.

What is a module?

- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.
  39% of all errors are caused by internal
  interface errors / errors in communication between routines.
- Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand.
- The art of software engineering is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.
  A study showed that when routines averaged 100 to 150 lines each, code was more stable and required less changes.

Good modularization...

- reduces complexity
- avoids duplicate code
- facilitates reusable code
- limits effects of the changes
- facilitates test
- results in easier implementation

Strong encapsulation/information hiding

- The principle of information hiding suggests that modules be characterized by design decisions that each hides from all others.
- The information contained within a module is inaccesible to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- Because most data and procedure are hidden from other parts of software , inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Cohesion & Coupling

- Cohesion: a measure of how closely the instructions in a module are related to each other.
- Coupling: a measure of how closely a modules’ execution depends upon other modules.

Cohesion & Coupling

- Cohesion: a measure of how closely the instructions in a module are related to each other.
- Coupling: a measure of how closely a modules' execution depends upon other modules.

- Avoid global variables.
- As a general rule, you should always aim to create modules that have strong cohesion and weak coupling.
- The routines with the highest coupling-to-cohesion ratios had 7 times more errors than those with the lowest ratios.

Cohesion & Coupling

- Cohesion: a measure of how closely the instructions in a module are related to each other.
- Coupling: a measure of how closely a modules' execution depends upon other modules.

- Avoid global variables.
- As a general rule, you should always aim to create modules that have strong cohesion and weak coupling.
- The routines with the highest coupling-to-cohesion ratios had 7 times more errors than those with the lowest ratios.

Spaghetti code is characterized by very strong coupling.

- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.

Preventing bugs via good writing

- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.

      Studies have shown that few people can understand nesting of conditional statements to more than 3 levels.

- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.
      Studies have shown that few people can understand nesting of conditional statements to more than 3 levels.
- Comments,comments,comments.
- Small size of functions,routines.

- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.
      Studies have shown that few people can understand nesting of conditional statements to more than 3 levels.
- Comments,comments,comments.
- Small size of functions,routines.
  A study at IBM found that the most error-prone routines were those larger than 500 lines of code.

Preventing bugs via good writing 2

- Use the smallest acceptable scope for:
  - Variable names.
  - Static/local functions.
- Use named intermediate values.
- Avoid "Magic numbers".
- Generalize your code.
- Document your program.
- Write standard language.

Clarity of writing


```
.....
do j=1,n
  do i=1,n
    v(i,j)=(i/j)*(j/i)
  end do
end do
.....
```

Clarity of writing


```
.....
do j=1,n
  do i=1,n
    v(i,j)=(i/j)*(j/i)
  end do
end do
.....
```

```
.....
do j=1,n
  do i=1,n
    v(i,j)=0.0
    v(i,i)=1.0
  end do
end do
.....
```

Use meaningful names

The name of a variable, function, etc. should answer all the big questions:

Use meaningful names

The name of a variable, function, etc. should answer all the big questions:

- why it exists

Use meaningful names

The name of a variable, function, etc. should answer all the big questions:

- why it exists
- what it does

Use meaningful names

The name of a variable, function, etc. should answer all the big questions:

- why it exists
- what it does
- how it is used.

Use meaningful names

The name of a variable, function, etc. should answer all the big questions:

- why it exists
- what it does
- how it is used.

If a name requires a comment, then the name does not reveal its intent.

Use meaningful names

Purpose of variables	Good names Good descriptors	Bad names Bad descriptors
Running total of checks written to date	runningTotal,checkTotal nChecks	written,ct,checks, CHKTTL,x,x1,x2
Velocity of a bullet train	velocity,trainVelocity velocityInMph	velt,v,tv, x,x1,x2,train
Current date	currentDate,todaysDate	cd,current,c,x,x1,x2,dat
Lines per page	LinesPerPage	lpp,lines,l,x,x1,x2

Optimum name length

Too long: numberOfPeopleOnTheUsOlympicteam
numberOfSeatsInTheStadium
maximumNumberOfPointsModernOlympics

Too short: n,np,ntn
n,ns,nsisd
m,mp,max,points

Just right: numberTeammembers,teamMemberCount
numSeatInStadium,seatCount
teamPointsMax,pointsRecord

Optimum name length

Too long: numberOfPeopleOnTheUsOlympicteam
numberOfSeatsInTheStadium
maximumNumberOfPointsModernOlympics

Too short: n,np,ntn
n,ns,nsisd
m,mp,max,points

Just right: numberTeammembers,teamMemberCount
numSeatInStadium,seatCount
teamPointsMax,pointsRecord

The effort required to debug a program is minimized when variables had names averaging 10 to 16 characters long.

Comments in the code

- Make sure comments and code are agree.

Comments in the code

- Make sure comments and code are agree.
- Don’t comment bad code - rewrite it.

Comments in the code

- Make sure comments and code are agree.
- Don't comment bad code - rewrite it.
- Use variable names that mean something.

Comments in the code

- Make sure comments and code are agree.
- Don’t comment bad code - rewrite it.
- Use variable names that mean something.
- Don’t over-comment,favor quality not quantity.

Comments in the code

- Make sure comments and code are agree.
- Don't comment bad code - rewrite it.
- Use variable names that mean something.
- Don't over-comment,favor quality not quantity.
- Good comments explain why not how.

Comments in the code

- Make sure comments and code are agree.
- Don’t comment bad code - rewrite it.
- Use variable names that mean something.
- Don’t over-comment,favor quality not quantity.
- Good comments explain why not how.
- Don’t duplicate code in a comment.

Comments in the code

- Make sure comments and code are agree.
- Don't comment bad code - rewrite it.
- Use variable names that mean something.
- Don't over-comment,favor quality not quantity.
- Good comments explain why not how.
- Don't duplicate code in a comment.
- Format a program to help the reader understand it.

Comments in the code

- Make sure comments and code are agree.
- Don't comment bad code - rewrite it.
- Use variable names that mean something.
- Don't over-comment,favor quality not quantity.
- Good comments explain why not how.
- Don't duplicate code in a comment.
- Format a program to help the reader understand it.
- Indent to show the logical structure of a program.

Avoid magic numbers:example

Conversion of mass mixing ratio in units of g/kg to volume mixing ratio in units of ppmv

```
......
REAL(8) :: ppmv
REAL(8) :: Mixing_Ratio
REAL(8) :: Molecular_Weight
...
ppmv = 1.0e+03 * Mixing_Ratio * 28.9648 / Molecular_Weight
```

Avoid magic numbers:example

Conversion of mass mixing ratio in units of g/kg to volume mixing ratio in units of ppmv

```
......
REAL(8) :: ppmv
REAL(8) :: Mixing_Ratio
REAL(8) :: Molecular_Weight
...
ppmv = 1.0e+03 * Mixing_Ratio * 28.9648 / Molecular_Weight
```

```
REAL(8), PARAMETER :: G_TO_KG      = 1.0d-03
REAL(8), PARAMETER :: PPV_TO_PPMV  = 1.0d+06
REAL(8), PARAMETER :: SCALE_FACTOR = G_TO_KG * PPV_TO_PPMV
REAL(8), PARAMETER :: MW_DRYAIR    = 28.9648
...
REAL(8) :: ppmv
REAL(8) :: Mixing_Ratio
REAL(8) :: Molecular_Weight
...
ppmv = SCALE_FACTOR * Mixing_Ratio * MW_DRYAIR / Molecular_Weight
```

Avoid magic number

- Changes can be made more reliably.
- Changes can be made more easily.
- Your code is more readable.

- Checking assumptions.
- "assert" macro.
- Custom "assert" macros.
- Assertions about intermediate values.
- Preconditions,postcondition.

- Checking assumptions.
- "assert" macro.
- Custom "assert" macros.
- Assertions about intermediate values.
- Preconditions,postcondition.

Defensive programming.

Checking assumptions

- That an input parameter's value falls within its expected range (or an output parameters' value does)
- That the value of an input-only variable is not changed by a routine.
- That a pointer is non-NULL.
- That an array or other container passed into a routine can contain at least X data number of data elements.
- That a table has been initialized to contain real values.
- Etc.

Assertions

- An assertion is a code (usually routine or macro) that allows a program to check itself as runs.
- When an assertion is true, that means everything is operating as expected
- When it's false, that means it has detected an unexpected error in the code.
- Use error handling code for conditions you expected to occur.

Error handling technique

- Return a neutral value.
- Substitute the closest legal value.
- Log a warning message to a file.
- Return an error code.
  - Set a value of a status variable.
  - Return status as the function's return value.
  - Throw an exception using the language's build-in exception mechanism.
- Display an error message wherever the error is encountered.
- Handle the error in whatever way works best locally.
- Shutdown.

Rules of programming style

From **"The Elements of Programming Style"** Kernighan and Plauger

- Write clearly don't be too clever.
- Say what you mean, simply and directly.
- Use library functions whenever feasible.
- Avoid too many temporary variables.
- Write clearly - don't sacrifice clarity for "efficiency".
- Let the machine do the dirty work.
- Replace repetitive expressions by calls to common functions.
- Parenthesize to avoid ambiguity.
- Choose variables names that won't be confused.
- Avoid unnecessary branches.

Rules of programming style

- If a logical expression is hard to understand, try transforming it.
- Choose a data representation that makes the program simple.
- Write first in easy-to-understand pseudo language; then translate into whatever language you have to use.
- Modularize. Use procedures and functions.
- Avoid *goto*s completely if you keep the program readable.
- Don't patch bad code - rewrite it.
- Write and test a big program in small pieces.
- Use recursive procedures for recursively-defined data structures.
- Test input for plausibility and validity.
- Make sure input doesn't violate the limits of the program.

Rules of programming style

- Terminate input by end-of-file marker, not by count.
- Identify bad input; recover if possible.
- Make input easy to prepare and output self-explanatory.
- Use uniform input formats.
- Make input easy to proofread.
- Use self-identifying input. Allow defaults. Echo both on output.
- Make sure all variables are initialized before use.
- Don't stop at one bug.
- Use debugging compilers.
- Watch out for off-by-one errors.

Rules of programming style

- Take care to branch the right way on equality.
- Avoid multiple exits from the loops.
- Make sure your code does "nothing" gracefully.
- Test programs at their boundary values.
- Check some answers by hand.
- 10.0 times 0.1 is hardly ever 1.0.
- 7/8 is zero while 7.0/8.0 is not zero.
- Don't compare floating point numbers solely equality.
- Make it right before you make it faster.
- Make it fail-safe before you make faster.

Rules of programming style

- Make it clear before you make it faster.
- Don’t sacrifice clarity for small gains of "efficiency".
- Let your compiler do the simple optimizations.
- Don’t strain to re-use code; reorganize instead.
- Make sure special cases are truly special.
- Keep it simple to make it faster.
- Don’t diddle code to make it faster - find a better algorithm.
- Instrument yout programs. Measure before making "efficiency" changes.
- Make sure comments and code agree.
- Don’t just echo the code with comments - make every comment count.

Rules of programming style

- Don’t comment bad code - rewrite it.
- Use variable names taht mean something.
- Use statement labels that mean something.
- Format a program to help the reader understand it.
- Document your data layouts.
- Don’t over-comment.

Outline

Bugs and Prevention

Testing

Static analysis

Run-time analysis

Debugging

Parallel debugging

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
Equilater.

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
Equilater.

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
Equilater.

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

Testing: a simple program

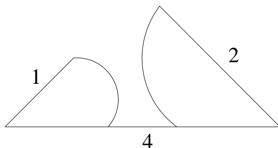
Q:1 Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third ?

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.

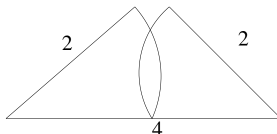
Patch definition of valid triangles:
$a < b + c$ $b < a + c$ and $c < a + b$

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:
$a < b+c$ and $b < a+c$ and $c < a+b$

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)

Q: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 possibile errors would be found.
- An adeguate test of this program should expose at least these errors.
- Higly qualified professional programmers score, on the average, 7.8 out of a possibile 14.

Classification of errors

Syntax

- Definition: errors in grammar (violations of the "rules" for forming legal language statements).
- Examples: undeclared identifiers, mismatched parentheses in an expression, an opening brace without a matching closing brace, etc.
- Occur: an error that is caught in the process of compiling the program.

Classification of errors

Syntax

- Definition: errors in grammar (violations of the "rules" for forming legal language statements).
- Examples: undeclared identifiers, mismatched parentheses in an expression, an opening brace without a matching closing brace, etc.
- Occur: an error that is caught in the process of compiling the program.
  The easiest errors to spot by a compiler or some aid in checking the syntax.
  As you get more practice using a language, you naturally make fewer errors, and will be able to quickly correct those that do occur.

Classification of errors

Runtime

- Definition: "Asking the computer to do the impossible!"
- Examples: division by zero, taking the square root of a negative number, referring to the 101th on a list of only 100 items, deferencing a null pointer, etc.
- Occur: an error that is not detected until the program is executed, and then causes a processing error to occur.

Classification of errors

Runtime

- Definition: "Asking the computer to do the impossible!"
- Examples: division by zero, taking the square root of a negative number, referring to the 101th on a list of only 100 items, deferencing a null pointer, etc.
- Occur: an error that is not detected until the program is executed, and then causes a processing error to occur.
  They are not easy to spot because they are not syntax or grammar errors, they are subtle errors and develop in the course of program's execution.
  Avoiding exceptions and correcting program behaviour is also largely a matter of experience.
  To prevent use defensive programming.

Classification of errors

Logic (semantic, meaning)

- Definition: the program compiles (no syntax errors) and runs to a normal completion (no runtime erros), but the output is wrong. A statement may be syntactically correct, but mean something other than what we intended. Therefore it has a different effect, causing the program output to be wrong.
- Examples: improper initialization of variables, performing arithmetic operations in the wrong order, using an incorrect formula to compute a value, etc.
- Occur: an error that affect how the code works.

Classification of errors

Logic (semantic, meaning)

- Definition: the program compiles (no syntax errors) and runs to a normal completion (no runtime erros), but the output is wrong. A statement may be syntactically correct, but mean something other than what we intended. Therefore it has a different effect, causing the program output to be wrong.
- Examples: improper initialization of variables, performing arithmetic operations in the wrong order, using an incorrect formula to compute a value, etc.
- Occur: an error that affect how the code works.
  It is a type of error that only the programmer can recognize.Finding and correcting logic errors in a program is known as debugging.

Outline

- Bugs and Prevention
- Testing
- Static analysis
  - Compiler options
  - Static analyzer
- Run-time analysis
- Debugging
- Parallel debugging

Definitions

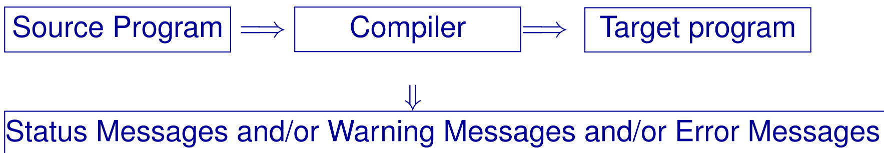
Static analysis refers to a method of examinig software that allows developers to discover dangerous programming pratices or potential errors in source code, without actually run the code.

Definitions

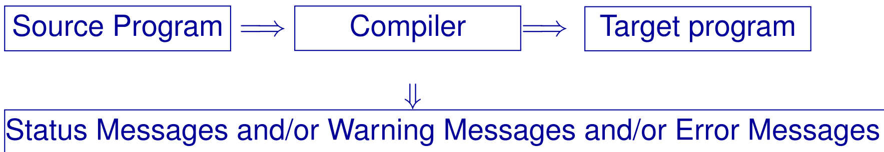
Static analysis refers to a method of examinig software that allows developers to discover dangerous programming pratices or potential errors in source code, without actually run the code.

- Using compiler options.
- Using static analyzer.

Compiler and errors



Compiler and errors



Compiler checks:

- Syntax.
- Semantic.

Lexical (Syntactic) errors

- Characters in the source that aren't in the alphabet of the language.
- Words in the source that aren't in the vocabulary of the language.

Syntactic errors

- Comment delimiters that have been put in wrong place or omitted.
- Literal delimiters that have been put in wrong place or omitted.
- Keywords that have been misspelled.
- Required punctuation that is missing.
- Construct delimiters such as parentheses or braces that have been misplaced.
- Blank or tab characters that are missing.
- Blank or tab characters that shouldn’t occur where they’re found.

Semantic errors

- Names that aren't declared.
- Operands of the wrong type for the operator they're used with.
- Values that have the wrong type for the name to which they're assigned.
- Procedures that are invoked with the wrong number of arguments.
- Procedures that are invoked with the wrong type of arguments.
- Function return values that are the wrong type for the context in which they're used.

Semantic errors

- Code blocks that are unreachable.
- Code blocks that have no effect.
- Local variables that are used before being initialized or assigned.
- Local variables that are initialized or assigned but not used.
- Procedures that are never invoked.
- Procedures that have no effect.
- Global variables that are used before being initialized or assigned.
- Global variables that are initialized or assigned, but not used.

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.

Outline

Bugs and Prevention

Testing

Static analysis
- Compiler options
- Static analyzer

Run-time analysis

Debugging

Parallel debugging

Static analysis:example 1 part one


```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f\n", 4);
5    return 0;
6  }
```

Static analysis:example 1 part one


```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f\n", 4);
5    return 0;
6  }
```

```
<ruggiero@shiva ~/CODICI>gcc bad.c
```

```
<ruggiero@shiva ~/CODICI>./a.out
```

```
Two plus two is 0.000000
```

Static analysis:example 1 part two


```
<ruggiero@shiva ~/CODICI>gcc -Wall bad.c
```

```
bad.c: In function main:
bad.c:4: warning: format ’%f’ expects type ’double’,
but argument 2 has type ’int’
```

Static analysis:example 1 part two


```
<ruggiero@shiva ~/CODICI>gcc -Wall bad.c
```

```
bad.c: In function main:
bad.c:4: warning: format '%f' expects type 'double',
but argument 2 has type 'int'
```

```
1 #include <stdio.h>
2 int main (void)
3 {
4   printf ("Two plus two is %f \n", 4);
5   return 0;
6 }
```

Static analysis:example 1 part two


```
<ruggiero@shiva ~/CODICI>gcc -Wall bad.c
```

```
bad.c: In function main:
bad.c:4: warning: format '%f' expects type 'double',
but argument 2 has type 'int'
```

```
1 #include <stdio.h>
2 int main (void)
3 {
4   printf ("Two plus two is %d\n", 4);
5   return 0;
6 }
```

gcc options

-Wall

turns on the most commonly-used compiler warnings option

-Waddress -Warray-bounds (only with -O2) -Wc++0x-compat -Wchar-subscripts -Wimplicit-int -Wimplicit-function-declaration -Wcomment -Wformat -Wmain (only for C/ObjC and unless -ffreestanding) -Wmissing-braces -Wnonnull -Wparentheses -Wpointer-sign -Wreorder -Wreturn-type -Wsequence-point -Wsign-compare (only in C++) -Wstrict-aliasing -Wstrict-overflow=1 -Wswitch -Wtrigraphs -Wuninitialized -Wunknown-pragmas -Wunused-function -Wunused-label -Wunused-value -Wunused-variable -Wvolatile-register-var

Oher compilers

```
[ruggiero@matrix1 ~]$ pgcc bad.c
```

Oher compilers


```
[ruggiero@matrix1 ~]$ pgcc bad.c
```

```
[ruggiero@matrix1 ~]$ icc bad.c
```

```
bad.c(4): warning #181: argument is incompatible
with corresponding format string conversion
    printf ("Two plus two is %f\n", 4);
                                    ^
```

check.c:source


```
1   #include <stdio.h>
2
3   int main (void)
4   {
5               return 0;
6   }
7   void checkVal(unsigned int n) {
8               if (n < 0) {
9                           /* Do something... */
10                      }
11                else if (n >= 0) {
12                          /* Do something else... */
13                      }
14  }
```

SuperComputing Applications and Innovation

check.c:compilation

Sapienza Università di Roma

```
<ruggiero@shiva:~> gcc -Wall check.c
```

check.c:compilation


```
<ruggiero@shiva:~> gcc -Wall check.c
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra check.c
```

```
check.c: In function ?checkVal?:
check.c:8: warning: comparison of unsigned expression < 0 is always false
check.c:11: warning: comparison of unsigned expression >= 0 is always true
```

gcc options

-Wextra (-W)
reports the most common programming errors and less-serious but potential problem

```
-Wclobbered -Wempty-body
-Wignored-qualifiers -Wmissing-field-initializers
-Wmissing-parameter-type (C only) -Wold-style-declaration (C only)
-Woverride-init -Wsign-compare
-Wtype-limits -Wuninitialized (only with -O1 and above)
-Wunused-parameter (only with -Wunused or -Wall)
```

check1.c:source


```
1   #include <stdio.h>
2    int main (void)
3    {
4            double x = 10.0;
5            double y = 11.0;
6            double z = 0.0;
7            if  (x == y)  {
8                    z = x * y;
9            }
10             return 0;
11   }
```

check1.c:source


```
1   #include <stdio.h>
2    int main (void)
3    {
4            double x = 10.0;
5            double y = 11.0;
6            double z = 0.0;
7            if (x == y) {
8                    z = x * y;
9            }
10             return 0;
11   }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra  check1.c
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra -Wfloat-equal check1.c
```

```
check1.c: In function main:
check1.c:8: warning: comparing  floating point
with == or != is unsafe
```

gcc: other compiler options

-Wno-div-by-zero -Wsystem-headers -Wfloat-equal -Wtraditional (C only)
-Wdeclaration-after-statement (C only) -Wundef -Wno-endif-labels -Wshadow
-Wlarger-than-len -Wpointer-arith -Wbad-function-cast (C only) -Wcast-align
-Wwrite-strings -Wconversion -Wsign-compare -Waggregate-return -Wstrict-prototypes
(C only) -Wold-style-definition (C only) -Wmissing-prototypes (C only)
-Wmissing-declarations (C only) -Wcast-qual -Wmissing-field-initializers
-Wmissing-noreturn -Wmissing-format-attribute -Wno-multichar
-Wno-deprecated-declarations -Wpacked -Wpadded -Wredundant-decls
-Wnested-externs (C only) -Wvariadic-macros -Wunreachable-code -Winline
-Wno-invalid-offsetof (C++ only) -Winvalid-pch -Wlong-long -Wdisabled-optimization
-Wno-pointer-sign

testinit.c:source


```
1  int main() {
2          int v[16];
3          int i,j,k;
4          j=i;
5          v[i]= 42;
6          return 0 ;
7  }
```

testinit.c:source


```
1  int main() {
2          int v[16];
3          int i,j,k;
4          j=i;
5          v[i]= 42;
6          return 0 ;
7  }
```

```
ruggiero@shiva:~> gcc -Wall -Wextra testinit.c
```

testinit.c:source


```
1  int main() {
2          int v[16];
3          int i,j,k;
4          j=i;
5          v[i]= 42;
6          return 0 ;
7  }
```

```
ruggiero@shiva:~> gcc -Wall -Wextra testinit.c
```

```
ruggiero@shiva:~> gcc -O1 -Wall -Wextra -Wuninitialized testinit.c
```

```
testinit.c: In function main
testinit.c:4: warning: unused variable k
testinit.c:5: warning: i is used uninitialized
in this function
```

testinit.c:source


```
1  int main() {
2          int v[16];
3          int i,j,k;
4          j=i;
5          v[i]= 42;
6          return 0 ;
7  }
```

```
ruggiero@shiva:~> gcc -Wall -Wextra testinit.c
```

```
ruggiero@shiva:~> gcc -O1 -Wall -Wextra -Wuninitialized testinit.c
```

```
testinit.c: In function main
testinit.c:4: warning: unused variable k
testinit.c:5: warning: i is used uninitialized
in this function
```

```
[ruggiero@matrix1 ~]$ icc testinit.c
```

```
testinit.c(5): warning #592: variable "i" is used
               before its value is set j=i;
```

Compilation: default options


```
program par
implicit none
integer,parameter  :: hacca=10

call sub(hacca)

end

subroutine sub(hacca)
implicit none
integer hacca

hacca=hacca+1
write(*,*) hacca

return

end
```

Compilation: default options


```
> xlf par.f -o par.x
```

```
> ./par.x
```

```
11
```

Compilation: default options


```
> xlf par.f -o par.x
```

```
> ./par.x
```

```
11
```

```
<ruggiero@ife2 ~>ifort par.f -o par.x
```

Compilation: default options


```
> xlf par.f -o par.x
```

```
> ./par.x
```

```
11
```

```
<ruggiero@ife2 ~>ifort par.f -o par.x
```

```
<ruggiero@ife2 ~> ./par.x
```

```
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
par.x              0000000000402688  Unknown            Unknown     Unknown
par.x              0000000000402670  Unknown            Unknown     Unknown
par.x              000000000040262A  Unknown            Unknown     Unknown
libc.so.6          00000036FF81C40B  Unknown            Unknown     Unknown
par.x              000000000040256A  Unknown            Unknown     Unknown
```

Compilation: default options


```
<ruggiero@ife2 ~> man ifort
```

```
? -assume noprotect_constants

Tells the compiler to pass a copy of a constant actual argument.
This copy  can  be  modified  by  the called routine, even though
the Fortran standard prohibits such modification. The calling
routine does not see any modification to the constant. The default
is -assume protect_constants, which  passes  the constant actual
argument.  Any attempt to modify it results in an error.
```

Compilation: default options


```
<ruggiero@ife2 ~> man ifort
```

```
? -assume noprotect_constants

Tells the compiler to pass a copy of a constant actual argument.
This copy  can  be  modified  by  the called routine, even though
the Fortran standard prohibits such modification. The calling
routine does not see any modification to the constant. The default
is -assume protect_constants, which  passes  the constant actual
argument.  Any attempt to modify it results in an error.
```

```
<ruggiero@ife2 ~>ifort par.f -assume noprotect_constants -o par.x
```

Compilation: default options


```
<ruggiero@ife2 ~> man ifort
```

```
? -assume noprotect_constants

Tells the compiler to pass a copy of a constant actual argument.
This copy  can  be  modified  by  the called routine, even though
the Fortran standard prohibits such modification. The calling
routine does not see any modification to the constant. The default
is -assume protect_constants, which  passes  the constant actual
argument.  Any attempt to modify it results in an error.
```

```
<ruggiero@ife2 ~>ifort par.f -assume noprotect_constants -o par.x
```

```
<ruggiero@ife2 ~>./par.x
```

```
11
```

Outline

Bugs and Prevention

Testing

Static analysis
- Compiler options
- Static analyzer

Run-time analysis

Debugging

Parallel debugging

splint

- Open Source Static Analysis Tool developed at University of Virginia by Professor Dave Evans
- Based on Lint
- www.splint.org
- splint [-option -option ...] filename [filename ...]

splint can detect with just source code...

- Unused declarations.
- Type inconsistencies.
- Variables used before being assigned.
- Function return values that are ignored.
- Execution paths with no return.
- Switch cases that fall through.
- Apparent infinite loops.

splint can detect with annotation information

- Dereferencing pointers with possible null values.
- Using storage that is undefined or partly undefined.
- Returning storage that is undefined or partly defined.
- Type mismatches.
- Using deallocated storage.

splint can detect with annotation information

- Memory leaks.
- Inconsistent modification of caller visible states.
- Violations of information hiding.
- Undefined program behaviour due to evaluation order, incomplete logic, infinite loops, statements with no effect, and so on.
- Problematic uses of macros.

Static analysis:example 1 part one


```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f\n", 4);
5    return 0;
6  }
```

check.c:source


```
1   #include <stdio.h>
2
3   int main (void)
4   {
5             return 0;
6   }
7   void checkVal(unsigned int n) {
8             if (n < 0) {
9                         /* Do something... */
10                      }
11             else if (n >= 0) {
12                         /* Do something else... */
13                       }
14  }
```

splint:examples


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint bad.c
```

```
Finished checking --- no warnings
```

splint:examples


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint bad.c
```

```
Finished checking --- no warnings
```

```
<ruggiero@shiva:~> splint-3.1.2/bin/splint check.c
```

```
splint 3.1.2 --- 28 Mar 2008
check.c: (in function checkVal)
check.c:8:15: Comparison of unsigned value involving zero: n < 0
 An unsigned value is used in a comparison with zero in a way that is
 a bug or confusing. (Use -unsignedcompare to inhibit warning)
check.c:11:22: Comparison of unsigned value involving zero: n >= 0
Finished checking --- 2 code warnings
```

check1.c:source


```
1  #include <stdio.h>
2   int main (void)
3   {
4           double x = 10.0;
5           double y = 11.0;
6           double z = 0.0;
7           if (x == y) {
8                   z = x * y;
9           }
10            return 0;
11  }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra  check1.c
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra -Wfloat-equal check1.c
```

```
check1.c: In function main:
check1.c:8: warning: comparing  floating point
with == or != is unsafe
```

splint:examples


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint check1.c
```

```
Splint 3.1.2 --- 28 Mar 2008

check1.c: (in function main)
check1.c:8:14: Dangerous equality comparison involving
double types: x == y Two real (float, double, or long double)
values are compared directly using == or != primitive. This
may produce unexpected results since floating point representations
are inexact. Instead, compare the difference to FLT_EPSILON
or DBL_EPSILON. (Use -realcompare to inhibit warning)

Finished checking --- 1 code warning
```

check1.c: rewritten


```
1   int main (void)
2     {
3             double x = 10.0;
4             double y = 11.0;
5             double z = 0.0;
6             double norma=0.1e-16;
7             double epsilon;
8            epsilon=x-y;
9            if  (epsilon < norma) {
10                       z = x * y;
11           }
12
13             return 0;
14   }
```

average.c: source

```
1  #include <stdio.h>
2  int main (void) {
3    int size = 5;
4    int a;
5    float total;
6    float art[size];
7
8   for(a = 0 ; a < size; ++ a) {
9     art[a] = (float) a;
10    total  += art[a]; }
11   printf(" %f\n" , total / (float) size);
12   return 0;
13 }
```

average.c: compilation e run


```
<ruggiero@shiva:~> gcc -Wall -Wextra average.c
```

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

```
1.999994
```

average.c: compilation e run


```
<ruggiero@shiva:~> gcc -Wall -Wextra average.c
```

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

```
1.999994
```

```
<ruggiero@shiva:~> gcc -Wall  -Wextra -O2 average.c
```

```
average.c: In function main:
average.c:5: warning: total may be used uninitialized
in this function
```

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

```
nan
```

average.c: splint


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint average.c
```

```
Splint 3.1.2 --- 28 Mar 2008

average.c: (in function main)
average.c:10:4 Variable total used before definition
An rvalue is used that may not be initialized to a value
on some execution path. (Use -usedef to inhibit warning)
average.c:11:20: Variable total used before definition

Finished checking --- 2 code warnings
```

average.c: new version


```
1  #include <stdio.h>
2  int main (void) {
3     int size = 5;
4     int a;
5     float total;
6     float art[size];
7      total=0;
8   for(a = 0 ; a < size; ++ a) {
9      art[a] = (float) a;
10     total  += art[a]; }
11    printf(" %f\n" , total / (float) size);
12    return 0;
13 }
```

assign.c: source and compilation

```
1  #include <stdio.h>
2  main()
3  {
4        int a=0;
5        while (a=1)
6             printf("hello\n");
7        return 0;
8  }
```

assign.c: source and compilation


```
1  #include <stdio.h>
2  main()
3  {
4         int a=0;
5         while (a=1)
6              printf("hello\n");
7         return 0;
8  }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra assign.c
```

```
assigna.c:3: warning: return type defaults to int
assign.c: In function main:
assign.c:5: warning: suggest parentheses around assignment used as
truth value
```

assign.c: splint

```
<ruggiero@shiva:~>splint-3.1.2/bin/splint assign.c
```

```
Splint 3.1.2 --- 28 Mar 2008
assign.c: (in function main)
assign.c:5:14: Test expression for while is assignment expression:a=1
The condition test is an assignment expression. Probably, you mean
to use == instead of =. If an assignment is intended, add an extra
parentheses nesting (e.g., if ((a = b)) ...) to suppress this message.
(Use -predassign to inhibit warning)
assign.c:5:14: Test expression for while not boolean, type int: a=1
Test expression type is not boolean or int. (Use -predboolint
to inhibit warning)
Finished checking --- 2 code warnings
```

memory.c: source and compilation


```
1
2   #include <stdlib.h>
3   int main()
4   {
5           int *p = malloc(5*sizeof(int));
6
7
8
9           *p = 1;
10          free(p);
11          return 0;
12  }
```

memory.c: source and compilation


```
1
2  #include <stdlib.h>
3  int main()
4  {
5          int *p = malloc(5*sizeof(int));
6
7
8
9          *p = 1;
10         free(p);
11         return 0;
12 }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra memory.c
```

memory.c: splint

```
<ruggiero@shiva:~> splint-3.1.2/bin/splint memory.c
```

```
Splint 3.1.2 --- 28 Mar 2008

memory.c: (in function main)
memory.c:9:10: Dereference of possibly null pointer p: *p
A possibly null pointer is dereferenced.  Value is either
the result of a function which may return null (in which
case, code should check it is not null), or a global,
parameter or structure field declared with the null
qualifier. (Use -nullderef to inhibit warning)
memory.c:5:18: Storage p may become null

Finished checking --- 1 code warning
```

memory.c: new version


```c
1  #include <stdlib.h>
2  #include <stdio.h>
3  int main()
4  {
5          int *p = malloc(5*sizeof(int));
6          if (p == NULL) {
7              fprintf(stderr, "error in malloc");
8              exit(EXIT_FAILURE);
9          } else  *p = 1;
10         free(p);
11         return 0;
12 }
```

CINECA SCAI SuperComputing Applications and Innovation

out_b.c: source

SAPIENZA UNIVERSITÀ DI ROMA

```
1   #include<stdio.h>
2   #define N 5
3   int main (void)
4   {
5           int t[N];
6           int i;
7
8           i=6;
9           t[i] = i+1;
10
11          return 0;
12  }
```

CINECA

out_b.c: splint


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint out_b.c
```

out_b.c: splint


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint out_b.c
```

```
<ruggiero@shiva:~> splint-3.1.2/bin/splint +bounds out_b.c
```

```
Splint 3.1.2 --- 28 Mar 2008

out_b.c: (in function main)
out_b.c:9:9: Likely out-of-bounds store: t[i]
    Unable to resolve constraint:
    requires 4 >= 6
     needed to satisfy precondition:
    requires maxSet(t @ out_b.c:9:9) >= i @ out_b.c:9:11
  A memory write may write to an address beyond the allocated buffer. (Use
  -likelyboundswrite to inhibit warning)

Finished checking --- 1 code warning
```

error.f: source


```
1            REAL FUNCTION COMPAV(SCORE,COUNT)
2                INTEGER SUM,COUNT,J,SCORE(5)
3                DO 30 I = 1,COUNT
4                    SUM = SUM + SCORE(I)
5    30          CONTINUE
6                COMPAV = SUM/COUNT
7                write(*,*) compav
8            END
9            PROGRAM AVENUM
10             PARAMETER(MAXNOS=10)
11             INTEGER I, COUNT
12             REAL NUMS(MAXNOS), AVG
13             COUNT = 0
14             DO 80 I = 1,MAXNOS
15                 READ (5,*,END=100) NUMS(I)
16                 COUNT = COUNT + 1
17  80          CONTINUE
18  100         AVG = COMPAV(NUMS, COUNT)
19           END
```

error.f: compilation


```
<ruggiero@ife2 /ftnchek>gfortran -Wall -Wextra -O2 error.f
```

```
In file error.f:2

             INTEGER SUM,COUNT,J,SCORE(5)
                                 1
CWarning: Unused variable j declared at (1)
error.f: In function compav:
error.f:2: warning: sum may be used uninitialized in this function
```

error.f: compilation


```
<ruggiero@ife2/ftnchek>pgf90 -Minform=inform error.f
```

```
PGF90-I-0035-Predefined intrinsic sum loses intrinsic property (error.f: 5)
PGF90-I-0035-Predefined intrinsic count loses intrinsic property (error.f: 16)
```

error.f: compilation


```
<ruggiero@ife2 /ftnchek>ifort -warn all error.f
```

```
fortcom: Warning: error.f, line 4: This name has not been given an explicit type.   [I]
             DO 30 I = 1,COUNT
-------------------^
fortcom: Info: error.f, line 2: This variable has not been used.   [J]
             INTEGER SUM,COUNT,J,SCORE(5)
-------------------------------^
fortcom: Warning: error.f, line 13: This name has not been given an explicit type.   [MAXNOS]
           PARAMETER(MAXNOS=10)
---------------------^
fortcom: Warning: error.f, line 21: This name has not been given an explicit type.   [COMPAV]
100         AVG = COMPAV(NUMS, COUNT)
------------------^
```

error.f: compilation

```
<ruggiero@ife2 /ftnchek>g95 -Wall -Wextra error.f
```

```
In file error.f:6

 30          CONTINUE
                     1
Warning (142): Nonblock DO statement at (1) is obsolescent
In file error.f:1

      REAL FUNCTION COMPAV(SCORE,COUNT)
                           1
Warning (163): Actual argument 'score' at (1) does not have an INTENT
In file error.f:1

      REAL FUNCTION COMPAV(SCORE,COUNT)
                                 1
Warning (163): Actual argument 'count' at (1) does not have an INTENT
In file error.f:2

             INTEGER SUM,COUNT,J,SCORE(5)
                               1
Warning (137): Variable 'j' at (1) is never used and never set
```

error.f: compilation


```
In file error.f:20

80          CONTINUE
                    1
Warning (142): Nonblock DO statement at (1) is obsolescent
In file error.f:21

100         AVG = COMPAV(NUMS, COUNT)
                  1
Warning (165): Implicit interface 'compav' called at (1)
In file error.f:15

           REAL NUMS(MAXNOS), AVG
                             1
Warning (112): Variable 'avg' at (1) is set but never used
In file error.f:21

100         AVG = COMPAV(NUMS, COUNT)
                         1
In file error.f:1

      REAL FUNCTION COMPAV(SCORE,COUNT)
                               2
Warning (155): Inconsistent types (REAL(4)/INTEGER(4))
in actual argument lists at (1) and (2)
```

```
<ruggiero@ife2/ftnchek>./bin/ftnchek error.f
```

```
FTNCHEK Version 3.3 November 2004

File error.f:
      7                COMPAV = SUM/COUNT
                              ^
Warning near line 7 col 21 file error.f:
 integer quotient expr SUM/COUNT  converted to real

Warning in module COMPAV in file error.f:
 Variables declared but never referenced:
    J declared at line 2 file error.f

Warning in module COMPAV in file error.f:
 Variables may be used before set:
    SUM used at line 5 file error.f
    SUM set at line 5 file error.f

Warning in module AVENUM in file error.f:
 Variables set but never used:
    AVG set at line 21 file error.f
```

www.dsm.fordham.edu/ftnchek


```
 0 syntax errors detected in file error.f
 4 warnings issued in file error.f

Warning: Subprogram COMPAV argument data type mismatch at position 1:
    Dummy arg SCORE in module COMPAV line 1 file
 error.f is type intg
    Actual arg NUMS in module AVENUM line 21 file
 error.f is type real

Warning: Subprogram COMPAV argument arrayness mismatch at position 1:
    Dummy arg SCORE in module COMPAV line 1 file
 error.f has 1 dim size 5
    Actual arg NUMS in module AVENUM line 21 file
 error.f has 1 dim size 10
```

Static analyzer for Fortran

- Forcheck can handle up Fortran 95 and some Fortran 2003. item Cleanscape FortranLint can handle up to Fortran 95.
- 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 occurence 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

- Bugs and Prevention
- Testing
- Static analysis
- Run-time analysis
  - Memory checker
- Debugging
- Parallel debugging

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  }
```

CINECA SCAI SuperComputing Applications and Innovation

ABORT example

SAPIENZA UNIVERSITÀ DI ROMA

```
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
```

CINECA

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: s 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'.
```

CINECA SCAI SuperComputing Applications and Innovation

Fortran Results

SAPIENZA UNIVERSITÀ DI ROMA

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
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
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
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

CINECA

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
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
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
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	AIoB	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
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
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
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
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
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
sun-12.1	0.0666667	0.0166667	0.05	0	0
sun-12.1+bcheck	1.13333	0.025	0	0	0

SCAI SuperComputing Applications and Innovation

C++ Results

SAPIENZA UNIVERSITÀ DI ROMA

Compiler	AD	AIoB	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
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
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
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

CINECA

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

Bugs and Prevention

Testing

Static analysis

Run-time analysis
- Memory checker

Debugging

Parallel debugging

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 through.
- 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.

Why should I use Valgrind?

- Valgrind will tell you about tough to find bugs.
- Valgrind is very through.
- 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 interpeter. 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: 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: 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 dealloacate 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 dealloacate 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 dealloacate 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: 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_memche
==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: 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: 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 dinamically 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: vfprintf (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: vfprintf (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==
```

CINECA SCAI SuperComputing Applications and Innovation

outbc.c

SAPIENZA UNIVERSITÀ DI ROMA

```
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         }
```

CINECA

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

- Bugs and Prevention
- Testing
- Static analysis
- Run-time analysis
- Debugging
  - gdb
- Parallel debugging

My program fails!

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

Assumption 1

Question: The same code still worked last week.

- Possible problem: An application runs in an environment with many components (OS, dynamic libraries, network, disks, ..). If any one has changed, this can impact the application.
- Fix: ask the helpdesk if anything changed since last week.
- Alternative problem: Often, the code is not exactly the same, only the user assumed that the change was not important.
- Fix: a version control system (cvs, svn, git, ..) will help you check that the code really was the same as last week or to see what changes were made.

Assumption 2

Question: The same program works for my colleague.

- Possible problem: do you really use the same executable?
- Check: use

```
which [command]
```

to see the location.

- Possible problem: dynamic executables load libraries at runtime. These are loaded from directories in your $*LD_LIBRARY_PATH*. Maybe you use different versions of libraries without knowing it?
- Check: use

```
ldd [program]
```

to check which libraries are used.

Assumption 3

Question: the program works fine on my own system, so something is wrong with yours

```
program helloworld
implicit none
print*,'Hello World!'
return
end program
```

- works fine with GNU and IBM compilers, but won't compile with the Intel compiler. Why?
- return statement is not allowed in Fortran mainprogram

```
> ifort return.f90
return.f90(5): error #6353: A RETURN statement is invalid in
the main program.
```

- Solution: check your program to see if it follows the language standard.

Assumption 4

Question: My program crashed. What did I do wrong?

- Answer: Your program depends on other libraries, the compiler, the OS, the network, etc. System libraries and compilers have bugs and hardware can fail, so it might not be your program's fault.

```
CALL MPI_Barrier(MPI_COMM_SELF, IERR)
```

- This call would segfault in some version of IBM MPI, although it is a correct MPI call.
- Solutions: read Changelogs or KnownBugs. Isolate the problem and send it to the helpdesk. Ask if it could be a known bug.
- However, 99% of the problems are related to the application.

Assumption 5

Question: It works only sometimes.

- Problem: Probably a race condition (bugs that cause undefined behaviour, depending on timing differences)

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 syinchronized 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.

What is a Debuggers?

Debuggers are a software tools that help determine why program does not behave correctly. They aid a programmer in understanding a program and then finding the cause of the discrepancy. The programmer can then repair the defect and so allow the program to work according to its original intent. A debugger is a tool that controls the application being debugged so as to allow the programmer to follow the flow of program execution and , at any desired point, stop the program and inspect the state of the program to verify its correctness.

"How debuggers works Algorithms, data,structure, and Architecture"

Jonathan B. Rosemberg

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 occuring).
- 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.
- Loss of data.
- 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

- Bugs and Prevention
- Testing
- Static analysis
- Run-time analysis
- Debugging
  - gdb
- Parallel debugging

What is gdb?

- The GNU Project debugger, is an open-source debugger.
- Released 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.
- 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`

prime-numbers 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 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[I]* will be 1 if *I* is prime, 0 otherwise.

Main.c


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

CheckPrime.c


```
#define MaxPrimes 50
extern int Prime[MaxPrimes];
void CheckPrime(int K)
{
    int J; J=2;
    while (1)  {
        if (Prime[J] == 1)
            if (K % J == 0)  {
                Prime[K] = 0;
                return;
            }
            J++;
    }
    Prime[K] = 1;
}
```

Compilation and run


```
<~>gcc Main.c CheckPrime.c  -o trova_primi
```

```
<~> ./trova_primi
```

```
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.

Starting gdb


```
<~>gcc Main.c CheckPrime.c -g  -o trova_primi
```

```
<~>gdb trova_primi
```

```
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

- run (r): start debugged program.
- kill (k): kill the child process in which program is running under gdb.
- where, backtrace (bt): print a backtrace of entire stack.
- quit(q): exit gdb.

- break (b) : set breakpoint at specified line or function

- print (p) expr: print value of expression expr
- display expr: print value of expression expr each time the program stops.

gdb: Basic commands

- continue: continue program being debugged, after signal or breakpoint.
- next: step program, proceeding through subroutine calls.
- step: step program until it reaches a different source line

- help (h): print list of commands.
- she: execute the rest of the line as a shell command.
- list (l) linenum: print lines centered around line number lineum in the current source file.

prime-number


```
(gdb) r
```

```
Starting program: trova_primi
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
```

```
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


```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7                if (Prime[J] == 1)
```

prime-number finding program


```
(gdb) p J
```

```
$1 = 1008
```

```
(gdb) l CheckPrime.c:7
```

```
2        extern int Prime[MaxPrimes];
3          CheckPrime(int K)
4          {
5              int J; J=2;
6              while (1)  {
7                 if (Prime[J] == 1)
8                    if (K % J == 0)   {
9                       Prime[K] = 0;
10                      return;
11                   }
```

CheckPrime.c:new version


```
1  #define MaxPrimes 50
2  extern int Prime[MaxPrimes];
3  void CheckPrime(int K)
4     {
5        int J; J = 2;
6         while (1){
7             if (Prime[J] == 1)
8                if (K % J == 0)   {
9                   Prime[K] = 0;
10                  return;
11               }
12        J++;
13        }
14        Prime[K] = 1;
15    }
```

CheckPrime.c:new version


```
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
13
14        Prime[K] = 1;
15     }
```

prime-number finding program


```
(gdb) kill
```

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

```
(gdb) she gcc -g Main.c CheckPrime.c -o triva_primi
```

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program exited normally.
```

gdb commands


```
(gdb) help break
```

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

```
(gdb) help display
```

```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

gdb commands

```
(gdb) help step
```

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

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

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

prime-number finding program

```
(gdb) r

Starting program: trova_primi
Breakpoint 1, main () at Main.c:7
9               printf("enter upper bound\n");

(gdb) next

10              scanf("%d",&UpperBound);

(gdb) next
20
11              Prime[2] = 1;

(gdb) next

12             for (N = 3; N <= UpperBound; N += 2)

(gdb) next

14                  CheckPrime(N);
```

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

13        }
```

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
```

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

prime-number finding program


```
(gdb) l  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


```
(gdb) kill
```

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

```
(gdb) she gcc -g Main.c CheckPrime.c -o trova_primi
```

```
(gdb) d
```

Delete all breakpoints? (y or n) y

prime-number finding program


```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

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
```

```
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
```

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

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /home/guest/Main.c:8
8             scanf("%d",&UpperBound);
```

```
(gdb) next
```

20

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

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 fo 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.pid
```

Graphical Debuggers

- gdb -tui or gdbtui (text user interface)
- ddd (data dysplay debugger) is a graphical front-end for command-line debuggers.
- allinea 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.
- Rouge Wave Totalview
- Etc.

SCAI SuperComputing Applications and Innovation

Outline

SAPIENZA UNIVERSITÀ DI ROMA

Bugs and Prevention

Testing

Static analysis

Run-time analysis

Debugging

Parallel debugging
- Totalview

Outline

Bugs and Prevention

Testing

Static analysis

Run-time analysis

Debugging

Parallel debugging
- Totalview

Totalview

- 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 Tv

- 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.

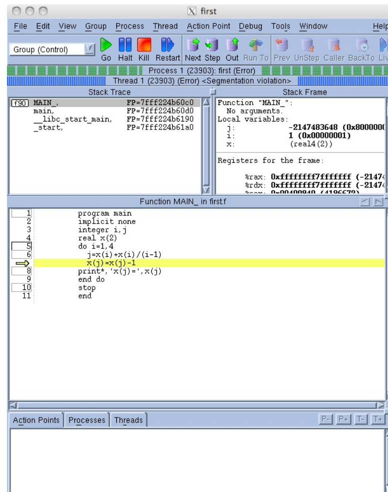
```
gcc [option]  -g file_source.c -o filename
```

Starting Totalview

Command	Action
totalview	Starts the debugger. You can then load a program or corefile, or else attach to a running process.
totalview *filename*	Starts the debugger and loads the program specified by *filename*.
totalview *filename corefile*	Starts the debugger and loads the program specified by *filename* and its core file specified by *corefile*.
totalview *filename* -a *args*	Starts the debugger and passes all subsequent arguments (specified by *args*) to the program specified by *filename*. The -a 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.



CINECA

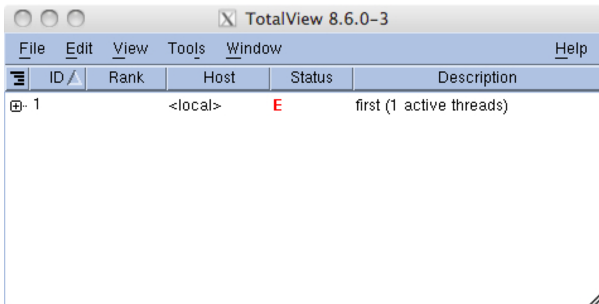
Totalview: Action points

- Breakpoint stops the excution of the process and threads that reach it.
  - Unconditional
  - Conditional: stop only if the condition is satisfied.
  - Evaluation: stop and excute 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

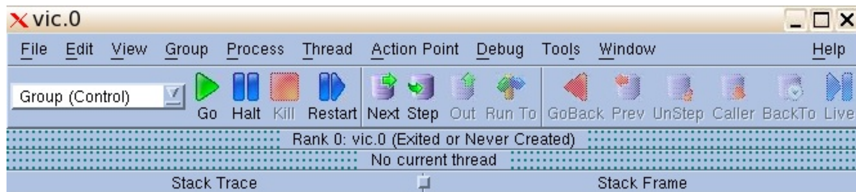
- Breakpoint
  - Right click on a source line → Set breakpoint
  - Click on the line number
- Watchpoint
  - Right click on a variable → Create watchpoint
- Barrier point
  - Right click on a source line → Set barrier
- Edit action point property
  - Rigth click on a action point in the Action Points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

Totalview: Status



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 excution
Halt	Stop excution
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 excuted 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, excution will stop within function.
Out	Excute 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.

SCAI SuperComputing Applications and Innovation

Totalview:Mouse buttons

SAPIENZA UNIVERSITÀ DI ROMA

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.
Rigth	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.

CINECA