Debugging HPC programs, C and Fortran

CINES, Montpellier

Slides by Benoît Leveugle
Talk by Victor Cameo Ponz

- The aim of this training is to be familiar with :
  - identifying recurrent bugs
  - tracing them with the appropriate tool
  - solving them
- Debugging is not magic, it is science. With the appropriate approach, you will solve 99% of code related bugs in no time
- Versions of compiler used :
  - gcc/gfortran : 4.8.2 (from ubuntu 14.04x64)
  - icc/ifort : 14.0.3 (from parallel studio 2013 SP1 update 3)

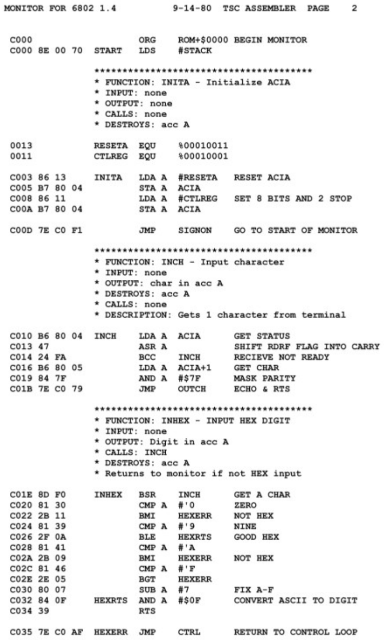
Summary

1. History
2. Unix in 10 minutes
3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB
4. Why debugging
5. Common bugs and method to catch them :
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow
6. Conclusion and useful links

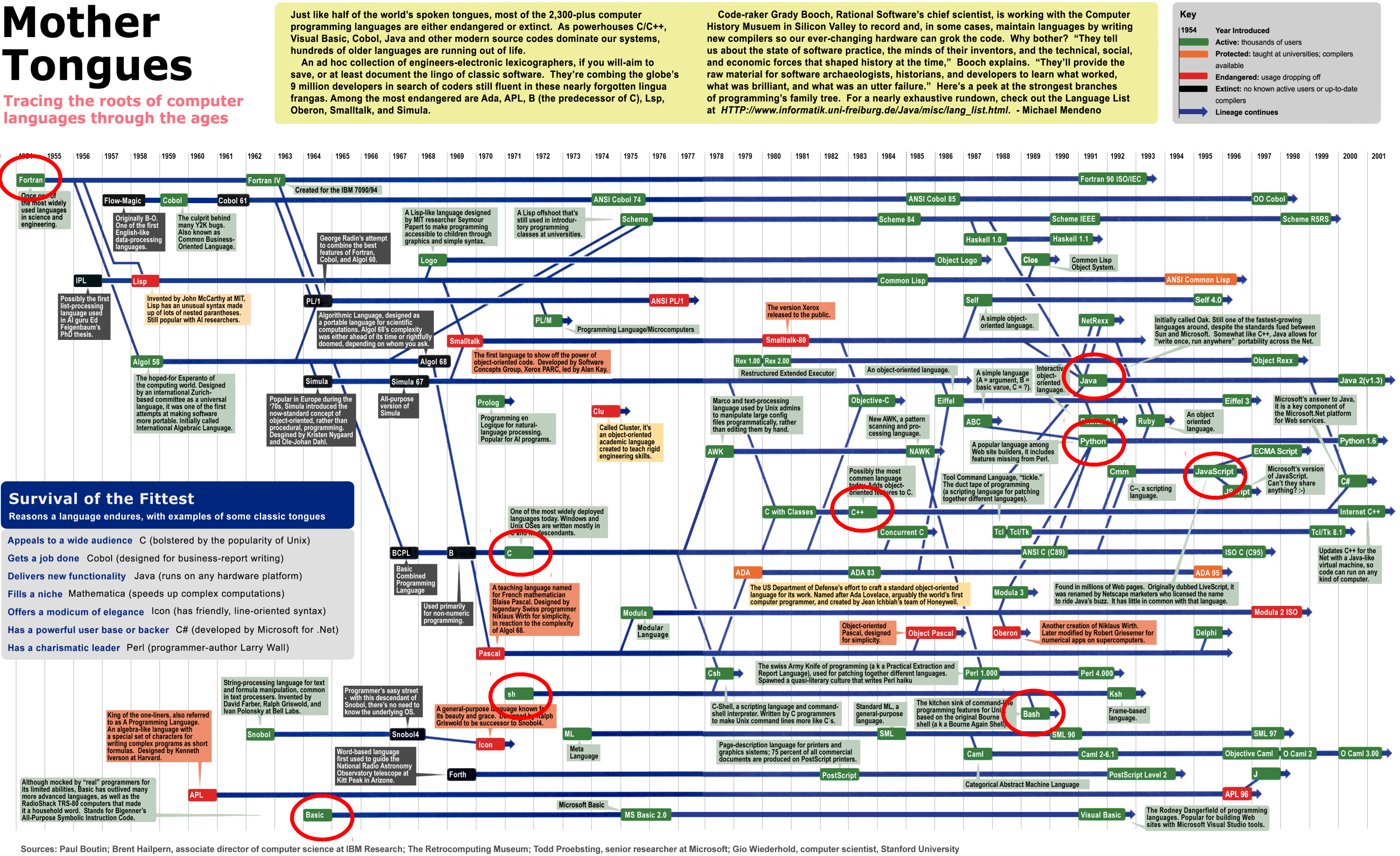
Summary

1. **History**
2. Unix in 10 minutes
3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB
4. Why debugging
5. Common bugs and method to catch them :
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow
6. Conclusion and useful links

- 1842 : Ada Lovelace, First program of history
- 1880s : Herman Hollerith, Data on physical medium
- 1940s : Von Neumann Architecture allow programs to be stored in memory
- 1949 : Assembly language replace machine specific instructions, Text format
- 1947 : Grace Hopper, debugging
- 1949 : Grace Hopper, first compiler (A)
- 1954 : FORTRAN, first high level language
- 1971 : C language replace B
- 1983 : B. Stroustrup, C++
- 1991 : HTML
- 1995 : JAVA



```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

 C000                  ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START  LDS    #STACK

                *************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA EQU    %00010011
 0011           CTLREG EQU    %00010001

 C003 86 13     INITA  LDA A  #RESETA   RESET ACIA
 C005 B7 80 04         STA A  ACIA
 C008 86 11            LDA A  #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04         STA A  ACIA

 C00D 7E C0 F1         JMP    SIGNON    GO TO START OF MONITOR

                *************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH   LDA A  ACIA      GET STATUS
 C013 47               ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA            BCC    INCH      RECIEVE NOT READY
 C016 B6 80 05         LDA A  ACIA+1    GET CHAR
 C019 84 7F            AND A  #$7F      MASK PARITY
 C01B 7E C0 79         JMP    OUTCH     ECHO & RTS

                *************************************
                * FUNCTION: INHEX - INPUT HEX DIGIT
                * INPUT: none
                * OUTPUT: Digit in acc A
                * CALLS: INCH
                * DESTROYS: acc A
                * Returns to monitor if not HEX input

 C01E 8D F0     INHEX  BSR    INCH      GET A CHAR
 C020 81 30            CMP A  #'0       ZERO
 C022 2B 11            BMI    HEXERR    NOT HEX
 C024 81 39            CMP A  #'9       NINE
 C026 2F 0A            BLE    HEXRTS    GOOD HEX
 C028 81 41            CMP A  #'A
 C02A 2B 09            BMI    HEXERR    NOT HEX
 C02C 81 46            CMP A  #'F
 C02E 2E 05            BGT    HEXERR
 C030 80 07            SUB A  #7        FIX A-F
 C032 84 0F     HEXRTS AND A  #$0F      CONVERT ASCII TO DIGIT
 C034 39               RTS

 C035 7E C0 AF  HEXERR JMP    CTRL      RETURN TO CONTROL LOOP
```


Mother Tongues
Tracing the roots of computer languages through the ages
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.
An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.
Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html. - Michael Mendeno
Key
1954 Year Introduced
Active: thousands of users
Protected: taught at universities; compilers available
Endangered: usage dropping off
Extinct: no known active users or up-to-date compilers
Lineage continues
1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
Fortran
Fortran IV
Created for the IBM 7090/94
Fortran 90 ISO/IEC
Once one of the most widely used languages in science and engineering.
Flow-Magic
Cobol
Cobol 61
ANSI Cobol 74
ANSI Cobol 85
OO Cobol
Originally B-O. One of the first English-like data-processing languages.
The culprit behind many Y2K bugs. Also known as Common Business-Oriented Language.
A Lisp-like language designed by MIT researcher Seymour Papert to make programming accessible to children through graphics and simple syntax.
A Lisp offshoot that's still used in introductory programming classes at universities.
Scheme
Scheme 84
Scheme IEEE
Scheme R5RS
Haskell 1.0
Haskell 1.1
George Radin's attempt to combine the best features of Fortran, Cobol, and Algol 60.
Logo
Object Logo
Clos
Common Lisp Object System.
IPL
Lisp
Common Lisp
ANSI Common Lisp
Possibly the first list-processing language used in AI guru Ed Feigenbaum's PhD thesis.
Invented by John McCarthy at MIT, Lisp has an unusual syntax made up of lots of nested parantheses. Still popular with AI researchers.
PL/I
ANSI PL/I
The version Xerox released to the public.
Self
Self 4.0
Algorithmic Language, designed as a portable language for scientific computations. Algol 68's complexity was either ahead of its time or rightfully doomed, depending on whom you ask.
PL/M
Programming Language/Microcomputers
A simple object-oriented language.
NetRexx
Initially called Oak. Still one of the fastest-growing languages around, despite the standards fued between Sun and Microsoft. Somewhat like C++, Java allows for "write once, run anywhere" portability across the Net.
Smalltalk
Smalltalk-80
Algol 58
Algol 68
The first language to show off the power of object-oriented code. Developed by Software Concepts Group, Xerox PARC, led by Alan Kay.
Rex 1.00
Rex 2.00
Object Rexx
Restructured Extended Executor
An object-oriented language.
The hoped-for Esperanto of the computing world. Designed by an international Zurich-based committee as a universal language, it was one of the first attempts at making software more portable. Initially called International Algebraic Language.
Simula
Simula 67
A simple language (A = argument, B = basic value, C = ?).
Interactive object-oriented language.
Java
Java 2(v1.3)
Popular in Europe during the '70s, Simula introduced the now-standard concept of object-oriented, rather than procedural, programming. Designed by Kristen Nygaard and Ole-Johan Dahl.
All-purpose version of Simula
Prolog
Programming en Logique for natural-language processing. Popular for AI programs.
Clu
Called Cluster, it's an object-oriented academic language created to teach rigid engineering skills.
Marco and text-processing language used by Unix admins to manipulate large config files programmatically, rather than editing them by hand.
Objective-C
Eiffel
Eiffel 3
Microsoft's answer to Java, it is a key component of the Microsoft.Net platform for Web services.
New AWK, a pattern scanning and processing language.
ABC
Ruby
An object oriented language.
AWK
NAWK
Python
Python 1.6
A popular language among Web site builders, it includes features missing from Perl.
ECMA Script
Possibly the most common language today. Adds object-oriented features to C.
Tool Command Language, "tickle." The duct tape of programming (a scripting language for patching together different languages).
Cmm
C--, a scripting language.
JavaScript
Microsoft's version of JavaScript. Can't they share anything? :-)
C#
C with Classes
C++
Internet C++
Concurrent C
Tcl
Tcl/Tk
Tcl/Tk 8.1
One of the most widely deployed languages today. Windows and Unix OSes are written mostly in C and its descendants.
Survival of the Fittest
Reasons a language endures, with examples of some classic tongues
Appeals to a wide audience C (bolstered by the popularity of Unix)
Gets a job done Cobol (designed for business-report writing)
Delivers new functionality Java (runs on any hardware platform)
Fills a niche Mathematica (speeds up complex computations)
Offers a modicum of elegance Icon (has friendly, line-oriented syntax)
Has a powerful user base or backer C# (developed by Microsoft for .Net)
Has a charismatic leader Perl (programmer-author Larry Wall)
BCPL
B
C
ANSI C (C89)
ISO C (C95)
Updates C++ for the Net with a Java-like virtual machine, so code can run on any kind of computer.
Basic Combined Programming Language
Used primarily for non-numeric programming.
ADA
ADA 83
ADA 95
A teaching language named for French mathematician Blaise Pascal. Designed by legendary Swiss programmer Niklaus Wirth for simplicity, in reaction to the complexity of Algol 68.
The US Department of Defense's effort to craft a standard object-oriented language for its work. Named after Ada Lovelace, arguably the world's first computer programmer, and created by Jean Ichbiah's team of Honeywell.
Modula 3
Found in millions of Web pages. Originally dubbed LiveScript, it was renamed by Netscape marketers who licensed the name to ride Java's buzz. It has little in common with that language.
Modula
Modular Language
Modula 2 ISO
Object-oriented Pascal, designed for simplicity.
Object Pascal
Oberon
Another creation of Niklaus Wirth. Later modified by Robert Griesemer for numerical apps on supercomputers.
Delphi
Pascal
Csh
The swiss Army Knife of programming (a k a Practical Extraction and Report Language), used for patching together different languages. Spawned a quasi-literary culture that writes Perl haiku
Perl 1.000
Perl 4.000
String-processing language for text and formula manipulation, common in text processers. Invented by David Farber, Ralph Griswold, and Ivan Polonsky at Bell Labs.
Programmer's easy street - with this descendant of Snobol, there's no need to know the underlying OS.
sh
Ksh
Frame-based language.
King of the one-liners, also referred to as A Programming Language. An algebra-like language with a special set of characters for writing complex programs as short formulas. Designed by Kenneth Iverson at Harvard.
C-Shell, a scripting language and command-shell interpreter. Written by C programmers to make Unix command lines more like C's.
Standard ML, a general-purpose language.
Bash
Snobol
Snobol4
ML
SML
SML 90
SML 97
Icon
Meta Language
Page-description language for printers and graphics sistems; 75 percent of all commercial documents are produced on PostScript printers.
Caml
Caml 2-6.1
Objective Caml
O Caml 2
O Caml 3.00
Word-based language first used to guide the National Radio Astronomy Observatory telescope at Kitt Peak in Arizone.
Forth
PostScript
PostScript Level 2
Categorical Abstract Machine Language
J
Although mocked by "real" programmers for its limited abilities, Basic has outlived many more advanced languages, as well as the RadioShack TRS-80 computers that made it a household word. Stands for Bigenner's All-Purpose Symbolic Instruction Code.
APL
APL 96
Basic
Microsoft Basic
MS Basic 2.0
Visual Basic
The Rodney Dangerfield of programming languages. Popular for building Web sites with Microsoft Visual Studio tools.
Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

- Learn (or remember) unix basic commands.
  - Unix in 10 minutes : http://freeengineer.org/learnUNIXin10minutes.html

Summary

1. *History*
2. *Unix in 10 minutes*
3. **Tools used**
    1. Preprocessing
    2. Valgrind
    3. GDB
4. Why debugging
5. Common bugs and method to catch them :
    1. Floating point exceptions (Invalid, Overflow, Zero)
    2. Uninitialized values reading
    3. Allocation/deallocation issues
    4. Array out of bound reading/writing
    5. IO issues
    6. Memory leak
    7. Stack overflow
    8. Buffer overflow
6. Conclusion and useful links

- Your allies in the battle:
  - Preprocessing
  - Valgrind
  - GDB
  - Intel Inspector (not seen in this training : https://software.intel.com/en-us/articles/intel-inspector-xe-2011-documentation)

- More tools can be found on the web

- Few words about Preprocessing
  - Tool used in many languages, here in C / Fortran
  - Allow to compile only wanted part of code
  - Useful to debug (MPI for example)

```
#ifdef MYVALUE
#ifndef MYVALUE
#else
#endif
```

- We will use preprocessing to simulate bugs one by one

- Few words about Preprocessing
  - Tool used in many languages, here in C / Fortran
  - Allow to compile only wanted part of code
  - Useful to debug (MPI for example)

```
#ifdef MYVALUE
#ifndef MYVALUE
#else
#endif
```

- We will use preprocessing to simulate bugs one by one

- Few words about Preprocessing

```
program helloorhey

implicit none

#ifdef HELLO
   print *, "Hello world ! «
#endif

#ifdef HEY
   print *,‘Hey !’
#endif

end program helloorhey
```

```
$ gfortran myfile.f90
Warning: myfile.f90:5: Illegal preprocessor directive
Warning: myfile.f90:7: Illegal preprocessor directive
Warning: myfile.f90:9: Illegal preprocessor directive
Warning: myfile.f90:11: Illegal preprocessor directive
$ ./a.out
 Hello world !
 Hey !
$ gfortran -cpp myfile.f90
$ ./a.out
$ gfortran -cpp -DHELLO myfile.f90
$ ./a.out
 Hello world !
$ gfortran -cpp -DHEY myfile.f90
$ ./a.out
 Hey !
$
```

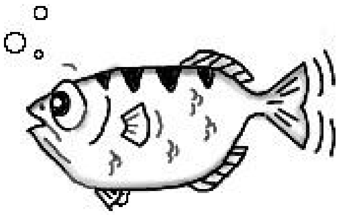
3.2. Valgrind

- Valgrind is a powerful memory checking tool. It is able to catch use of uninitialized values, out of bound access, stack overflow, etc. However, it will not see fpe and some other bugs.
- Valgrind has a tool to **check memory**, a tool to **check memory leak**, a tool to **profile the code** (use with KCacheGrind), etc.
- Valgrind is able to watch only a part of the code in order to avoid other warnings or slowdowns.
- If compiled manually, Valgrind is able to **debug MPI communications**.
- WARNING : Valgrind will displays errors if intel environment is not loaded when debugging an intel compiled program

- How to use valgrind ? (very verbose)

```
$ gfortran -g -fbacktrace myfile.f90 -o myprog.exe
```

```
$ valgrind ./myprog.exe
==3306== Memcheck, a memory error detector
==3306== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3306== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==3306== Command: ./a.out
==3306==
==3306== Invalid read of size 8
==3306==    at 0x40060F: main (deb_c.c:191)
==3306==  Address 0x51fd090 is 0 bytes after a block of size 80 alloc'd
==3306==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3306==    by 0x4005CE: main (deb_c.c:185)
==3306==
 10.000000 0.000000
==3306==
==3306== HEAP SUMMARY:
==3306==     in use at exit: 0 bytes in 0 blocks
==3306==   total heap usage: 1 allocs, 1 frees, 80 bytes allocated
==3306==
==3306== All heap blocks were freed -- no leaks are possible
==3306==
==3306== For counts of detected and suppressed errors, rerun with: -v
==3306== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
$
```



GDB

- GDB is the gnu debugger, available with gcc/gfortran
- GDB is able to **execute program step by step watching desired variables**, **break when a condition is true or at a specific line** then display code for this area, etc.
- GDB is able to **modify a variable on the fly**
- GDB is able to **backtrace an error** to provide more information on it
- GDB overhead is lower than valgrind's overhead

- How to use gdb? (basic commands, more at http://en.wikibooks.org/wiki/GCC_Debugging/gdb)

```
$ gfortran -g -fbacktrace myfile.f90 -o myprog.exe
```

```
$ gdb myprog.exe
```

- "run" run the program
- "break" set a "breakpoint" at a certain area \ function
- "next" execute next line of code (after a break)
- "continue" go to next breakpoint or end of program
- "print" print out a variables \ expressions contents
- "disp" print out a variable \ expression value every step
- "cond" conditional
- "set" change a value
- "quit" exit gdb
- "backtrace" get informations on program state at exit

```
(gdb) set variable x=12
(gdb) break test.cpp:2
Breakpoint 1 at 0x1234: file test.cpp, line 2.
(gdb) cond 1 i==2147483648
(gdb) run
```

Summary

1. *History*
2. *Unix in 10 minutes*
3. *Tools used*
   1. *Preprocessing*
   2. *Valgrind*
   3. *GDB*
4. **Why debugging**
5. Common bugs and method to catch them :
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow
6. Conclusion and useful links

4. Why debugging

- When should you think there is a bug ?
  - Program returns an error message
  - Program returns an error exit code (other than 0)
  - Program finishes with NaN or +Inf values
  - Program ends unexpectedly
  - Other cases, many scenario are possible

ow to get the exit code of a program ?

- **$?** gives you the exit code of the last executed command.
- Other than 0 means something went wrong, and this code may help you understand why.

```
$ gfortran myfile.f90
$ echo $?
0
$ ./a.out
 Hello world !
$ echo $?
0
$
```

```
$ gfortran myfile.f90
myfile.f90:3:

mplicit none
1
Error: Unclassifiable statement at (1)
$ echo $?
1
$
```

```
$ ./a.out

Program received signal SIGSEGV:
Segmentation fault - invalid memory reference.

Backtrace for this error:
#0  0x7FFC993C87D7
#1  0x7FFC993C8DDE
#2  0x7FFC9901FC2F
Segmentation fault (core dumped)
$ echo $?
139
$
```

Summary

1. *History*
2. *Unix in 10 minutes*
3. *Tools used*
   1. *Preprocessing*
   2. *Valgrind*
   3. *GDB*
4. *Why debugging*
5. **Common bugs and method to catch them :**
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow
6. Conclusion and useful links

- Common bugs :
  - Floating point exceptions (Invalid, Overflow, Zero)
  - Uninitialized values reading
  - Allocation/deallocation issues
  - Array out of bound reading/writing
  - IO issues
  - Memory leak
  - Stack overflow
  - Buffer overflow

  - Algorithm/mathematical bugs (the worsts, especially with iterative methods). This last one will not generate an error, but results will be wrong. No specific methods, be smart.

- Floating point exceptions
  - Zero
    - When you divide by zero, very common in HPC
    - $\frac{A}{0.0} = +\infty$
  - Invalid
    - When the operation is mathematicaly impossible
    - $\text{acos}(10.0) = NaN$
  - Overflow/Underflow
    - When you reach maximum/minimum number that system can hold
    - $\exp(10E15) = A\ huge\ number$
- FPEs will not generate errors at runtime !

Compiler	Fortran
gfortran	**-g -fbacktrace-ffpe-trap=zero,underflow,overflow,invalid** will catch fpe at runtime
ifort	**-g -traceback -fpe0** will catch fpe at runtime

Compiler	C
gcc icc	Add #include <fenv.h> and start with feenableexcept(FE_DIVBYZERO| FE_INVALID|FE_OVERFLOW); Or use: if (fetestexcept(FE_OVERFLOW | ...)) puts ("FE_OVERFLOW is set");

```
#include <fenv.h>

int main(int argc, char **argv)
{
feenableexcept(FE_DIVBYZERO| FE_INVALID|FE_OVERFLOW);
...
}
```

- Uninitialized values reading
  - When you try to read a non initialized value
  - The program may not stop, and all following calculations will be based on a random value
  - Common with MPI programs (Ghost, etc)

- Static variable : variable uninitialized is static
  - no error at runtime
- Dynamic variable : variable uninitialized is dynamic
  - no error at runtime
- Not allocated variable : try to use a non allocated dynamic variable
  - error : segmentation fault

Compiler	Fortran
gfortran	When needed to use a debugging tool, do not forget **-g -fbacktrace** to get information on bug position in code ▪ static variable : **-Wuninitialized -O -g -fbacktrace** will display a warning **Valgrind** : "Conditional jump or move depends on uninitialised value(s)" ▪ dynamic variable : **Valgrind** : "Conditional jump or move depends on uninitialised value(s)" ▪ not allocated variable : **-g -fbacktrace** will catch it (size 0 or huge random number)
ifort	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code ▪ static variable : **-check all** (or -check uninit) catch it, -ftrapuv may help ▪ dynamic variable : **Valgrind** : "Conditional jump or move depends on uninitialised value(s)" ▪ not allocated variable : **-g -traceback** will catch it (size 0 or huge random number)

Compiler	C
gcc	When needed to use a debugging tool, do not forget **-g** to get information on bug position in code ▪ <u>static variable :</u> **-Wuninitialized** or **-wall** will display a warning **Valgrind** : “Conditional jump or move depends on uninitialised value(s)” ▪ <u>dynamic variable :</u> **Valgrind** : “Conditional jump or move depends on uninitialised value(s)” ▪ <u>not allocated variable :</u> **-Wuninitialized** or **-wall** will display a warning **Valgrind** : “Conditional jump or move depends on uninitialised value(s)” **gdb** : with backtrace
icc	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code ▪ <u>static variable :</u> **-Wuninitialized** will display a warning, **-g -check=uninit** will catch it at runtime ▪ <u>dynamic variable :</u> **Valgrind** : “Conditional jump or move depends on uninitialised value(s)” ▪ <u>not allocated variable :</u> **-Wuninitialized** will display a warning, **-g -check=uninit** will catch it at runtime

- Allocation issues

- Try do free an non allocated variable
  - Will generate an error at runtime (not with gcc)
- Try do allocate an already allocated variable
  - Will generate an error at runtime (not in C)
- Not freed memory
  - No errors

Compiler	**Fortran**
gfortran	When needed to use a debugging tool, do not forget **-g -fbacktrace** to get information on bug position in code ▪ free an non allocated variable: **-g -fbacktrace** will catch it at runtime ▪ allocate an already allocated variable: **-g -fbacktrace** will catch it at runtime ▪ Not freed memory: **Valgrind** will catch it with **--leak-check=full**
ifort	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code ▪ free an non allocated variable: **-g -traceback** will catch it at runtime ▪ allocate an already allocated variable: **-g -traceback** will catch it at runtime ▪ Not freed memory: **Valgrind** will catch it with **--leak-check=full**

Compiler	C
gcc	When needed to use a debugging tool, do not forget **-g** to get information on bug position in code ▪ free an non allocated variable: **-Wuninitialized** or **-wall** will display a warning **Valgrind** : “Conditional jump or move depends on uninitialised value(s)” ▪ allocate an already allocated variable: **Valgrind** will catch it with **--leak-check=full** ▪ Not freed memory: **Valgrind** will catch it with **--leak-check=full**
icc	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code ▪ free an non allocated variable: **-Wuninitialized** will display a warning, **-g -check=uninit** will catch it at runtime ▪ allocate an already allocated variable: **Valgrind** will catch it with **--leak-check=full** ▪ Not freed memory: **Valgrind** will catch it with **--leak-check=full**

5.4. Common bugs - Array out of bounds

- Array out of bound reading/writing
  - Will not generate errors most of the time
  - Very common in HPC

- Often called "Gardening" when memory is not protected

Compiler	**Fortran**
gfortran	**-g -fbacktrace -fbounds-check** will catch it at runtime
ifort	**-g -traceback -check all** (or -check bounds) will catch it at runtime

Compiler	**C**
gcc	When needed to use a debugging tool, do not forget **-g** to get information on bug position in code **Valgrind** : “Invalid read/write of size 8” Or patch gcc and recompile it with bounds checking (http://sourceforge.net/projects/boundschecking/)
icc	**-g -traceback -check-pointers=rw** will catch it at runtime, however -check-pointers=rw makes all other debugging options not working, be careful

5.5. Common bugs - IO

- IO issues
- Errors are often very explicit. No need to use a debugging tool. However, Valgrind and fpe options can detect some related errors (bad reading = bad initialized value or = fpe, etc.)
- Do not forget to put -g -fbacktrace (gcc/gfortran) or -g -traceback (icc/ifort) to get useful error information.

- Memory leak
- Can be the reason of a segmentation fault (signal 11) or an unexpected code halt. Memory growth and growth until it reach limit which halts the program.
- Impossible with recent Fortran compilers if not using "pointers".
- If using Fortran pointers or C, then Valgrind will catch it !

Compiler	Fortran
gfortran	When needed to use a debugging tool, do not forget **-g -fbacktrace** to get information on bug position in code **Valgrind** will catch it with **--leak-check=full**
ifort	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code **Valgrind** will catch it with **--leak-check=full**

Compiler	C
gcc	When needed to use a debugging tool, do not forget **-g** to get information on bug position in code **Valgrind** will catch it with **--leak-check=full**
icc	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code **Valgrind** will catch it with **--leak-check=full**

- Stack Overflow

- Extremely common with bad written programs
- More common with gcc/gfortran programs (icc/ifort are often smarter with memory)
- More common with multithreaded programs, like OpenMP programs
  - Each son has a very small stack which is rapidly ful

- Will result in a segmentation fault

Compiler	Fortran
gfortran	When needed to use a debugging tool, do not forget **-g -fbacktrace** to get information on bug position in code **Valgrind** will catch it **gdb** will catch it with backtrace but not a lot informations
ifort	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code **Valgrind** will catch it **gdb** will catch it with backtrace but not a lot informations

Compiler	C
gcc	When needed to use a debugging tool, do not forget **-g** to get information on bug position in code **Valgrind** will catch it **gdb** will catch it with backtrace but not a lot informations
icc	When needed to use a debugging tool, do not forget **-g -traceback** to get information on bug position in code **Valgrind** will catch it **gdb** will catch it with backtrace but not a lot informations

- Buffer Overflow

- Famous for security reasons

- More common in C than in Fortran
- Will generate an error, except with icc

- Can ask gcc to ignore it using : -fno-stack-protector

Compiler	Fortran
gfortran	Error is self explaining
ifort	Error is self explaining

Compiler	C
gcc	When needed to use a debugging tool, do not forget **-g** to get information on bug position in code **gdb** will catch it with backtrace
icc	**-g -traceback -check-pointers=rw** will catch it at runtime, however -check-pointers=rw makes all other debugging options not working, be careful

1. *History*
2. *Unix in 10 minutes*
3. *Tools used*
   1. *Preprocessing*
   2. *Valgrind*
   3. *GDB*
4. *Why debugging*
5. *Common bugs and method to catch them :*
   1. *Floating point exceptions (Invalid, Overflow, Zero)*
   2. *Uninitialized values reading*
   3. *Allocation/deallocation issues*
   4. *Array out of bound reading/writing*
   5. *IO issues*
   6. *Memory leak*
   7. *Stack overflow*
   8. *Buffer overflow*
6. **Conclusion and useful links**

6. Conclusion and useful links

Veteran advices :

- NaN is not equal to itself generally (depends on platform and compiler)
- Programs may not give the same results depending of the optimizations options. Using multi threading/MPI also provides different results for each run.
- Some optimization options may alter precision
- Remember that terminal ouput may not refresh instantly: using "hello 1", "hello 2", etc may result in wrong location, use flush (may slow down the program)

gcc/icc	fflush(stdout);
gfortran	call flush()
ifort	call flush(ierror) *(segfault if no ierror)*

6. Conclusion and useful links

Veteran advices :

- If your bug is impossible to locate (mathematical or algorithm error), ask someone else to check, most of the time bug is right in front of you but your knowledge of the code prevent you from seeing it
- **Never debug more than half a day, this could be worst (introduce more bugs to resolve one), and your brain needs to "think" away from a screen**
- Automatic testing can prevent debuging

Thank you for your attention