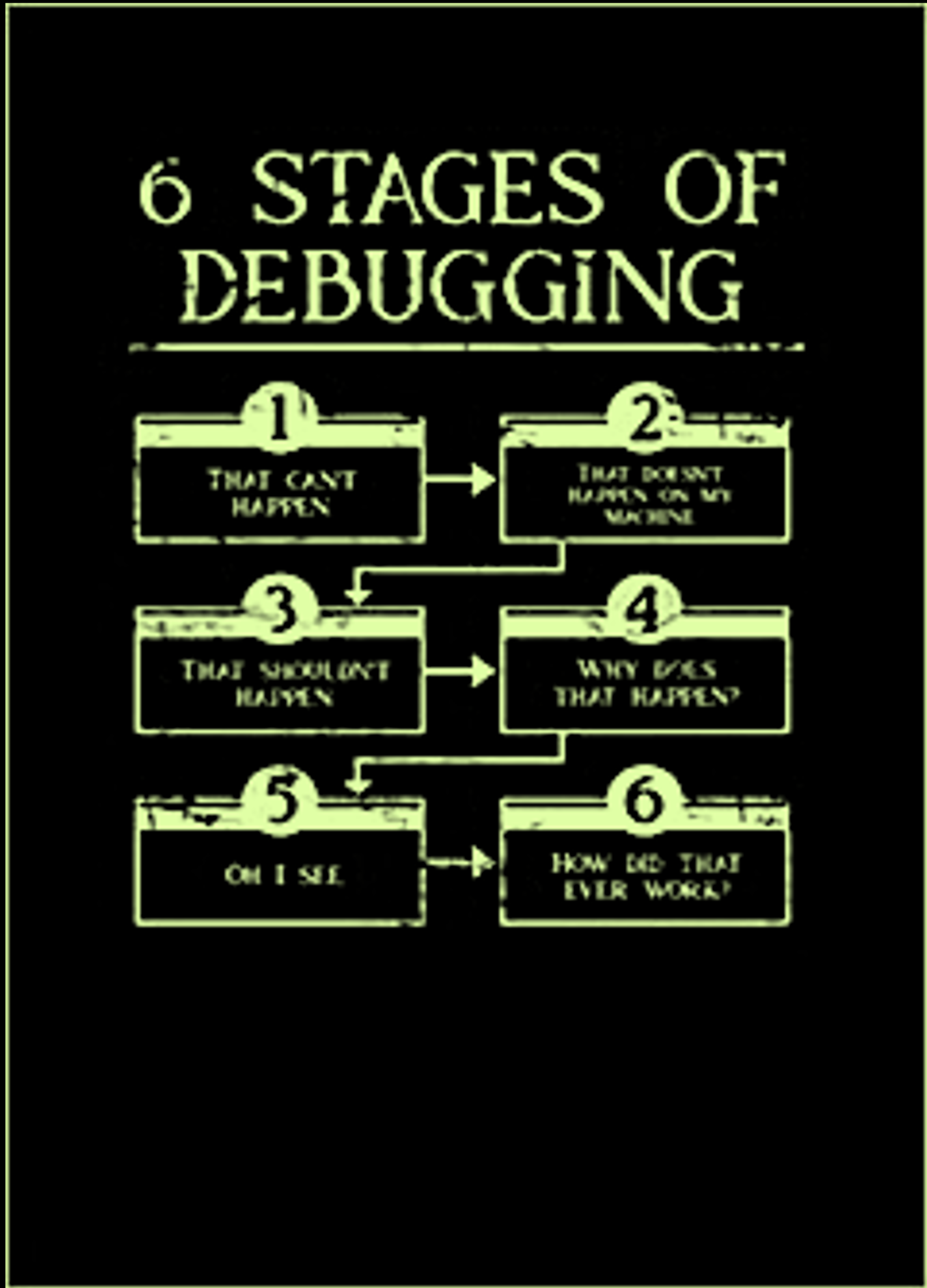
# Debugging

From Theoretical/Concept Considerations to Weapons for Efficient Coding

**Carlos J. Barrios H. PhD.**
cbarrios@uis.edu.co


6 STAGES OF DEBUGGING
1
THAT CAN'T HAPPEN
2
THAT DOESN'T HAPPEN ON MY MACHINE
3
THAT SHOULDN'T HAPPEN
4
WHY DOES THAT HAPPEN?
5
OH I SEE
6
HOW DID THAT EVER WORK?

## Sources of This Presentation:

Guest lecture DR. Robert Oates

Icos Research Group

University of Nottingham

(http://www.cs.nott.ac.uk/~pszjg1/FSE12/FSE_9.pdf)

SC_CAMP Debbuing and Prolifing Lectures DR. Xavier Besseron

University of Luxembourg

(https://www.sc-camp.org)

# Why Debug Software?

- It’s an important work skill
- You ‘re becoming professionals!
- It’s an important development skill
- It’s an important life skill
- You’ re not perfect (and perfect is not good)

# General Purpose Debugging



1. Understanding
2. Reproduction and Data Gathering
3. Hypothesis
4. Experiment Design
5. Test
6. **Implementation**
   - **Design**
   - **Implement**
   - **Test**

# General Purpose Debugging

## Understanding the System

- Look at the system specifications if available
  - Inputs
  - Outputs

- How do the components of the system fit together?

- Information Flow
  - Output backwards
  - Input forwards

# General Purpose Debugging

## Reproduction and Data Gathering

- A hugely important part of debugging
  - Simple if bug is persistent
  - Hard if the bug is transient
  - Opportunity to study the system when the bug is occurring

# General Purpose Debugging

- **Questions to ask**
  - What state was the system in?
    - Operating system
    - Program Settings
  - When does the bug happen?
    - When the system was first turned on
    - After years of use
    - When OK is clicked

- The program settings in particular will rule out huge swathes of code!

**Remember: A Machine makes what do you want and order (or the architect orders!)**

# General Purpose Debugging

## Hypothesis

- A hypothesis is an explanation which explains the observed behaviour of the bug IN CONTEXT
  - i.e. Knowing how the system works!

- Sometimes a hypothesis will only explain some of the behaviour
  - There may be multiple bugs manifesting at the same time!

# General Purpose Debugging

## Experiment Design

- Design an experiment which will falsify your hypothesis
- If you can falsify it then it is an incorrect hypothesis
- If you can’t then it **MIGHT** be the source of the problem

# General Purpose Debugging

## Test

- If your test fails to falsify your hypothesis
  - Move on to step 6, implementation

- If your test falsifies your hypothesis
  - Move to step 3 (hypothesis)

  OR

  - Move to step 1 (system understanding)

- Regardless, this test has given you more information about the nature of the bug

**Remember: It is important design your workflow/roadmap of your system/implementation**

# General Purpose Debugging

## Implementation

- Sometimes this makes up part of the Hypothesis and Test phases
  - “If I ‘fix’ it and the problem goes away, then the hypothesis isn’t falsified”

- Undefined scenarios
  - If the bug happens because a customer is using the software in a way you hadn’t predicted
    - Requires a whole new design cycle

# Software Debugging

- Traces
  - Screen
  - Log

- Breakpoints
  - Code execution
  - Watches
  - Memory inspection

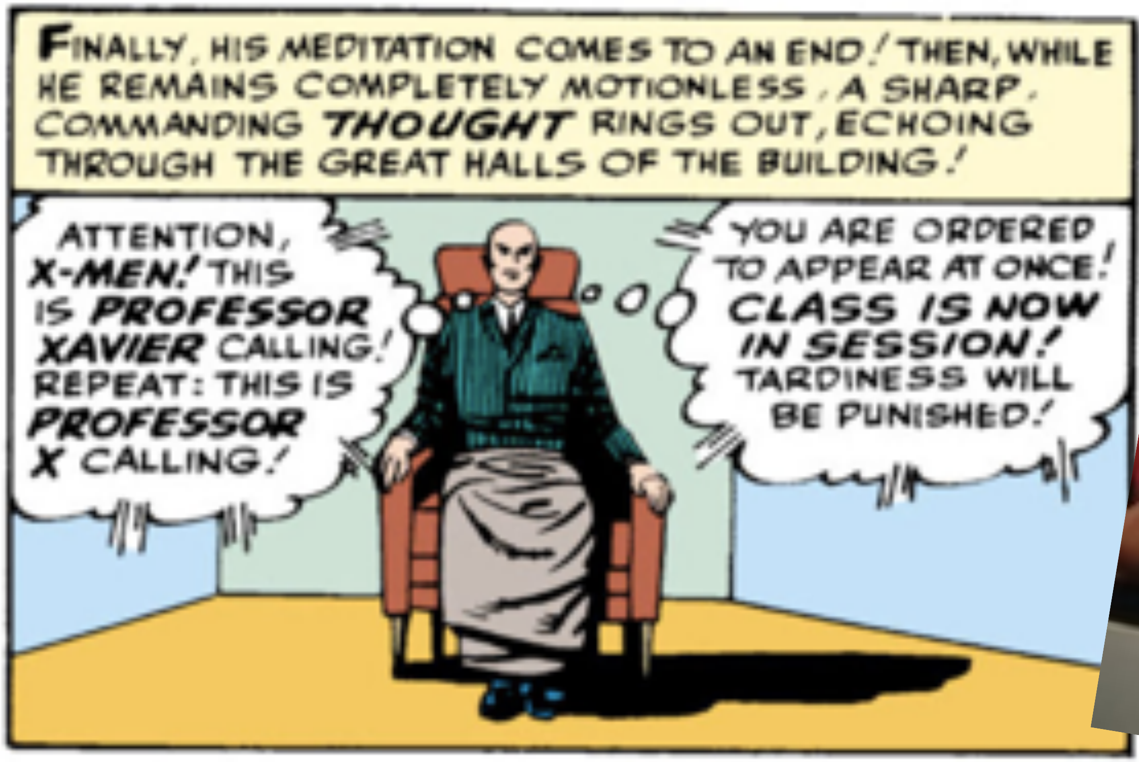
# Software Debugging

## Traces

- Traces are streams of data taken from the system
  - On-screen
  - Log File

- Uses
  - Embedded systems with no direct connection
  - Any system producing large volumes of data

# Now, Debugging in Practice




www.sc-camp.org

# Know Your Bugs:
# Weapons for Efficient Debugging

Xavier Besseron

SuperComputing Summer Camp

# Why debugging?

## Bugs are in every programs

- Industry Average:
  "*about 15 - 50 errors per 1000 lines of delivered code*" [1]

## Bugs in High Performance Computing

- Even more difficult due to concurrency
- Can crash super-computers
- Can waste large amount of CPU-time

## Famous bugs and consequences

- Ariane 5 rocket destoryed in 1996: 1 billion US $
- Power blackout in US in 2003: 45 million people affected
- Medtronic heart device vulnerable to remote attack in 2008
- ...

[1] *Code Complete* by Steve McConnell

# Outline

2 Tools for Debugging
- Compilers
- GNU Debugger
- Valgrind

# Tools for debugging

## Compilers

- It's the first program to check your code
- GCC,Intel Compiler,CLang, MS Compiler, ...

## Static code analyzers

- Check the program without executing it
- Splint, Cppcheck, Coccinelle, ...

## Debuggers

- Inspect/modify a program during its execution
- GDB: the GNU Project Debuggerfor serial and multi-thread programs
- Parallel debuggers (commercial): RogueWave Totalview, Allinea DDT

## Dynamics code analyzers and profilers

- Check the program while executing it
- Valgrind, Gcov, Gprof, ...
- Commercial software: Purify, Intel Parallel Inspector, ..

# Compilers 1/2

## What does a compiler do?

- Translate source code to machine code
- 3 phases:
  - Lexical analysis: recognize "words" or tokens
  - Syntax analysis: build syntax tree according to language grammar
  - Semantic analysis: check rules of the language, variable declaration, types, etc.
- With this knowledge, a compiler can find many bugs

→ Pay attention to compiler warnings and errors of a program

A compiler can find out if your program makes sense according to the language. However, it cannot guess what you are trying to do.

# Compilers 2/2

## How to use the compiler

- Choose your compiler

| | GCC | CLang | Intel Compiler |
|---|---|---|---|
| C | `gcc` | `clang` | `icc` |
| C++ | `g++` | `clang++` | `icpc` |
| Fortran | `gfortran` | | `ifort` |

- Activate warning messages with the `-Wall` parameters
- Warnings can be enabled/disabled individually, *cf* documentation
- Compile with debug symbols with `-g` parameters

## Example

```
$ gcc -g -Wall program.c -o program
program.c: In function 'main':
program.c:4:15:error: 'y' undeclared (first use in this function)
   int z = x + y;
               ^
program.c:4:15: note: each undeclared identifier is reported only once for
each  program.c:4:7:warning: unused variable 'z' [-Wunused-variable]
   int z = x + y;
       ^
```

# GNU Debugger 1/2

## GDB is the GNU Debugger

- Allow to execute a program step by step
- Watch the value of variables
- Stop the execution on given condition
- Show the backtrace of an error
- Modify value of variables at runtime

## Starting GDB

- Compile your program with the `-g` option
- Start program execution with GDB

  ```
  gdb --args myprogram arg1 arg2
  ```

- Or open a core file (generated after a crash)

  ```
  gdb myprogram corefile
  ```

# GNU Debugger 2/2

## Using GDB

- Command line tool
- Many graphical frontends available too: DDD, Qt Creator, ...
- Online documentation & tutorial:

http://sourceware.org/gdb/current/onlinedocs/gdb/
http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.html

## Main commands

- `help <command>`: get help about a command
- `run`: start execution
- `continue`: resume execute
- `next`: execute the next line
- `break`: set a breakpoint at a given line or function
- `bracktrace`: show the backtrace
- `print`: print the value of a variable
- `quit`: quit GDB

# Valgrind 1/2

## Valgrind is a dynamic analysis tool

- Execute your program with dynamic checking tool: Memcheck, Callgrind, Helgrind, etc.

## Memcheck: memory error detector

- Enable with `-tool=memcheck` (by default)
- Check for memory-related errors:
  unitialized values, out of bound access, stack overflow, memory leak, etc.
- For memory leaks, add option `-leak-check=full`
- http://valgrind.org/docs/manual/mc-manual.html

## Callgrind: performance profiler

- Enable with `-tool=callgrind`
- Check the time you spend in each function of your code
- Visualize results with KCachegrind
- http://valgrind.org/docs/manual/cl-manual.html

# Valgrind 2/2

## Example

```
$ valgrind --tool=memcheck --leak-check=full --track-origins=yes
./program  [...]
==12534==Conditional jump or move depends on uninitialised value(s)
==12534==    at 0x40055E: main (program.c:11)
==12534==Uninitialised value was created by a stack allocation
==12534==    at 0x400536: main (program.c:5)
==12534==
==12534==Invalid write of size 8
==12534==    at 0x4005CE: main (program.c:19)
==12534==Address 0x5203f80 is 0 bytes after a block of size 8,000 alloc'd
==12534==    at 0x4C2BBA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64
==12534==    by 0x400555: main (program.c:9)
==12534==
==12534==
==12534== HEAP SUMMARY:
==12534==     in use at exit: 8,000 bytes in 1 blocks
==12534==   total heap usage: 1 allocs, 0 frees, 8,000 bytes allocated
==12534==
==12534==8,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12534==    at 0x4C2BBA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64
==12534==    by 0x400555: main (program.c:9)
[...]
```

# Outline



# Logic and syntax bugs

## Due to careless programming

- Infinite loop / recursion
- Confusing syntax error,
  *eg* use of $=$ (affectation) instead of $==$ (equality)
- Hard to detect, because everything is right in your mind

## What to do?

- Compile with warnings enabled
- Get some rest and/or an external advice

# Integer overflow 1/2

## Integer variables have limited size

| | Size | Minimum | Maximum |
|---|---|---|---|
| `signed short` | 16 bits | $-2^{15}$ | $2^{15} - 1$ |
| `unsigned short` | 16 bits | 0 | $2^{16} - 1$ |
| `signed int` | 32 bits | $-2^{31}$ | $2^{31} - 1$ |
| `unsigned int` | 32 bits | 0 | $2^{32} - 1$ |
| `signed long long int` | 64 bits | $-2^{63}$ | $2^{63} - 1$ |
| `unsigned long long int` | 64 bits | 0 | $2^{64} - 1$ |

If the result of an operation cannot fit in the variable, most-significant bits are discarded

$\Rightarrow$ we have an Integer Overflow

# Integer overflow 2/2

## Overflow example

```
unsignedcharA = 200;

unsignedcharB =  60;

//Overflow!
unsignedcharS = A + B;
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 200 |
| + | | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | + 60 |
| = | ~~1~~ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | = 4 |

→ No error at runtime!

## What to do?

- Use the right integer type for your data
- In C/C++/Fortran, overflow needs to be checked manually
- CLang and GCC 5.X offer builtin functions to check for overflow
  `__builtin_add_overflow`, `__builtin_sub_overflow`, `__builtin_mul_overflow`, ...

# Floating-Point Number bugs 1/2

## Floating-Point Exceptions (FPE)

- Division by zero:

$$\frac{X}{0.0} = \infty$$

- Invalid operation:

$$\sqrt{-1.0} = NaN\ (Not\ A\ Number)$$

- Overflow / Underflow:

$$e^{1e30} = \infty \qquad e^{-1e30} = 0.0$$

## Loss of precision

- The order of the operations matters:

$$(10^{60} + 1.0) - 10^{60} = 0.0$$

$$(10^{60} - 10^{60}) + 1.0 = 1.0$$

# Floating-Point Number bugs 2/2

## Floating-Point Exceptions and Errors

- No error at runtime by default
- Errors can propagate through all the computation

## What to do?

- Enable errors at runtime in C/C++

```
#define _GNU_SOURCE
#include <fenv.h>

int main()
{
  feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW);
  ...
```

- Read "*What Every Computer Scientist Should Know About Floating-Point Arithmetic*" by David Goldberg

# Memory allocation/deallocation

## Dynamic memory management in C

- `void *p = malloc(size)` allocates memory
- `free(p)` de-allocates the corresponding memory
- In C++, equivalents are `new` and `delete` operations

## Common mistakes

- Failed memory allocation
- Free non-allocated memory
- Free memory twice (double free error)

These mistakes might not trigger an error immediately
Later on, they can cause crashes and undefined behavior

## What to do?

- Check return code (cf documentation)
- Use Valgrind with `-leak-check=full` to catch it

# Memory leaks

## Memory is allocated but never freed

- Allocated memory keeps growing until it fills the computer memory
- Can causes a crash of the program or of the full computer
- Very common is C program, almost impossible in Fortran, Java

## What to do?

- For each `malloc()`, there should be a corresponding `free()`
- Use Valgrind with `-leak-check=full` to catch it

# Using undefined values

## Undefined values

- Uninitialized variable
- Not allocated or already freed memory

## Can cause undefined/unpredictable behavior

- Difficult to track
- Error might not occur immediately
- It can compute incorrect result

## What to do?

- Compile with `-Wuninitialized` or `-Wall`
- Use Valgrind, it should show error `Conditional jump or move depends on uninitialised value(s)`

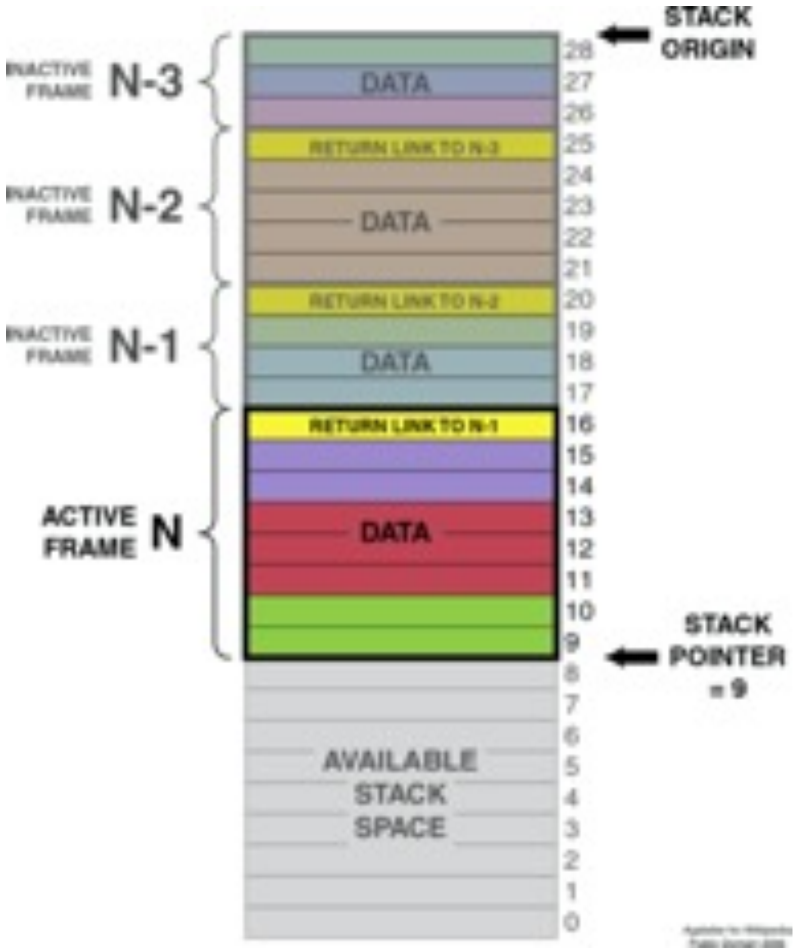
# Stack overflow

## Program stack

- Each function call create a new frame
- Function parameters and local variables are allocated in the frame

## Stack overflow

- Too many function calls usually not-ending recursive calls
- Oversized local data



# Buffer overflow

## Buffer overflow

- Write data in a buffer with an insufficient size
- Overwrite other data (variable, function return address)
- Can be a major security issue
- Can make the stack trace unreadable

## What to do?

- Use functions that check the buffer size: `strcpy()` → `strncpy()`, `sprintf()` → `snprintf()`, etc.
- GCC option `-fstack-protector` checks buffer overflow

# Out of bound access

## Read/write of the bound of an array

- Mismatch in the bound of an array: $[0, N-1]$ in C, $[1, N]$ in Fortran
- Out of bound reading can cause undefined behavior
- Out of bound writing can cause memory corruption

## What to do?

- Use Valgrind, it should show error
  `Invalid read/write of size X`

# Input/Output errors

## Errors when reading/writing in files

- Usually have an external cause:
  - Disk full
  - Quota exceeded
  - Network interruption
- System call will return an error or hang

## What to do?

- Always can check the return code
- Usually stop execution with an explicit message

# Race condition 1/3

"Debugging programs containing race conditions is no fun at all."
Andrew S. Tanenbaum, *Modern Operating Systems*

## Race condition

- A timing dependent error involving shared state
- It runs fine most of the time, and from time to time, something weird and unexplained appears

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{

  account->balance += amount;

}
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)


ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)



ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ  balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)



ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)


ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)



ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls **deposit(A,10)**

```
READ balance (0)



ADD   10
WRITE balance (10)
```

Thread 2 calls **deposit(A,1000)**

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)



ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

# Race condition 2/3

## Code example

```
void deposit(Account * account, double amount)
{
  READ balance
  ADD   amount
  WRITE balance
}
```

## Concurrent execution

Thread 1 calls `deposit(A,10)`

```
READ balance (0)



ADD   10
WRITE balance (10)
```

Thread 2 calls `deposit(A,1000)`

```
READ balance (0)
ADD   1000
WRITE balance (1000)
```

Result: *balance* is 10 instead of 1010

Without protection, any interleave combination is possible!

# Race condition 3/3

## Different kind of race conditions

- Data race: Concurrent accesses to a shared variable
- Atomicity bugs: Code does not enforce the atomicity for a group of memory accesses, *eg* Time of check to time of use
- Order bugs: Operations are not executed in order
  Compilers and processors can actually re-order instructions

## What to do?

- Protect critical sections: Mutexes, Semaphores, etc.
- Use atomic instructions and memory barriers (low level)
- Use compiler builtin for atomic operations[2] (higher level)

[2] https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gcc/_005f_005fatomic-Builtins.html

# Deadlock 1/3

Deadlock, photograph by David Maitland

*"I would love to have seen them go their separate ways, but I was exhausted. The frog was all the time trying to pull the snake off, but the snake just wouldn't let go."*

# Deadlock 2/3

## Code example

```
void deposit(Account * account,
             double amount)
{
  lock(account->mutex);
  account->balance += amount;
  unlock(account->mutex);
}
```

```
void transfer(Account * accA,
              Account* accB,
              amount)
{
  lock(accA->mutex);
  lock(accB->mutex);
  accA->balance += amount;
  accB->balance -= amount;
  unlock(accA->mutex);
  unlock(accB->mutex);
}
```

# Deadlock 3/3

## Concurrent execution

Thread 1 calls `transfer(A,B,10)`

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls `transfer(B,A,20)`

```
lock(B->mutex);

lock(A->mutex); // wait until
                   A is unlocked
...
```

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Deadlock 3/3

## Concurrent execution

Thread 1 calls `transfer(A,B,10)`

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls `transfer(B,A,20)`

```
lock(B->mutex);

lock(A->mutex); // wait until
                   A is unlocked
...
```

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Deadlock 3/3

## Concurrent execution

Thread 1 calls **transfer(A,B,10)**

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls **transfer(B,A,20)**

```
lock(B->mutex);

lock(A->mutex); // wait until
                   A is unlocked
...
```

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Deadlock 3/3

## Concurrent execution

Thread 1 calls `transfer(A,B,10)`

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls `transfer(B,A,20)`

```
lock(B->mutex);

lock(A->mutex); // wait until
                   A is unlocked

...
```

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Deadlock 3/3

## Concurrent execution

Thread 1 calls `transfer(A,B,10)`

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls `transfer(B,A,20)`

```
lock(B->mutex);


lock(A->mutex); // wait until
                   A is unlocked
...
```

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Deadlock 3/3

## Concurrent execution

Thread 1 calls `transfer(A,B,10)`

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls `transfer(B,A,20)`

```
lock(B->mutex);

lock(A->mutex); // wait until
                   A is unlocked
...
```

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Deadlock 3/3

## Concurrent execution

Thread 1 calls `transfer(A,B,10)`

```
lock(A->mutex);

lock(B->mutex); // wait until
                   B is unlocked

...
```

Thread 2 calls `transfer(B,A,20)`

```
lock(B->mutex);

lock(A->mutex); // wait until
                   A is unlocked
...
```

We have a deadlock!

## What to do?

- Think before writing multithread code
- Use high level programming model: Open MP, Intel TBB, MPI, etc.
- Theoretical analysis
- Software for thread safety analysis

# Performance bugs

## Bad Performance can be seen as a bug

- Bad algorithm: too high computation complexity
  Example: *Insertion Sort* is $O(N^2)$, *Quick Sort* is $O(N.log(N))$
- Memory copies can be a problem,
  specially with Object Oriented languages
- Some memory allocator have issues:
  memory alignment constraints, multithread context

## What to do?

- Try use existing proven libraries when possible:
  *eg* Eigen library for linear algebra, C++ STL, Boost, etc.
- Use a profiler to see where your program spend most of its time
  Valgrind with Callgrind, GNU gprof, many commercial tools ...
- ...

# Outline

4 Good practices to catch bugs

# Be a good programmer

## Write good code

- Use explicit variable names, don't re-use variable
- Avoid global variables (problematic in multi-threads)
- Comment and document your code
- Keep your code simple, don't try to over-optimize

## Use defensive programming

- Add assertions, *cf* `assert()`
- Always check return codes, *cf* manpages and documentation

## Re-use existing libraries

- Use existing libraries when available/possible
- Probably better optimized and tested than your code

⇒ Code easier to understand and maintain

⇒ Catch bugs as soon as possible

# Compilers and Tests

## Use your compilers

- Enable (all) warnings of the compiler
- Vary the compilers and configurations
  - Different compilers (GCC, CLang, Intel Compiler, MS Compiler)
  - Various architectures (Windows/Linux, x86/x86_64/ARM)

## Testing and Code Checking

- Write unit tests and regression tests
- Use coverage analysis tools
- Use static and dynamic code analysis tools
- Continuous integration:
  - Frequent compilation, testing, execution
  - Different configurations and platforms

⇒ Catch more warnings and errors

⇒ Better portability

# Know your tools

## Know the error messages

- Look in the documentation / online
- Compiler errors/warnings
- Runtime errors:

  `Segmentation fault`, `Floating point exception`, `Double free`, etc.
- Valgrind errors:

```
Invalid read of size 4
Conditional jump or move depends on uninitialised value(s)
8 bytes in 1 blocks are definitely lost
```

  ...

## Use the right tool

- Know your tools and when to use them
  - GDB: locate a crash
  - Valgrind: memory-related issue
  - ...

# Debug with methodology

## Find a minimal case to reproduce the bug

- Some bugs are intermittent
- Easier to debug
- Help you to understand the cause
- Allow to check that the bug is really fixed
- Bonus: make a regression test

## Use a Control Version System (GIT, SVN, ...)

- Keep history, serve as a backup, allow to go back in time
- GIT has a nice feature of code bisection in history to find when a bug has been introduced

# Thank you for your attention!

## Course Practice

Follow the SC-CAMP Practice in:

https://gitlab.uni.lu/SC-Camp/2019/debug