Game Connection 2012

Debugging memory stomps and other atrocities

molecular-matters.com

Motivation

- People make mistakes
- In relation to memory allocations, mistakes can be fatal

Common mistakes

- Forgetting to initialize class members
  - Code then relies on garbage or stale values
- Writing out-of-bounds
- Reading out-of-bounds
- Memory leaks

Common mistakes (cont'd)

- Stack overflows
  - Seldom in main thread
  - More often in other threads
- Memory overwrites
  - Off-by-one
  - Wrong casting
  - Reference & pointer to temporary
  - Dangling pointers

Possible outcomes

- From best to worst
  - Compile error, warning
  - Analysis tool error, warning
  - Assertion
  - Consistent crash or corruption
  - Random, unpredictable crash
  - Crashes after N hours of gameplay
  - Seemingly works, crashes at certification

Class member initialization

- Non-initialized members get garbage
  - Whatever the memory content is
- Lucky case
  - Leads to a crash later
- Unlucky case
  - Works, but not as intended
  - Subtle bugs N frames later

Class member initialization (cont'd)

- Extra painful with pool allocators
  - Most recently freed entry is re-used
  - Members point to stale data
- Remedy
  - Crank up compiler warning level
  - Fill memory with distinct pattern when allocating
  - Fill memory with distinct pattern when freeing
  - Use static code analysis tools

Fill patterns

- If you can, do not fill with zero
  - Hides bugs and uninitialized pointer members
    - E.g. delete 0 still works, but pointer was never initialized
  - Often not possible when dealing with legacy code
- Fill with „uncommon“ values
  - Numerically odd
  - Unused address range
  - E.g. 0xCD, 0xFD, 0xAB

Writing out-of-bounds

- Could stomp unrelated data
- Could stomp allocator book-keeping data
- Could write to non-committed memory
- Lucky case
  - Immediate crash
- Unlucky case
  - Weird behaviour N frames later

Writing out-of-bounds (cont'd)

- Remedy
  - Bounds checking
    - Insert guard bytes at the front and back of each allocation
  - Use static code analysis tools
  - Use page access protection

Reading out-of-bounds

- Reads garbage data
- Garbage could lead to crash later
- Read operation itself could also crash
  - Very rare, reading from protected page
- Worst case
  - Goes unnoticed for months
  - Seldomly crashes

Reading out-of-bounds (cont'd)

- Remedy
  - Bounds checking, somewhat insufficient
    - At least the garbage read is always the same
  - Use static code analysis tools
  - Use page access protection

Memory leaks

- Lead to out-of-memory conditions after hours of playing the game
- Can go unnoticed for weeks
- Different kinds of leaks
  - „User-code“ leaks
  - API leaks
  - System memory leaks
  - Logical leaks

Memory leaks (cont'd)

- Remedy
  - Memory tracking
    - Override global new & delete
      - Or build DIY functions
    - Hook low-level functions
      - Disassemble & patch (PC)
      - Weak functions (consoles)
      - Compiler-specific wrapping (consoles)
      - Linker-specific wrapping (--wrap, GCC)
  - Soak tests

Stack overflows

- An overflow can stomp single bytes, not just big blocks
  - Thread stack memory comes from heap
  - Stomps completely unrelated data
- Often lead to immediate crash upon writing
  - E.g. in function prologue

Stack overflows (cont'd)

- Debugging help
  - Find stack start & end
    - Highly platform-specific
    - Linker-defined symbols for executable
  - Fill stack with pattern
  - Check for pattern at start & end
    - Each frame
    - In offending functions

Stack overflows (cont'd)

```
void Log(const char* format, ...)
{
    char buffer[4096];

    // stuff...

    int someLocalVariable = 0;
}
```

Memory overwrites

- Off-by-one
  - Write past the end of an array
  - 0-based vs. 1-based
- Debugging tools
  - Bounds checking
    - Finds those cases rather quick
  - Static code analysis tools

Memory overwrites (cont'd)

- Wrong casting
  - Multiple inheritance
  - Implicit conversions to void*
    - No offsets added
  - reinterpret_cast
    - No offsets added
  - static_cast & dynamic_cast
    - Correct offsets

Memory overwrites (cont'd)

- Reference & pointer to temporary
  - Most compilers will warn
    - Crank up the warning level
  - Compilers are tricked easily
  - Reference vs. const reference
    - Reference to temp. = illegal
    - Const ref. to temp. = legal

Memory overwrites (cont'd)

- Dangling pointers
  - Point to already freed allocation
  - Lucky case
    - Memory page has been decommited
      - Immediate crash upon access
  - Unlucky case
    - Other allocation sits at the same address

Debugging optimized builds

- Debug builds
  - printf(), assert(), etc. slow down the application
    - Hides data races
- Crashes in optimized builds
  - Local variables mostly gone
    - Not on stack, but in registers (PowerPC!)
  - Memory window is your friend
    - Never lies!

Debugging optimized builds (cont‘d)

- Knowing stack frame layout helps
  - Parameter passing
  - Dedicated vs. volatile vs. non-volatile registers
- Knowing assembly helps
- Knowing class layout helps
  - V-table pointer, padding, alignment, …
- Verify your assumptions!

Debugging optimized builds (cont‘d)

- Familiarize yourself with
  - Function prologue
  - Function epilogue
  - Function calls
  - Virtual function calls
  - Different address ranges
    - Heap, stack, code, write-combined, …
- Study compiler-generated code

Debugging optimized builds (cont‘d)

- Help from the debugger
  - Cast any address to pointer-type in watch window
  - Cast zero into any instance
    - Work out member offsets
  - Pseudo variables
    - @eax, @r1, @err, platform-specific ones
  - Going back in time
    - „Set next instruction“

Debugging optimized builds (cont‘d)

Watch 1

Name	Value
(Derived*)0x00121d48	0x00121d48 {c=2 }
Base1	{a=0 }
__vfptr	0x00da87e0 const Derived::`vftable'{for `Base1'}
[0]	0x00da121c Derived::SomeVirtual(void)
a	0
Base2	{b=1 }
__vfptr	0x00da87d4 const Derived::`vftable'{for `Base2'}
[0]	0x00da1041 Derived::OtherVirtual(void)
b	1
c	2

Watch 1

Name	Value
&(((Derived*)0)->c),d	16

Watch 1

Name	Value
@eax,x	0x003b1d48
(Derived*)@eax	0x003b1d48 {c=2 }
Base1	{a=0 }
Base2	{b=1 }
c	2

Debugging optimized builds (cont‘d)

- Help from the debugger (cont'd)
  - Crash dumps
    - Registers
      - Work backwards from point of crash
      - Find offsets using debugger
      - Find this-pointer (+offset) in register, cast in debugger
    - Memory contents
  - On-the-fly coding
    - Nop'ing out instructions
    - Changing branches

Debugging optimized builds (cont‘d)

```
000228C6  mov         dword ptr [ebp-10Ch],0
000228D0  mov         ecx,dword ptr [ebp-10Ch]
000228D6  cmp         ecx,dword ptr [b2]
000228D9  jne         WrongCasting+0DFh (228EFh)
    {
        OutputDebugStringA("d equals b2\n");
000228DB  mov         esi,esp
```

Memory 1

Address: 0x000228D9

0x000228D9 75 14 8b f4 68 c0 87

```
000228C6  mov         dword ptr [ebp-10Ch],0
000228D0  mov         ecx,dword ptr [ebp-10Ch]
000228D6  cmp         ecx,dword ptr [b2]
000228D9  nop
000228DA  nop
    {
        OutputDebugStringA("d equals b2\n");
000228DB  mov         esi,esp
```

Memory 1

Address: 0x000228D9

0x000228D9 90 90 8b f4 68 c0 87

molecular-matters.com

Debugging overwrites

- Hardware breakpoints
  - Supported by debuggers
  - Supported by all major platforms
    - x86: Debug registers (DR0-DR7)
    - PowerPC: Data Access Breakpoint Register (DABR)
  - Help finding reads & writes
  - Setting from code helps finding random stomps
  - CPU only
    - GPU/SPU/DMA/IO access doesn't trigger

Debugging overwrites (cont‘d)

- Page protection
  - Move allocations in question to the start or end of a page
    - Restrict access to surrounding pages
    - Needs a lot more memory
  - Protect free pages
    - Walk allocations upon freeing memory
    - Don't re-use recently freed allocations

Debugging overwrites (cont'd)

- Page protection (cont'd)
  - Needs virtual memory system
  - If not available, most platforms offer similar features

If all else fails...

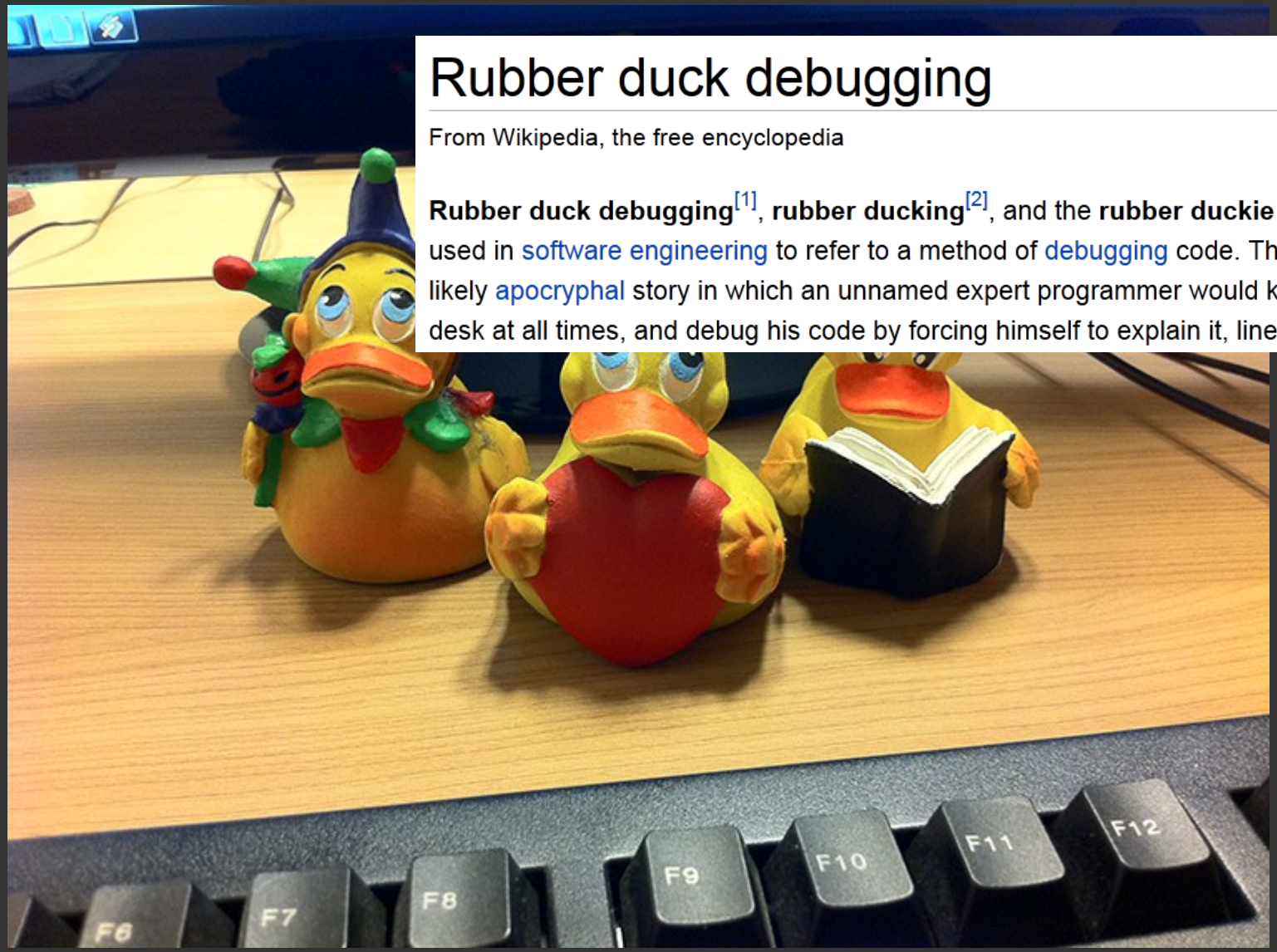
- Go home, get some sleep
  - Let your brain do the work
- Grab a colleague
  - Pair debugging
  - Fresh pair of eyes
- Talk to somebody
  - Non-programmers, other team members
  - Your family, rubber ducks, ...

Yes, rubber ducks!

molecular matters

molecular-matters.com

Yes, rubber ducks!



Rubber duck debugging

From Wikipedia, the free encyclopedia

Rubber duck debugging[1], **rubber ducking**[2], and the **rubber duckie test**[3] are informal terms used in software engineering to refer to a method of debugging code. The name is a reference to a likely apocryphal story in which an unnamed expert programmer would keep a rubber duck by his desk at all times, and debug his code by forcing himself to explain it, line-by-line, to the duck.

molecular matters

molecular-matters.com

That's it!

Questions?

Contact:
stefan.reinalter@molecular-matters.com

Bonus slides

Memory tracking

Memory tracking

- Two possibilities
  - Allocate & free memory with new & delete only
    - Override global new and delete operators
    - Static initialization order fiasco
      - Memory tracker must be instantiated first
      - Platform-specific tricks needed
    - Still need to hook other functions
      - malloc & free
      - Weak functions or linker function wrapping
    - Problems with 3rd party libraries doing the same

Memory tracking (cont'd)

- Static initialization order
  - Top to bottom in a translation unit
  - Undefined across translation units
  - Platform-specific tricks
    - #pragma init_seg(compiler/lib/user) (MSVC)
    - __attribute__ ((init_priority (101))) (GCC)
    - Custom code/linker sections

Memory tracking (cont'd)

- Other possibility
  - Disallow new & delete completely
    - Use custom functions instead
      - Allows more fine-grained tracking in allocators
    - Demand explicit startup & shutdown
    - Harder to do with legacy codebase
  - Need to hook platform-specific functions
    - Easier on consoles
    - Code-patching on PC

Memory tracking (cont'd)

- Hooking functions
  - Weak functions (if supported)
  - Implemented at compiler-level
    - XMemAlloc (Xbox360)
  - Function wrapping at linker-level
    - --wrap (GCC)
      - __wrap_* & __real_*
  - Code-patching (Windows)
    - Disassemble & patch at run-time

Memory tracking (cont'd)

- Storing allocations
  - Intrusive
    - In-place linked list
    - Changes memory layout
    - Cannot use debug memory on consoles
  - External
    - Associative container, e.g. hash map
    - Same memory layout as with tracking disabled
    - Can use debug memory on consoles

Memory tracking (cont'd)

- Levels of sophistication
  - Simple counter
  - File & line & function, size
  - Full callstack, ID, allocator name, size, …
- Tracking granularity
  - Global, e.g. #ifdef/#endif
  - Per allocator
    - More on that later

Memory tracking (cont'd)

- Finding logical leaks
  - Snapshot functionality
    - Capture all live allocations
  - Compare snapshots e.g. upon level start
  - Report snapshot differences