Memory safety in programming languages

Tom Peters, May 23, 2019

Roughly 3 categories of memory bugs

1. **Spatial** -- out-of-bounds access
   a. Heap-buffer overflow
   b. Stack buffer overflow
2. **Temporal**
   a. Use-after-free
   b. Stack use after return
   c. Stack use after scope
3. **Data races** - two concurrent accesses to a memory location, one of which is a write.

How do we deal with memory unsafety in C++?

- Best practices, code review
- Compiler warnings
- Static analysis
- Dynamic analysis

Static analysis

Analysis of program (either source or binary) in non-runtime environment.

Industrial tools ($):

- Klocwork
- Codesonar

Free tools:

- Clang-check
- Visual studio static analyizer
- CppCheck

Dynamic analysis

Catching bugs as they happen at runtime (mostly linux only).

- Valgrind suite, in particular memcheck
- Google Sanitizers, in particular AddressSanitizer

Comparison of memcheck, AddressSantizer

Asan wins:

- **Speed**
- Instrumentation of globals, better stack instrumentation, use after return, container overflow

Memcheck wins:

- Also checks (bitwise) validity
- Inline assembly, jitted code, external libs handled
- Requires no compilation, linking changes.
- Able to run alongside CUDA libraries.

Aside #1: cuda-memcheck

CUDA C/C++ are memory unsafe

NVIDIA toolkit comes with cuda-memcheck, much like Valgrind's memcheck, but for CUDA code

Aside #2: Data race detection

Dynamic tools for data race detection:

Helgrind, a Valgrind tool.

ThreadSanitizer, from Google/LLVM

Practical recommendations for memory safety

Use a combination of static and dynamic analysis.

How should you take advantage of dynamic analysis?

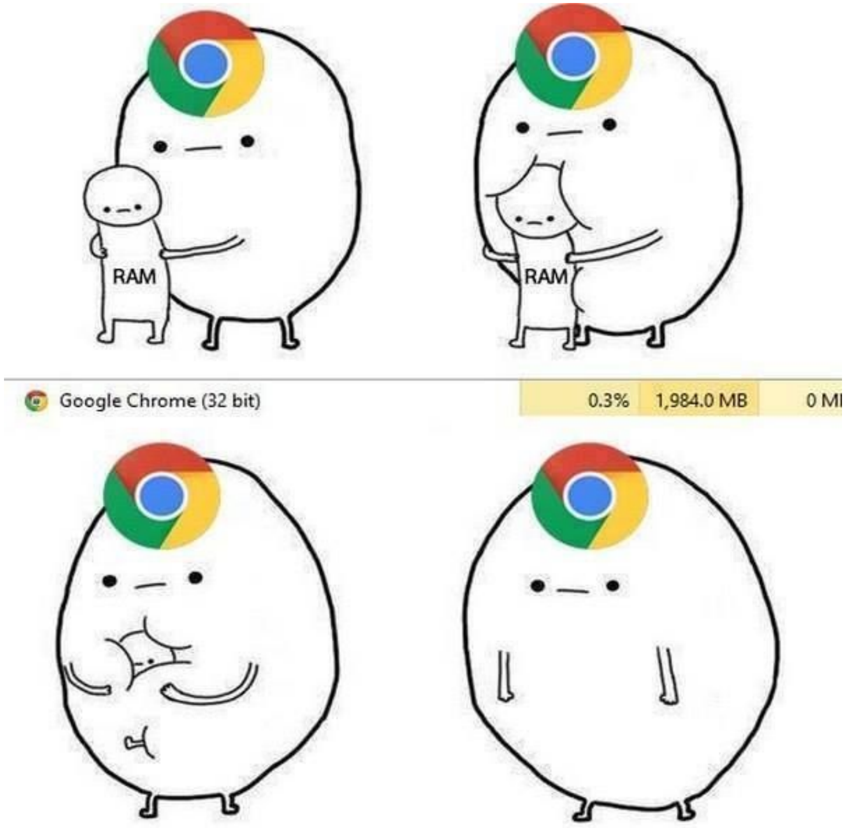
- Run them as necessary, much like debuggers.
- Keep on during development, or on "canary" releases.
- Run them on your unit tests, eg periodic memory checks.
  - This is particularly easy for valgrind and cmake: `ctest -T memcheck`
  - For sanitizers, it requires multiple rebuilds (no two sanitizers which require shadow memory can be run simultaneously)
- Run instrumented fuzz tests.

But it's not proof of safety

Eg, Google Chome:

- Large scale fuzzing
- Google developers

Can't find all the bugs.



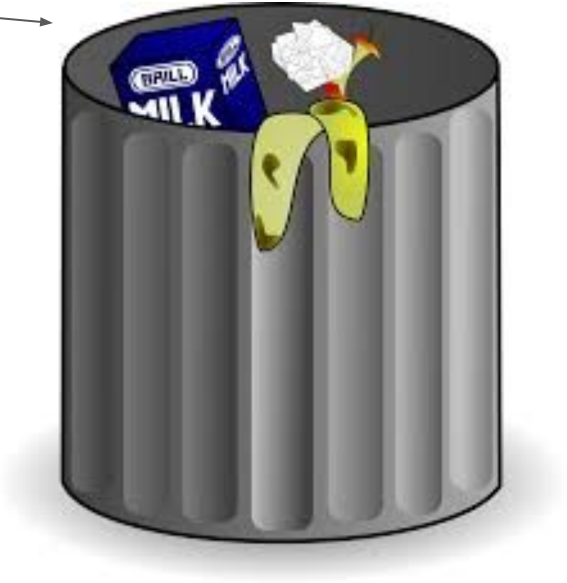
Other approaches

Most safe languages use a combination of runtime bounds checking and garbage collection.

Garbage collection

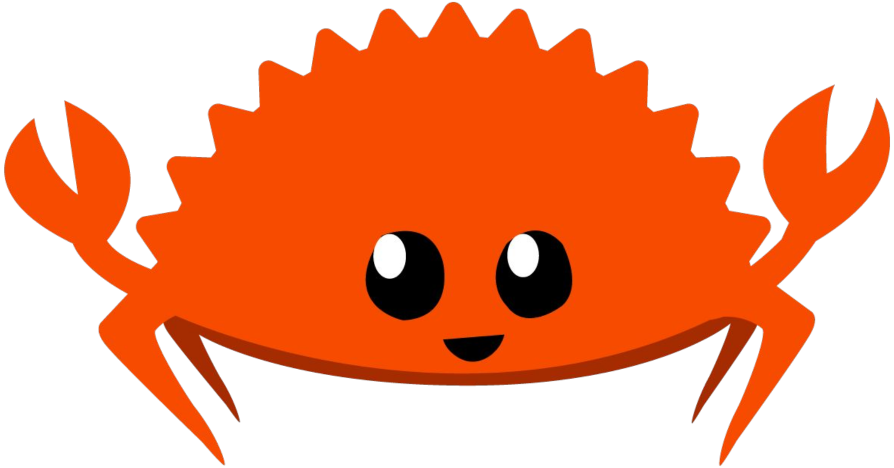
Java

C#



Is there another way?

Enter: **Rust**



Conclusion

Different approaches to memory safety:

- Memory unsafe: C, C++, Fortran -- require discipline knowledge and tools.
- Runtime GC: Java, C#, Haskell -- memory safe by default, but must pay cost of GC.
- Rust -- memory safe (by default), no GC.