Program failure seen from C

Jens Gustedt

INRIA – Camus
ICube – ICPS
Université de Strasbourg

https://gustedt.gitlabpages.inria.fr/modern-c/

Section 1

What about C?

Reality in the field

- C is one of the most used programming languages
  - operating systems
  - communication systems
  - visualization
  - embedded devices
  - high performance computing

- C is *the* description language for
  - processing capabilities
  - platform ABI
  - cross-language specification

Standardization

Timeline of the C language

Year	Name	Alias	Standard	Changes
1972	first release		—	
1978	K&R C		—	
1989	C89	ANSI C	ANSI X3.159-1989	++
1990	C90	ISO C	ISO/IEC 9899:1990	same
1995	C95		.../AMD1:1995	bugfix +
1999	C99	C9X	ISO/IEC 9899:1999	++
2011	C11	C1X	ISO/IEC 9899:2011	bugfix +
2018	C17		ISO/IEC 9899:2018	bugfix
2024	C23	C2X	ISO/IEC 9899:2024	++
202Y		C2Y	ISO/IEC 9899:202Y	++

Standardization

A tedious process

- constrained by the existing code base
- guided by existing compiler implementations
- driven by some passionate individuals
- time consuming
- slooooow
- supported by
  - very few companies (mostly US)
  - some academia (mostly EU)

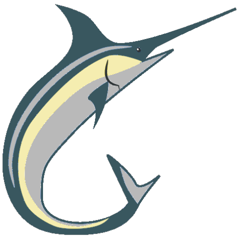
In France

- driven by AFNOR
- a national committee that is
  - historically interested in C++
  - open minded towards other programming languages

Standardization

POSIX

- C is closely tied to the development of Unix
- Single Unix Specification (SUS) — Portable Operating System Interface (POSIX)
- latest standards
  - ISO/IEC/IEEE 9945:2009/Cor 2:2017
  - ISO/IEC 9945:2024
- POSIX uses C as normative reference



- C is closely tied to the development of computing
- C describes the basic features of computing devices
- C is portable
- C is stable
- C is here to stay

Overview

Section 2

C's error model

Error model

Errors result in failure

- the best situation
  - compiler error
- visible manifestations of runtime errors
  - processor halt
  - crash (computer, plane, satellite, ...)
  - intrusion
  - program exit
  - raising a signal
  - calling a signal handler
  - calling a constraint handler
  - wrong results
  - loss of money
  - program state corruption
  - platform corruption
  - data corruption
  - nothing at all

Error models

Undefined Behavior versus Error

That, what is not defined … in the C standard

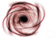
- Omission
- Identified error
  - detectable, but different resolution strategies
  - highly complex, undetectable
  - disputed
- Optimization point
- Open design space

Section 3

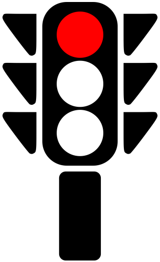
A program failure classification

A program failure classification

four classes

-  wrongdoings
-  program state degeneration
-  unfortunate incidents
-  series of unfortunate events

Wrongdoings



Wrongdoings

Arithmetic violations

- division by zero
- modulo by zero

These are math problems!

Wrongdoings

Arithmetic violations … continued

- negation of `INT_MIN`
- negative bit shift
- big positive bit shift
- bit shift into the sign bit

These are number representation/operation problems!

Wrongdoings

Arithmetic violations ... continued

- comparison of `signed` and `unsigned` integers

These are programming language design problems!

Wrongdoings

Arithmetic violations ... continued

- pointer addition that overflows array bounds
- pointer comparison if not the same array object

These computer architecture problems!

Wrongdoings

Arithmetic violations ... end

Check your operands!

Wrongdoings

Invalid conversions

- from an unsigned to a signed integer type, `UINT_MAX` → `signed int`
- between floating point and integers, `2147483648.0` → `signed int`
- between different floating point, `2147483648.0` → `float`
- from pointer to small integer, `p` → `unsigned int`
- from different pointer types, alignment!

Check your operands!

Don't use casts!

- Implicit conversions are mostly ok (with good compiler options)
- Explicit conversions (casts) are evil

Wrongdoings

Value violations

Invalid calls to the C library:

- calling functions with wrong arguments
  - null pointer
  - large number
  - zero size on allocation
- result of operation is not representable

Check your operands!

Wrongdoings

Type violations

- Accessing an object with the wrong type.
- Accessing a function with the wrong type.

Don’t use casts!

- Implicit conversions are mostly ok (with good compiler options)
- Explicit conversions (casts) are evil

Wrongdoings

Access violations, ...

- null pointer dereference
- accessing
  - a missing object
  - an element out-of-bounds
    - fixed: array length +1
    - dynamic: failed size tracking
  - a member of an atomic structure or union

Wrongdoings

Access violations, … continued

- modifying and reading from unsequenced subexpressions
- modifying an unmutable object
- storing from an overlapping object
- calling `free` for an already freed pointer

Wrongdoings

Access violations, ... end

- accessing
  - an element of a flexible array member with no elements
  - a `volatile` object from a non-`volatile` lvalue
  - an object based on a `restrict` pointer non-exclusively
  - a function through a falsely attributed prototype (`[[unsequenced]]`, `[[noreturn]]`)
- issuing a call to `longjmp` with a dead function context
- returning from a signal handler from a computational exception

Wrongdoings

Value misinterpretation

- Access of uninitialized object
- Access of object with “non-value representation”

Initialize, always!

Don’t fiddle with bits!

Don’t overlay types that have padding bits!

Wrongdoings

Explicit invalidation

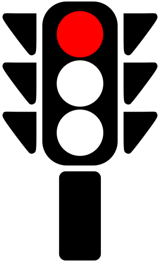
```
ptrdiff_t do_ptrdiff(unsigned char const* p,
                     unsigned char const* q) {
  if (!p || !q) unreachable();
  return p - q;
}
```

"I solemnly swear that execution will never reach this place!"

annotate the interface!

```
ptrdiff_t do_ptrdiff(unsigned char const p[static 1],
                     unsigned char const q[static 1]) {
  return p - q;
}
```

Wrongdoings



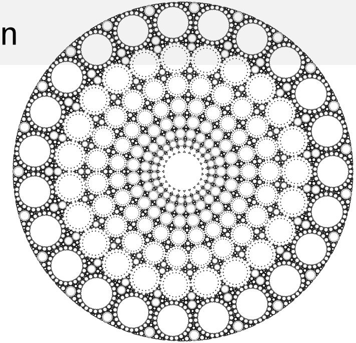
Stick to the rules!

- you need a good coding style
- you need a good compiler
- you need a good analyzer

Program state degradation

Program state degradation

Unbounded recursion



logical design error!

there is no generic solution

- when cautious: manifests as crash or infinite loop
- when unlucky: state corruption, data loss, money loss, crashing rockets, dead people

Program state degradation

Storage exhaustion

design and capacity problem

- when cautious: leaks caught at compile time or testing, errors caught at runtime
- when hazardous: state corruption, data loss, money loss, crashing rockets, dead people

Program state degradation

scarce system resources

- file (on disk or remote)
- memory
- bandwidth
- CPUs
- power

scarce process resources

- streams (# open `FILE`)
- function call contexts
- thread contexts
- mutexes
- condition variables
- thread-specific storage

Program state degradation

Monitor the program state

Not one single action at fault!

You are the traffic jam!

Unfortunate incidents

Unfortunate incidents

Collisions and race conditions

- between different processes
- between different threads
- with signal handlers
- when executing unsequenced expressions with side effects

use atomic tools

- on the file system
- for control data

no side effects in expressions!

Unfortunate incidents

Inappropriate library calls and macro invocations

- `signal` is allergic to multi-threading
- `setjmp`
  - is restricted to certain syntactic constructs
  - can only handle explicitly coded return values from `longjmp`
- **`#pragma`** can change rounding mode and other FP state

inform yourself!

- system manual
- C standard
- colleagues (caution!)
- Internet (caution!)

Unfortunate incidents

Deadlocks

avoid using locks!

use atomic tools where possible

Unfortunate incidents

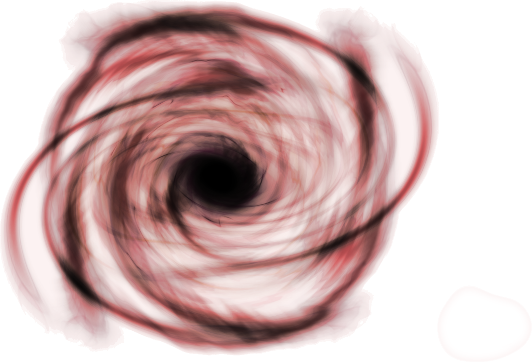
Escalating state degradation

- after having ignored warning signs from
  - wrondoings
  - program state degradation
- difficult to trace
- errors appear in seemingly random locations

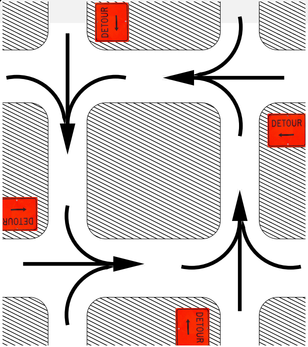
Never ignore an error indication!

- imminent risk: state corruption, data loss, money loss, crashing rockets, dead people

Series of unfortunate events



Series of unfortunate events



Livelock

Questions about goals and design!

- What is the global state that you want to achieve?
- Should there even be an exit?

Section 4

Dealing with possible failure

Avoiding failure

- C $\neq$ C++
- don't use casts
  - casts paint over design errors
  - implicit conversion `void*` $\rightarrow$ `data*` is fine
  - don't even cast the return of `malloc`!
- make your code zero-safe
  - zero is *the* universal value in C
  - default initialization uses it
  - for all data types the all-zero state must be valid

Avoiding failure

- initialize your variables
  - use initializers wherever possible
  - since C23, `{}` just works, even for VLA
- prefer `calloc` over `malloc`
- initialize static state needing runtime information
  - at the start of `main` before all threads
  - since C23, by means of `call_once`

Avoiding failure

- with C23 comes **constexpr**

```
constexpr int a = VERYBIGNUMBER;
```

  only works if value is well defined for the target type

- use **signed** and **unsigned** integers consistently
  - **sizeof** has the **unsigned** type **size_t**
- use **nullptr**
  - NULL is problematic, in particular as a sentinel

Avoiding failure

- use checked integer arithmetic
  - with C23 comes <`stdckdint.h`>

```
if (ckd_add(&result, a, b)) error_out();
```

- use proven tools for bit-fiddling
  - with C23 comes <`stdbit.h`>
  - `stdc_bit_width(x)` $\rightarrow 1 + \lfloor \log_2 x \rfloor$

Avoiding failure

- use [static n] parameters in headers and implementation
  - says that the caller has to provide at least n elements

```
int printf(const char format[restrict static 1], ...);
```

  - in particular, no null pointer
  - modern compilers can track misuse of null pointers
- use const qualification where you may
  - modern compilers can track if an object is modified/mutable

Avoiding failure

- use variable length array parameters *(VLA)*

```
void mycpy(size_t         n,
           double         x[restrict static n],
           double const   y[restrict static n]);
```

  - modern compilers can track the size of arrays
- use pointers to variable length arrays *(VLA)* for large allocations
  - permits comfortable use of multi-dimensional arrays
  - avoids erroneous index calculations
- use variable length arrays *(VLA)* for medium sized allocations
  - yes, this uses the stack (in general)
  - avoids over-pessimation of stack usage

Avoiding failure

- prefer atomics to locks
  - all accesses to an atomic variable are atomic

```
_Atomic(uint64_t) counter = 0;
...
++counter;                          // atomic operation
```

  - also the file system can be accessed atomically
    - `tmpfile`
    - `tmpnam`
    - `fopen` with mode `x`
- check the return of C library functions

Develop a failure model

What is tollerable?

- program crash
- user feedback
- controlled unwinding
- nothing

What is possible?

- signal handler
- `atexit` handler
- `at_quick_exit` handler
- thread specific destructors (`tss_dtor_t`)
- retry after
  - manual cleanup
  - garbage collection

Detecting faulty code

- use a modern compiler and modern C
  - `-Wall -std=C2x`
- use an analyzer
  - `-fanalyzer`
- use `valgrind` or similar for tests

No errors allowed!