Memory-Safety in C

Document: N3211
Author: Martin Uecker, TU Graz
Date: 2024-01-14

The goal of this paper is to explore the idea of memory safe compilation modes for C as strict subsets of the C language. This should be considered work in progress and will require much more work, but it is proposed here as a starting point that can be incrementally refined, made precise and improved.

The proposed modes will not make use of any language extensions and will not change semantics in incompatible ways, but instead aims to define a minimal subset of the language that is already memory safe and for which this can be easily be shown at compile time. Thus, a C program using the proposed memory safety modes can be compiled correctly by existing compilers also when ignoring the memory safety features. More importantly, existing code can gradually be made safe by activating the memory safety modes only for specific parts of the program and by refactoring the code where necessary. This is achieved by restricting the language using additional compile-time constraint and by requiring run-time traps for certain behavior. Initially, these safe modes will be severely restricted. Many restrictions can gradually be relaxed using some more static analysis and using additional annotations, e.g. using attributes to indicate bounds, life-time constraints (e.g. a borrow checker), etc.. Still, even a basic memory-safe mode can be useful to check specific critical parts of a larger code base. This, it will immediately provide value to programmers that want to understand whether specific parts of their program is memory safe. The specification can then be incrementally improved. Our goal is to preserve fast compilation speed and simple implementation of C compilers, so we aim to avoid expensive or complicated static analysis and concentrate on features that can be implemented in a compiler frontend for the compile-time part, while building on existing technology such as sanitizers for the run-time checking.

We consider a memory-safe operation an operation that (under certain preconditions specified below) can be shown at compile-time not to cause any run-time out-of-bounds accesses, i.e. accesses outside the bounds of the objects accessible via reachable pointers. In general, memory safety necessarily implies that there is no run-time undefined behavior, because undefined behavior includes out-of-bounds accesses. Nevertheless, we will ignore some compile-time undefined behavior. In principle, any compile-time undefined behavior would allow a compiler to produce an executable that later behaves arbitrarily at run-time, but this is more a conceptual than a real problem, which will (hopefully) be addressed elsewhere.

The basic idea to ensure memory safety is to always maintain certain run-time invariants: In particular, pointers are either NULL or point to at least one object of the correct type storing a valid value of this type and functions have exclusive access to writable pointers, etc. Thus, additional rules need to be enforced in safe mode that are not needed to directly prevent undefined behavior, but to maintain these invariants at all times. For example, it is forbidden to cast pointer types to other pointer types. Bounds safety then builds on top of type safety by forbidding unsafe pointer arithmetic or subscription of incomplete arrays. In the first versions of this proposal, we disallow many language constructs that could potentially be made safe later with additional care. We will also likely have overlooked some undefined behavior or made other mistakes. **This is work in progress and an experiment.**

Memory Safety Modes

We define two memory safety modes. A static and a dynamic mode, controlled by a pragma similar to floating point modes.

```
#pragma MEMORY_SAFETY STATIC | DYNAMIC | OFF
```

In **STATIC** memory-safety mode, no operation can have undefined behavior at run-time. All operations that could allow undefined behavior already violate a constraint at compile-time. The forbidden operations are listed below. (In the likely case that we missed an unsafe operation in this list, this is considered a defect in this specification.)

In **DYNAMIC** memory-safety mode, some operations that have undefined behavior at run-time in ISO C are allowed, but then defined to trap, i.e. they terminate the execution of the program before any out-of-bounds access can occur. All other operations that could allow undefined behavior are again already rejected at compile-time.

Thus, if the code compiles without diagnostic in any of the two memory safe modes, the programmer can be sure that there is no run-time UB. These modes are to be toggled at file-scope (this could later be relaxed to full expressions / declarations / statements).

Example (some unsigned arithmetic is already safe):

```
#pragma MEMORY_SAFETY STATIC

unsigned int f(unsigned int a, unsigned int b)
{
     return a * b;
}
```

In the following, we start by formulating a subset of the language which we believe is memory safe. This initially rules out many features of the C language. In the following, we will then make certain exceptions to these general rules. Use of any disallowed construct must be diagnosed at compile-time by a conforming compiler.

Generally Forbidden Operations / Constructs

Some of these operations are then allowed in STATIC or DYNAMIC mode under certain specific conditions as described below.

1. Pointer arithmetic and comparisons
2. Casts and implicit conversions that can have UB
3. Automatic variables without initializer
4. **goto**
5. Function calls
6. Declaration or use of variadic functions
7. Use of standard library macros or functions (in particular: memory management, **longjmp**, etc.)
8. rray subscription
9. Dereferencing of pointers

10. Use of the addressof operator
11. Use of variably modified types
12. Use of the **restrict** qualifier
13. Declaration of _**Noreturn** functions
14. Falling of the end of the function returning a value
15. Referring to a file-scope object of a type with possible non-value representations
16. Any use of attributes or objects/functions/types declared with attributes
17. Any use of unions
18. Arrays declared with **register**
19. Accessing a field of an atomic qualified structure or union
20. Use of storage class combinations that are UB and nested extern
21. value conversion of incomplete types
22. Declaration of use of structures / unions without members
23. Declaration or use of functions with qualifiers

Function call preconditions: Function call preconditions have to be guaranteed by the caller. This is generally true for a correct implementation. In safe mode, the preconditions can be made stricter than in regular C which helps working around some of the issues that make it difficult to improve memory safety in regular C code. At the same time, safe mode then has to ensure that a violation of the precondition by a safe-mode caller is not possible. When proving safety of the callee it can then be assumed that the preconditions are true. Essentially, this shifts responsibilities from a callee to a caller. In particular, we will assume that a pointer argument implies that the pointer is NULL or dereferencable, i.e. the pointed-to object exists, has a life time for the duration of the execution of the callee, has a valid (value) representation, does not partially overlap with any other object (but can be contained in), and that a writable object can exclusively be accessed by the callee for the duration of the execution. An argument pointer can be NULL except when annotated with **[static 1]**.

General Assumption / Requirements on the Implementation: All variables or functions defined (also in a different TU) are accessed using a declaration of a compatible type, have correct alignment and size, matching noreturn, matching array sizes and use of **static** in prototypes. Across TUs, this is currently **not** checked by most toolchains, but such consistency checks then need to be supported somehow, for example by checking debug information at link-time. Stack overflow must be reliably intercepted by the implementation and then cause a trap.

STATIC Memory-Safe Mode

In static safe modes, we allow certain operations where one can statically ensure that there is no run-time undefined behavior. We aim for rules that do not require expensive or complicated static analysis.

Main Invariant: Non-null pointers received as arguments point to arrays of known **constant** size with valid content.

Allowed operations:

- Arithmetic that can not cause UB such as unsigned integers without division/shits (possibly also some arithmetic using very small types), or checked arithmetic operations
- Subscription of arrays with known length n and of parameters declared as arrays with **[static n]** where n is an integer constant expression larger than zero
- Use of control structures such a **do, for, while, switch, if, break, continue** (TODO: UB related to infinite loops)
- Calling other STATIC memory-safe functions directly (i.e. not via pointers). All declarations of a function must have the same safety mode.
- Assignment of pointers to variables and passing of pointers to functions.
- Passing of pointer arguments declared with **[static n]** to other functions as such arguments
- Passing of pointer expressions using addressof applied to automatic variables as arguments declared with **[static 1]**
- Use of pointer expressions using addressof applied to automatic variables as first argument of checked arithmetic.
- Passing of pointer expressions with array-to-pointer decay as **[static n]** where n is the integer constant expression with n larger than zero and smaller or equal than the length of the array.
- Returning of pointers received as arguments.

Examples:

```
#include <limits.h>
#include <stdckdint.h>
#include <stdio.h>

#pragma MEMORY_SAFETY STATIC

static int foo(const int a[static 2])
{
     int r = 0;
     if (ckd_mul(&r, a[0], a[1]))
          return -1;
     return r;
}

static int bar(int x)
{
     int a[2] = { x, x };
     return foo(a);
}
```

DYNAMIC Memory-Safe Mode

In dynamic mode we additionally allow operations where we can reliably detect a memory-safety violation at run-time using the information that is readily available (i.e. not tracking of complicated additional state). A memory safety violation is then **required** to be detected at run-time and needs to trap, i.e. terminate the program without allowing other operations.

Main Invariant: Pointers received as arguments are either NULL or point to arrays of known size with valid content.

Allowed Operations

- All arithmetic.
  Requirement: Signed overflow, division by zero, invalid shifts, etc. is required to trap. (UB sanitizer: signed-integer-overflow)

- All arithmetic conversions between arithmetic types.
  Requirement: Conversion is required to trap when the target type can not represent the value.

- Declaration and definition of VLAs and use of VM types.
  Requirement: The use of negative sizes as well as the construction of types with sizer larger than the maximum size is required to trap. (UB sanitizer: vla-bound for $N < 0$)

- Subscription of arrays of known length and parameters declared as such arrays.
  Requirement: Subscription with negative values and values equal or larger than the bound is required to trap. (UB sanitizer: bounds)

- Assignment of values to objects involving VM-types.
  Requirement: Mismatching corresponding bounds cause a trap. (patch for GCC exists)

- Size expression in parameters must be side-effect free and be equivalent in all declarations of the same function and when forming composite type of two function types (partial support in GCC)

- Function calls to DYNAMIC and STATIC memory-safe functions.
  Requirement: When calling a function with VM-types as parameters, the size expression is additionally evaluated on the caller side and the result is compared to the size of the type of the passed expression. For a mismatch a trap must be generated. (patch for GCC exists)

- Dereferencing of all pointers passed as arguments or returned from a function.
  Requirement: When a NULL pointer is dereferenced, a trap is generated. (UB sanitizer: null)

- Function calls via pointers.
  Requirement: For a NULL pointer, a trap is generated. (UB sanitizer: null)

- Noreturn functions can be declared, defined, and called.
  Requirement: A trap must be generated in the callee if such a function returns. (UB sanitizer: unreachable)

Example:

```
#include <stdio.h>

#pragma MEMORY_SAFETY DYNAMIC

static int prod(int n, const double a[n])
{
     if (n > 1)
          return a[n - 1] * prod(n - 1, a);
     return a[0];
}

static int bar(int x)
{
     double a[7] = { 7, 5, 3, 2, x, x, x };
     return prod(7, a);
}

#pragma MEMORY_SAFETY OFF

int main(void)
{
     printf("%d\n", bar(1));
     return 0;
}
```

Next Steps

- An initial prototype implementation will be completed.
- The rules need to be checked for completeness and need to be formulated precisely. In particular the following omissions need to be addressed:
  - Rules related sequence points .
  - Requirements for cross-TU type compatibility
  - Rules for concurrency
- New rules to allow more operations.
  - Annotations for accessing objects via pointers in structures
  - Annotations for pointer ownership for dynamic memory management.
  - Annotations for safe use of unions
  - Implicit tracking of bounds for regular C pointers (cf. Clang bounds checking project)
  - Light-weight flow-sensitive analysis (e.g. assuming that an integer or pointer is non-zero in a branch of an if statement checking this condition, also see N3196)