Proposal for C2y
WG14 N3322

Title:	Allow zero length operations on null pointers
Author, affiliation:	Nikita Popov, Red Hat
	Aaron Ballman, Intel
Date:	2024-08-28
Proposal category:	Change in requirements
Target audience:	Implementers, power users

Abstract: Proposes allowing null + 0, null - null and passing null to certain C standard library functions if the length argument is also zero. This removes a class of undefined behavior and brings the specification more in line with C++.

Prior art: C++, LLVM

Allow zero length operations on null pointers

Reply-to: Nikita Popov (npopov@redhat.com), Aaron Ballman (aaron@aaronballman.com)
Document No: N3322
Revises Document No: N3261
Date: 2024-08-28

Summary of Changes

N3322

- Added more information about subtracting null pointer values in different address spaces
- Updated references to latest C2y draft

N3261

- Updated several footnotes
- Added wording to allow ordered comparison of two null pointers

N3177

- Initial proposal

Introduction

This proposal seeks to make the following three operations involving null pointers and zero lengths well-defined:

- Adding or subtracting 0 to/from a null pointer results in a null pointer.
- Subtracting two null pointers results in 0.
- Calling `memcpy`, `memmove`, `memset`, `memcmp` and various other functions with a null pointer argument and length 0 is well-defined.

The primary motivation is to avoid certain pitfalls and performance issues when using the combination of a null pointer and zero length to represent an empty slice/span.

Additionally, this aligns C semantics closer to C++ semantics, which already considers the first two cases well-defined.

Current wording

From 6.5.6 (Additive operators) paragraph 9:

When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. [...] ***If the pointer operand and the result do not point to elements of the same array object or one past the last element of the array object, the behavior is undefined****. [...]*

From 6.5.6 (Additive operators) paragraph 10:

When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object*; the result is the difference of the subscripts of the two array elements. [...]*

From 7.24.5 (Searching and sorting utilities) paragraph 1:

These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as size_t nmemb specifies the length of the array for a function, nmemb can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, and sorting performs no rearrangement. ***Pointer arguments on such a call shall still have valid values, as described in 7.1.4.***

From 7.26.1 (String function conventions) paragraph 2:

Where an argument declared as size_t n specifies the length of the array for a function, n can have the value zero on a call to that function. ***Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4.*** *On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.*

The wording for the mem* functions does not "explicitly state otherwise", so it applies to these functions.

From 7.31.4 (General wide string utilities) paragraph 2:

Where an argument declared as size_t n determines the length of the array for a function, n can have the value zero on a call to that function. ***Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4.*** *On such a call, a function that locates a*

wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters.

Motivation

It is common to use null pointers for the data pointer of empty containers or views. For example, a known-length string could be represented using

```
struct str {
    char *data;
    size_t len;
};
```

where data may be a null pointer for empty strings. A string equality operation on such strings could naively be implemented as follows:

```
bool str_eq(const struct str *str1, const struct str *str2) {
    return str1->len == str2->len &&
           memcmp(str1->data, str2->data, str1->len) == 0;
}
```

However, this would result in undefined behavior if both str1 and str2 were empty. Instead, a variant like the following needs to be used:

```
bool str_eq(const struct str *str1, const struct str *str2) {
    return str1->len == str2->len &&
           (str1->len == 0 ||
            memcmp(str1->data, str2->data, str1->len) == 0);
}
```

Similarly, a naive implementation of concatenation could look like this:

```
void str_concat(struct str *res, const struct str *str1,
                const struct str *str2) {
    res->len = str1->len + str2->len;
    res->data = malloc(res->len); // error handling omitted for brevity
    memcpy(res->data, str1->data, str1->len);
    memcpy(res->data + str1->len, str2->data, str2->len);
}
```

However, this would once again cause undefined behavior if str1 or str2 are empty. Instead, explicit zero length checks are necessary:

```
void str_concat(struct str *res, const struct str *str1,
                const struct str *str2) {
    res->len = str1->len + str2->len;
    res->data = malloc(res->len); // error handling omitted for brevity
    if (str1->len != 0) {
        memcpy(res->data, str1->data, str1->len);
    }
    if (str2->len != 0) {
        memcpy(res->data + str1->len, str2->data, str2->len);
    }
}
```

The requirement for the additional zero-length checks has a number of disadvantages:

- The check increases code size, potentially in common (inlined) code paths.
- The check decreases performance. Unless empty strings are common, it is preferable to fall through to a call, which will handle the zero-length case implicitly.
- The check increases code complexity and requires careful attention from the programmer to perform it in all necessary cases.
- Failure to perform the null check may result in a security vulnerability.

This problem extends beyond library functions to pointer arithmetic. Consider the following two implementations of a memcpy-like function:

```
void copy(char *dst, const char *src, size_t n) {
    for (size_t i = 0; i < n; i++) {
        *dst++ = *src++;
    }
}

void copy2(char *dst, const char *src, size_t n) {
    for (const char *end = src + n; src < end; src++) {
        *dst++ = *src;
    }
}
```

These functions differ by whether they use an integer or a pointer as the induction variable. While the former implementation is well-defined if src is a null pointer, the latter is not. Instead, it would be necessary to explicitly guard against zero lengths:

```
void copy2(char *dst, const char *src, size_t n) {
    if (n == 0) {
        return;
    }
    for (const char *end = src + n; src < end; src++) {
        *dst++ = *src;
    }
}
```

As some library function implementations use these kinds of patterns, it is important to change the requirements for both at the same time.

The pointer subtraction case is less common in practice, but may come up when a span/slice is represented using a start and end pointer, instead of a start pointer and length:

```
struct slice {
   // Both may be NULL for empty slice.
   char *data;
   char *end;
};

size_t slice_len(struct slice *s) {
   return s->end - s->data;
}
```

Additionally, this is important for self-consistency: If `p+i` is well-defined and produces `p2`, then `p2-p` should also be well-defined and produce `i`. As such, `null - null` should be well-defined and produce `0`, at least when the pointers are within related address spaces.

Optimization impact

Library changes

Implementations for memory library functions will automatically handle the n==0 case correctly, without dereferencing pointers. The current requirement for a non-null pointer for n==0 does not allow implementations to assume that there is at least one dereferenceable byte, because the pointer may legally point to the end of an object.

As such, allowing null pointers for n==0 will not require additional checks or other pessimizations in C library implementations.

While not useful for library implementations, the current requirements *can* improve optimization at the call-site. For example, a memcpy call on a known null pointer but unknown length can be statically determined to cause undefined behavior, and the relevant code path optimized away.

After the proposed change, this would only be possible if the compiler can prove that the length is non-zero. Of course, preventing this optimization from happening is an intentional part of the proposed change (to enable other, more useful optimizations).

Pointer arithmetic changes

Clang/LLVM already do not perform any optimizations based on the fact that null+0 and null-null result in undefined behavior. We are not aware of any practical cases where such optimizations would be useful.

In fact, LLVM has recently relaxed its internal IR requirements to make adding zero to any pointer (including "invalid" pointers) well-defined. This resulted in significant optimization *improvements*, because it made transforms like replacing `(c ? p : p + i)` with `p + (c ? 0 : i)` well-defined.

Architectures with multiple null pointer values

Some architectures may have multiple null pointer values. For such architectures, the requirement that subtracting any two null pointer values results in zero could, in theory, impose additional performance overhead or implementation complexity. However, there is reason to believe that this is not the case in practice.

All null pointer values (of compatible type within the same address space) are already required to compare equal. As such, if an architecture can support efficient pointer comparison, it stands to reason that it will also be able to support efficient pointer subtraction, which will now have the equivalent requirement.

To provide a specific example, in CHERI architectures, pointers consist of two parts: An address and metadata. As such, there may be multiple null pointer values, where the address part is zero, while the metadata is different. However, pointer subtraction on CHERI architectures is defined as the subtraction of the address parts only. As such, the new requirement is satisfied automatically, without additional performance overhead or implementation complexity.

The behavior of subtracting null pointer values in different address spaces depends on whether the address spaces are disjoint, equivalent, or have a subset relationship. Similar to casting behavior for null pointers, subtraction of two null pointers in different address spaces is only

valid if the address spaces are equivalent or have a subset relationship. As usual, this constraint can be circumvented by a cast of one or both of the null pointer values to a common type. Note, in TR 18037:2008, overlapping address spaces are required to perform masking operations to support equality testing (which is expected to occur more often than pointer subtraction in practice), so the overhead involved in pointer subtraction should not be a burden.

Current implementation behavior

Compiler behavior

Clang never makes use of the fact that null+0 and null-null result in undefined behavior for optimization purposes. Under `-fsanitize=undefined`, Clang diagnoses adding zero to a null pointer, but does not diagnose subtracting two null pointers.

GCC does not diagnose either case under `-fsanitize=undefined`. It is unknown whether it performs optimizations despite the failure to diagnose, but simple tests do not show GCC taking advantage of the UB: https://godbolt.org/z/qob5v6fh4.

Clang only respects `__attribute__((nonnull))` on function definitions, but does not respect it on declarations. This is done specifically so that nonnull annotations in libc headers are ignored. This does not affect sanitizer behavior or compiler warnings, which do respect the nonnull attributes.

GCC does respect nonnull attributes on declarations, and will optimize based on them.

Libc behavior

An interesting consideration is to what degree existing libc implementations are compatible with the proposed changes for library call requirements. That is, are any issues expected if code relying on the new requirements is linked against an old libc implementation?

Glibc declares mem* functions in headers using the following pattern:

```
extern int memcmp (const void *__s1, const void *__s2, size_t __n)
    __THROW __attribute_pure__ __nonnull ((1, 2));
```

The `__nonnull` macro will expand to `__attribute__((__nonnull__ params))` when headers are included by users of glibc, and as such the non-null requirements will apply. However, when the glibc implementation *itself* is compiled, the macro will be defined as empty, such that the implementation itself never introduces non-null assumptions, even when compiled with sanitizers. (The actual implementation will often be provided in assembly, in which case non-null assumptions are not made for the reasons outlined above: They aren't useful for anything.)

Musl libc and Apple libc do not use nonnull annotations even in user-facing headers. For example, memcmp is declared as follows:

```
int memcmp (const void *, const void *, size_t);
```

Bionic libc annotates pointers using `_Nonnull`:

```
int memcmp(const void* _Nonnull __lhs, const void* _Nonnull __rhs, size_t __n)
    __attribute_pure__;
```

This means that null pointers can be diagnosed, but are explicitly not considered undefined behavior.

Libc implementations that currently use some form of nonnull annotations should remove them to comply with this change, but compilers can also mitigate their presence in the same way clang currently does. No issues are expected when linking against old libc objects.

Proposed Wording

The wording proposed is a diff from the N3301 working draft of ISO/IEC 9899. Green text is new text, while ~~red~~ text is deleted text.

Modify 6.5.7p9:
… the expression (Q)-1 points to the last element of the array object. If the pointer operand is not null, and the pointer operand and result do not point to elements of the same array object or one past the last element of the array object, the behavior is undefined. If the pointer operand is a null pointer value and the integer operand is nonzero, the behavior is undefined.[fnt)] If the addition or subtraction produces an overflow, the behavior is undefined. If the pointer operand is null or the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.

Add new footnote:
[fnt)] Thus, the expression `ptr + N` (where `ptr` is a null pointer value) is defined to result in a null pointer value when `N` is zero and has undefined behavior otherwise.

Modify 6.5.7p10:
When two pointers are subtracted, both shall be null pointer values, point to elements of the same array object, or one past the last element of the array object~~;~~. If both pointers are null pointer values, the result is zero. Otherwise, the result is the difference of the subscripts of the two array elements.

Modify 6.5.9p6:
When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to, if any. If two pointers to object types are both null pointer

values, both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression P points to an element of an array object and the expression Q points to the last element of the same array object, the pointer expression Q+1 compares greater than P. In all other cases, the behavior is undefined.

Modify 7.24.6p1:
These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as size_t nmemb specifies the length of the array for a function, nmemb can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, and sorting performs no rearrangement. Pointer arguments on such a call ~~shall still have valid values, as described in 7.1.4~~ may be null pointers.

Remove footnote 351:
~~If the argument is a null pointer and the call is executed, the behavior is undefined.~~

Modify 7.26.1p3:
Where an argument declared as size_t n specifies the length of the array for a function, n can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call ~~shall still have valid values, as described in 7.1.4~~ may be null pointers. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

Modify footnote 361:
The null pointer constant is not a pointer to a const-qualified type, and therefore the result expression has the type of a pointer to an unqualified element~~; however, evaluating such a call is undefined~~.

Modify 7.31.4p2:
Where an argument declared as size_t n determines the length of the array for a function, n can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call ~~shall still have valid values, as described in 7.1.4~~ may be null pointers. On such a call, a function that locates a wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters.

Modify footnote 409:

The null pointer constant is not a pointer to a const-qualified type, and therefore the result expression has the type of a pointer to an unqualified element~~; however, evaluating such a call is undefined~~.

Affected and not affected functions

The following table lists which functions from the relevant section are affected or not affected by the change.

Function	Affected	Comment
bsearch	**Yes**	
qsort	**Yes**	
memcpy	**Yes**	
memccpy	**Yes**	
memmove	**Yes**	
strcpy	No	No argument n
strncpy	**Yes**	
strdup	No	No argument n
strndup	**Yes**	
strcat	No	No argument n
strncat	**Yes**	
memcmp	**Yes**	
strcmp	No	No argument n
strcoll	No	No argument n
strncmp	**Yes**	
strxfrm	No	This is an "unless otherwise specified" function. n == 0 is used to determine the size of the required buffer.
memchr	**Yes**	
strchr	No	No argument n
strpbrk	No	No argument n

strrchr	No	No argument n
strstr	No	No argument n
strtok	No	No argument n
memset	**Yes**	
memset_explicit	**Yes**	
strerror	No	No argument n
strlen	No	No argument n
wcstod, wcstof, wcstold	No	No argument n
wcstodN	No	No argument n
wcstol, wcstoll, wcstoul, wcstoull	No	No argument n
wcscpy	No	No argument n
wcsncpy	**Yes**	
wmemcpy	**Yes**	
wmemmove	**Yes**	
wcscat	No	No argument n
wcsncat	**Yes**	
wcscmp	No	No argument n
wcscoll	No	No argument n
wcsncmp	**Yes**	
wcsxfrm	No	See strxfrm
wmemcmp	**Yes**	
wcschr	No	No argument n
wcscspn	No	No argument n
wcspbrk	No	No argument n
wcsrchr	No	No argument n

wcsspn	No	No argument n
wcsstr	No	No argument n
wcstok	No	No argument n
wmemchr	**Yes**	
wcslen	No	No argument n
wmemset	**Yes**	

Acknowledgements

We would like to recognize the following people for their help in this work: Jens Gustedt, Joseph Myers, David Stone, Robert Seacord, and David Benjamin.