Introduce the term storage instance

Jens Gustedt

To cite this version:

Jens Gustedt. Introduce the term storage instance. [Technical Report] N2388, ISO JTC1/SC22/WG14. 2019. hal-02046329v2

HAL Id: hal-02046329

https://hal.inria.fr/hal-02046329v2

Submitted on 28 Jun 2019

ISO/IEC JTC 1/SC 22/WG14 **N 2388**

June 7, 2019 **v 2**

Introduce the term storage instance
Modification request for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

There is a lack of terminology to describe the entity that is reserved and released by either an allocation (`malloc`/`free`) or by the definition of a variable or compound literal. We introduce the new term *storage instance* to distinguish it clearly from the term *object*.

1. INTRODUCTION

The current revision of the C standard has no precise words to describe the maximal area of storage that is either obtained from

— allocating through `malloc`/`calloc`/`realloc`/`aligned_alloc` (allocated storage duration)
— instantiations of objects through the encouter of definitions (all other storage durations).

Already the term *storage duration* suggest that the "something" that is created through such an event would be "storage", but there are no precise words for it. In the contrary these beast are called very differently in different places. Some citations from the C standard:

— ... a *new instance of the object* is created each time ..
— ... *Allocated objects* have no declared type. ...
— ... that would not make the structure *larger than the object* being accessed ...
— The value of a pointer that refers to *space* deallocated by a call to the `free` or `realloc` function ...
— ... functions return a null pointer or a pointer to *an allocated object* ...
— The `longjmp` that returns control back to the point of the `setjmp`-invocation might cause *memory associated* with a variable length array object to be squandered.

The terms "space", "storage", "region of storage", "memory", and (maximal)"object" describe basically all the same thing, namely a byte array that is reserved by a certain event during execution (or startup). Also, the term "storage" is also used as a short hand for external storage devices, which does not help to clarify the terms.
Especially the use of the term "object" for such a varying range of concepts is unfortunate and produces a lot of confusion. There is a footnote

> *When referenced, an object can be interpreted as having a particular type.*

This seems to imply that all objects, in contrast to "allocated space" for example, have a type. Also objects can have subobjects, e.g the members of a structure object are themselves objects.

2. FIX TERMINOLOGY

We propose to add the term *storage instance*, as being a

> *the inclusion-maximal region of data storage in the execution environment that is created when either an object definition or an allocation is encountered*

Two aspects of this definition are important and distinguish this term from the term object:

(1) *A storage instance is inclusion-maximal.* In contrast to that "objects" can also be subobjects of a larger object.
(2) *A storage instance is created by a specific event, an object definition or an allocation.* By the effective type rule objects can pop into existence when allocated storage is written

by using an lvalue of a specific type, and thus may not have such a defining event that creates them.

The choice for the term itself (storage instance) stems from the fact that it seemed the easiest to integrate to the closely related concepts of *storage duration* and *storage class*. Also, this ensures consistent terminology: storage should be the entity that is target of a *store* operation. The *instance* part of the term is important to emphasize that this is an entity that has temporal limits within the execution.
Other terms than are specified as we go:

— the *start address* as the address of the first byte of a storage instance;
— the *end address* as the one-past address of the storage instance;
— the *byte address* of an object as being the address of the first byte in the storage instance that represents it;
— the *byte offset* of an object as being the byte position of the first byte in the storage instance that represents it;
— the *address space* of a program execution by the collection of all the byte addresses that occur during that execution.

Other properties of storage instances:

— Each addressable storage instance gives access to an array of untyped bytes that has the allocated or defined size and that has a constant address, if any.
— Each object definition and each call to an allocation function creates a new and unique storage instance, even if an address for the byte array is reused.
— Some object literals can share storage instances that represent them, namely string literals and compound literals with a const-qualified type.
— Two distinct storage instances that are alive simultaneously have disjoint byte arrays.
— The end address of one storage instance can be the start address of another. These are then said to *follow* each other *immediately*.
— No assumption about the specific ordering of storage instances in the address space can be made, neither through syntax (declaration order) or sequencing of allocation events.

3. SUGGESTED CHANGES

This proposal is only intended to clarify the existing model and not to add any new features. For clarification, realitively few text additions and modifications are needed. They can all be found in the appendix that consists of the relevant pages of diff-mark to the current C2x draft. *Beware*, that these pages are *not contiguous*.

3.1. Text additions

We propose three text additions, that introduce the term and put it into context:

***3.19new*:.** A definition of the term with four notes that clarify where “storage instances” come from, their relative placement and accessibility by different threads.
***6.2.6.1, p1*:.** This puts storage instances into their context (object representations) and states their basic properties, in particular that most of them can be viewed as a byte array. A footnote clarifies the absence of any induced positioning between any storage instances.
***6.2.6.1, p3*:.** Introduce object representations more clearly.

3.2. Text modifications

With these additions there are two types of text modifications remaining, namely some that really only are replacements of terms (space → storage instance, e.g.) and others that undergo deeper changes. For the latter we have:

6.2.4:. Here the paragraphs 1 and 2 are swapped, and the now paragraph 2 is a bit sharpened.

6.4.5, p7 *and* ***6.5.2.5, p7***. We unify the footnotes that state that storage instances may be shared for string literals and compound literals.

6.5, p18:. Clarify the extend of a flexible array member.

6.7, p5. We emphasize on the fact that the storage instance created by a variable definition is unique.

7.22.3:. The text for "Storage management functions" is clarified by a consequent use of the new terminology. Clarification that **`realloc`** does a byte-wise copy of the initial part (as if by **`memcpy`**), and a detailed note (7.22.3.5 p5) to emphasize on the implications for the created storage instance.

J.1. The fact that the relative ordering of storage instances is unspecified is summarized in a single bullet point.

Appendix: pages with diffmarks of the proposed changes

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

contains four separate memory locations: The member a, and bit-fields d and e.ee are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields b and c together constitute the fourth memory location. The bit-fields b and c cannot be concurrently modified, but b and a, for example, can be.

3.15

1 **object**

region of data storage in the execution environment, the contents of which can represent values

2 **Note 1 to entry:** When referenced, an object can be interpreted as having a particular type; see 6.3.2.1.

3.16

1 **parameter**

formal parameter

DEPRECATED: formal argument

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.17

1 **recommended practice**

specification that is strongly recommended as being in keeping with the intent of the standard, but that might be impractical for some implementations

3.18

1 **runtime-constraint**

requirement on a program when calling a library function

2 **Note 1 to entry:** Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.

3 **Note 2 to entry:** Implementations that support the extensions in Annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.

4 **Note 3 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.19

1 **storage instance**

the inclusion-maximal region of data storage in the execution environment that is created when either an object definition or an allocation is encountered

2 **Note 1 to entry:** Storage instances are created and destroyed when specific language constructs (6.2.4) are met during program execution, including program startup, or when specific library functions (7.22.3) are called.

3 **Note 2 to entry:** A given storage instance may or may not have a memory address, and may or may not be accessible from all threads of execution.

4 **Note 3 to entry:** A storage instance with a memory address occupies a region of zero or more bytes of contiguous data storage in the execution environment.

5 **Note 4 to entry:** One or more objects may be represented within the same storage instance, such as two subobjects within an object of structure type, two **const**-qualified compound literals with identical object representation, or two string literals where one is the terminal character sequence of the other.

3.20

1 **value**

precise meaning of the contents of an object when interpreted as having a specific type

3.20.1

1 **implementation-defined value**

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the **goto** statement (6.8.6.1).

6.2.4 Storage durations and object lifetimes

~~An object has a that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in ??.~~

1 The *lifetime* of an object is the portion of program execution during which ~~storage~~ a storage instance is guaranteed to be reserved for it.[35] An object exists, has a constant address,[36] if any, and retains its last-stored value throughout its lifetime.[37] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

2 ~~An~~ The lifetime of an object is determined by its *storage duration* . There are four storage durations: static, thread, automatic, and allocated. Allocated storage and its duration are described in 7.22.3.

3 The storage instance of an object whose identifier is declared without the storage-class specifier **_Thread_local**, and either with external or internal linkage or with the storage-class specifier **static**, has *static storage duration* ~~. Its~~ , as do storage instances for string literals and some compound literals. The object's lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 ~~An~~ The storage instance of an object whose identifier is declared with the storage-class specifier **_Thread_local** has *thread storage duration*. ~~Its~~ The object's lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There is a distinct ~~object~~ instance of the object and associated storage per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

5 ~~An~~ The storage instance of an object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*, as ~~do~~ are storage instances of temporary objects and some compound literals. The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

6 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object and associated storage is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[38] If the scope is entered recursively, a new instance of the object and associated storage is created each time. The initial value of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to ~~an object~~ a *temporary object* with automatic storage duration and *temporary lifetime*.[39] Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify

[35] This storage instance might not be unique if the object is a string literal, a compound literal or has temporary lifetime.

[36] The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.

[37] In the case of a volatile object, the last store need not be explicit in the program.

[38] Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

[39] The address of such an object is taken implicitly when an array member is accessed.

32 **EXAMPLE 2** The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type **`float`**. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7).

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in this subclause. An object is represented (or held) by a storage instance (or part thereof) that is either created by an allocation (for allocated storage duration), at program startup (for static storage duration), at thread startup (for thread storage duration), or when the lifetime of the object starts (for automatic storage duration). An addressable storage instance[54] of size m provides access to a byte array of length m such that the addresses of all bytes composing the array shall be unique and constant during the lifetime of the storage instance. The address of the first byte of the array is the *start address* of the storage instance, the address one element beyond the array at index m is its *end address* . A storage instance Y *immediately follows* storage instance X if the start address of Y coincides with the end address of X. Other than that, no two distinct and life storage instances shall share a byte address and this document imposes no constraints about a relative ordering of storage instances whenever they are created.[55] The addresses of the bytes of all storage instances of a program execution form its *address space* .

2 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

3 Values stored in unsigned bit-fields and objects of type **`unsigned char`** shall be represented using a pure binary notation.[56]

4 Values stored in non-bit-field objects of any other object type consist of $n \times$ **`CHAR_BIT`** bits, where n is the size of an object of that type, in bytes. ~~The value may be copied into an object of type~~ Converting a pointer of such an object to a pointer to a character type or **`void`** yields a pointer into the byte array of the storage instance such that the values of the first n ~~] (e.g., by **`memcpy`**); the resulting~~ bytes determine the value of the object; the position of the first byte of these in the byte array is the *byte offset* of the object in its storage instance, the converted address is called the *byte address* of the object, and the set of bytes is called the *object representation* of the value. The object representation may be used to copy the value of the object into another object (e.g., by **`memcpy`**). Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the set of m bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

5 Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[57] Such a representation is called a trap representation.

6 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[58]

[54] All storage instances that do not originate from an object definition with **`register`** storage class are addressable by using the pointer value that was returned by their allocation (for allocated storage duration) or by applying the address-of operator & (6.5.3.2) to the object that gave rise to their definition (for other storage durations).

[55] This means that no relative ordering between storage instances and the objects they represent can be deduced from syntactic properties of the program (such as declaration order or order inside a parameter list) or sequencing properties of the execution (such as one instantiation happening before another).

[56] A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.) A byte contains **`CHAR_BIT`** bits, and the values of type **`unsigned char`** range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$.

[57] Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[58] Thus, for example, structure assignment need not copy any padding bits.

The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

7 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

8 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[59] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

9 Loads and stores of objects with atomic types are done with **`memory_order_seq_cst`** semantics.

Forward references: declarations (6.7), expressions (6.5), address and indirection operators (6.5.3.2), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3).

6.2.6.2 Integer types

1 For unsigned integer types other than **`unsigned char`**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified.[60]

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; **`signed char`** shall not have any padding bits. There shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

— the corresponding value with sign bit 0 is negated (*sign and magnitude*);

— the sign bit has the value $-(2^M)$ (*two's complement*);

— the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a *negative zero*.

3 If the implementation supports negative zeros, they shall be generated only by:

— the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with operands that produce such a value;

— the `+`, `-`, `*`, `/`, and `%` operators where one operand is a negative zero and the result is zero;

— compound assignment operators based on the above cases.

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

4 If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with operands that would produce such a value is undefined.

[59] It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts. In particular, if `==` is defined for type `T`, then `x == y` does not imply that `memcmp(&x, &y, sizeof (T))== 0`. Furthermore, `x == y` does not necessarily imply that `x` and `y` have the same value; other operations on values of type `T` might distinguish between them.

[60] Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

6.3.2.3 Pointers

1 A pointer to **void** may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to **void** and back again; the result shall compare equal to the original pointer.

2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

3 An integer constant expression with the value 0, or such an expression cast to type **void** *, is called a *null pointer constant*.[73] If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

4 Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

5 An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.[74]

6 Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

7 A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned[75] for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type or **void**, the result ~~points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes~~ is the byte address of the object.

8 A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

Forward references: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.20.1.4), simple assignment (6.5.16.1).

[73]The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.19.

[74]The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.

[75]In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

char32_t, respectively, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by successive calls to the **mbrtoc16**, or **mbrtoc32** function as appropriate for its type, with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values.[88] If the program attempts to modify such an array, the behavior is undefined.

8 **EXAMPLE 1** This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are `'\x12'` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 **EXAMPLE 2** Each of the sequences of adjacent string literal tokens

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: common definitions `<stddef.h>` (7.19), the **mbstowcs** function (7.22.8.1), Unicode utilities `<uchar.h>` (7.28).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

```
[   ]   (  )   {   }   .    ->
++  --  &   *   +   -   ~   !
/   %   <<  >>  <   >   <=  >=  ==  !=  ^   |   &&  ||
?   :   ::   ;   ...
=   *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,   #   ##
<:   :>  <%  %>  %:   %:%:
```

Semantics

2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an

[88]This allows implementations to share storage instances for string literals and constant compound literals (6.5.2.5) with the same or overlapping representations.

Semantics

3 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.[111)]

4 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.9, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

5 The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.

6 All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.[112)]

7 String literals, and compound literals with const-qualified types, need not designate distinct objects.[113)]

8 **EXAMPLE 1** The file scope definition

```
int *p = (int []){2, 4};
```

initializes p to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

9 **EXAMPLE 2** In contrast, in

```
void f(void)
{
      int *p;
      /*...*/
      p = (int [2]){*p};
      /*...*/
}
```

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by p and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

10 **EXAMPLE 3** Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
      (struct point){.x=3, .y=4});
```

Or, if `drawline` instead expected pointers to **`struct point`**:

```
drawline(&(struct point){.x=1, .y=1},
      &(struct point){.x=3, .y=4});
```

11 **EXAMPLE 4** A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

12 **EXAMPLE 5** The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

[111)]Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or **`void`** only, and the result of a cast expression is not an lvalue.
[112)]For example, subobjects without explicit initializers are initialized to zero.
[113)]This allows implementations to share storage instances for string literals and constant compound literals with the same or overlapping representations.

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

13 **EXAMPLE 6** Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
        (const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage instance is shared.

14 **EXAMPLE 7** Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object `endless_zeros` below:

```
        struct int_list { int car; struct int_list *cdr; };
        struct int_list endless_zeros = {0, &endless_zeros};
        eval(endless_zeros);
```

15 **EXAMPLE 8** Each compound literal creates only a single object in a given scope:

```
        struct s { int i; };

        int f (void)
        {
              struct s *p = 0, *q;
              int j = 0;

        again:
              q = p, p = &((struct s){ j++ });
              if (j  <  2) goto again;

              return p == q && q->i == 1;
        }
```

The function `f()` always returns the value 1.

16 Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

Forward references: type names (6.7.7), initialization (6.7.9).

6.5.3 Unary operators

Syntax

1 *unary-expression:*
  postfix-expression
  ++ *unary-expression*
  -- *unary-expression*
  unary-operator cast-expression
  sizeof *unary-expression*
  sizeof (*type-name* **)**
  _Alignof (*type-name* **)**

unary-operator: one of
  & * + - ~ !

6.5.3.1 Prefix increment and decrement operators

Constraints

1 The operand of the prefix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

6.7 Declarations

Syntax

1 *no-leading-attribute-declaration:*
  declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**
  static_assert-declaration

declaration:
  no-leading-attribute-declaration
  attribute-specifier-sequence declaration-specifiers init-declarator-list **;**
  attribute-declaration

declaration-specifiers:
  declaration-specifier attribute-specifier-sequence$_{\text{opt}}$
  declaration-specifier declaration-specifiers

declaration-specifier:
  storage-class-specifier
  type-specifier-qualifier
  function-specifier

init-declarator-list:
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator:
  declarator
  declarator **=** *initializer*

attribute-declaration:
  attribute-specifier-sequence **;**

Constraints

2 A declaration other than a static_assert or attribute declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;

— tags may be redeclared as specified in 6.7.2.3.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics

5 A declaration specifies the interpretation and properties of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:

— for an object, causes ~~storage~~ a unique storage instance to be reserved for that object;

— for a function, includes the function body;[129]

— for an enumeration constant, is the (only) declaration of the identifier;

— for a typedef name, is the first (or only) declaration of the identifier.

6 The declaration specifiers consist of a sequence of specifiers, followed by an optional attribute specifier sequence, that indicate the linkage, storage duration, and part of the type of the entities that

[129]Function definitions have a different syntax, described in 6.9.1.

the structure or union. If the member declaration list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined. The type is incomplete until immediately after the `}` that terminates the list, and complete thereafter.

11 A member of a structure or union may have any complete object type other than a variably modified type.[133] In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[134] its width is preceded by a colon.

12 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[135] If the value 0 or 1 is stored into a nonzero-width bit-field of type **`_Bool`**, the value of the bit-field shall compare equal to the value stored; a **`_Bool`** bit-field has the semantics of a **`_Bool`**.

13 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

14 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[136] As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

15 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are considered to be members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

16 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

17 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

18 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

19 There may be unnamed padding at the end of a structure or union.

20 As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the ~~object~~ storage instance being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to

[133] A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

[134] The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

[135] As specified in 6.7.2 above, if the actual type specifier used is **`int`** or a typedef-name defined as **`int`**, then it is implementation-defined whether the bit-field is signed or unsigned.

[136] An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

The first declares x to be a pointer to **int**; the second declares y to be an array of **int** of unspecified size (an incomplete type), the storage instance for which is defined elsewhere.

9 **EXAMPLE 3** The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;

void fcompat(void)
{
      int a[n][6][m];
      int (*p)[4][n+1];
      int c[n][n][6][m];
      int (*r)[n][n][n+1];
      p = a;      // invalid:  not compatible because 4 != 6
      r = c;      // compatible, but defined behavior only if
                  // n == 6 and m == n+1
}
```

10 **EXAMPLE 4** All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **_Thread_local**, **static**, or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                           // invalid:  file scope VLA
extern int (*p2)[n];                // invalid:  file scope VM
int B[100];                         // valid:  file scope but not VM

void fvla(int m, int C[m][m]);      // valid:  VLA with prototype scope

void fvla(int m, int C[m][m])       // valid:  adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];        // valid:  block scope typedef VLA

      struct tag {
            int (*y)[n];            // invalid:  y not ordinary identifier
            int z[n];               // invalid:  z not ordinary identifier
      };
      int D[m];                     // valid:  auto VLA
      static int E[m];              // invalid:  static block scope VLA
      extern int F[m];              // invalid:  F has linkage and is VLA
      int (*s)[m];                  // valid:  auto pointer to VLA
      extern int (*r)[m];           // invalid:  r has linkage and points to VLA
      static int (*q)[m] = &B;      // valid:  q is a static block pointer to VLA
}
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.9).

6.7.6.3 Function declarators (including prototypes)

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

2 The only storage-class specifier that shall occur in a parameter declaration is **register**.

3 An identifier list in a function declarator that is not part of a definition of that function shall be empty.

4 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

6.9 External definitions

Syntax

1 *translation-unit:*
  external-declaration
  translation-unit external-declaration

external-declaration:
  function-definition
  declaration

Constraints

2 The storage-class specifiers **`auto`** and **`register`** shall not appear in the declaration specifiers in an external declaration.

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **`sizeof`** or **`_Alignof`** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit.

Semantics

4 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes ~~storage~~ a storage instance to be reserved for an object or provides the body of a function named by the identifier is a definition.

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **`sizeof`** or **`_Alignof`** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[175)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
  attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator*
    declaration-list$_{opt}$ *compound-statement*

declaration-list:
  no-leading-attribute-declaration
  declaration-list no-leading-attribute-declaration

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[176)]

3 The return type of a function shall be **`void`** or a complete object type other than array type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either **`extern`** or **`static`**.

5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **`void`**,

175)Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

in which case there shall not be an identifier. No declaration list shall follow.

6 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations.

Semantics

7 The optional attribute specifier sequence in a function definition appertains to the function.

8 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,[177) the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.6.3 for a parameter type list; the resulting type shall be a complete object type.

9 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

10 Each parameter has automatic storage duration; its identifier is an lvalue.~~The layout of the storage for parameters is unspecified.~~[178)

11 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)

12 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

13 Unless otherwise specified, if the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

14 **EXAMPLE 1** In the following:

```
extern int max(int a, int b)
{
      return a  >  b ? a: b;
}
```

extern is the storage-class specifier and **int** is the type specifier; `max(int a, int b)` is the function declarator; and

[176)The intent is that the type category in a function definition cannot be inherited from a typedef:

```
typedef int F(void);          // type F is "function with no parameters
                              // returning int"
F f, g;                       // f and g both have type compatible with F
F f { /* ... */ }             // WRONG: syntax/constraint error
F g() { /* ... */ }           // WRONG: declares that g returns a function
int f(void) { /* ... */ }     // RIGHT: f has type compatible with F
int g() { /* ... */ }         // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }      // e returns a pointer to a function
F *((e))(void) { /* ... */ }  // same:  parentheses irrelevant
int (*fp)(void);              // fp points to a function that has type F
F *Fp;                        // Fp points to a function that has type F
```

[177)See "future language directions" (6.11.7).

[178)A parameter identifier cannot be redeclared in the function body except in an enclosed block. As any object with automatic storage duration, each parameter gives rise to a unique storage instance representing it. Thus the relative layout of parameters in the address space is unspecified.

2 No other identifiers are reserved. If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), or defines a reserved identifier or attribute token described in 6.7.11.1 as a macro name, the behavior is undefined.

3 If the program removes (with **`#undef`**) any macro definition of an identifier in the first group listed above or attribute token described in 6.7.11.1, the behavior is undefined.

7.1.4 Use of library functions

1 Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow:

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to a non-modifiable storage instance when the corresponding parameter is not const-qualified) or a type (after default argument promotion) not expected by a function with a variable number of arguments, the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

— Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[201] The use of **`#undef`** to remove any macro definition will also ensure that an actual function is referred to.

— Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.[202]

— Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called.[203]

— All object-like macros listed as expanding to integer constant expressions shall additionally be suitable for use in **`#if`** preprocessing directives.

2 Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

3 There is a sequence point immediately before a library function returns.

[201]This means that an implementation is required to provide an actual function for each library function, even if it also provides a macro for that function.

[202]Such macros might not contain the sequence points that the corresponding function calls do.

[203]Because external identifiers and some macro names beginning with an underscore are reserved, implementations can provide special semantics for such names. For example, the identifier `_BUILTIN_abs` could be used to indicate generation of in-line code for the **`abs`** function. Thus, the appropriate header could specify

```
    #define abs(x) _BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.
In this manner, a user desiring to guarantee that a given library function such as **`abs`** will be a genuine function can write

```
    #undef abs
```

whether the implementation's header provides a macro implementation of **`abs`** or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

Description

2 The **longjmp** function restores the environment saved by the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the **setjmp** macro has terminated execution[271] in the interim, or if the invocation of the **setjmp** macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

3 All accessible objects have values, and all other components of the abstract machine[272] have state, as of the time the **longjmp** function was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **setjmp** macro that do not have volatile-qualified type and have been changed between the **setjmp** invocation and **longjmp** call are indeterminate.

Returns

4 After **longjmp** is completed, thread execution continues as if the corresponding invocation of the **setjmp** macro had just returned the value specified by `val`. The **longjmp** function cannot cause the **setjmp** macro to return the value 0; if `val` is 0, the **setjmp** macro returns the value 1.

5 EXAMPLE The **longjmp** function that returns control back to the point of the **setjmp** invocation might cause ~~memory~~ the storage instance associated with a variable length array object to be squandered.

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
      int x[n];          // valid:  f is not terminated
      setjmp(buf);
      g(n);
}

void g(int n)
{
      int a[n];          // a may remain allocated
      h(n);
}

void h(int n)
{
      int b[n];          // b may remain allocated
      longjmp(buf, 2);   // might cause memory loss
}
```

[271]For example, by executing a **return** statement or because another **longjmp** call has caused a transfer to a **setjmp** invocation in a function earlier in the set of nested calls.

[272]This includes, but is not limited to, the floating-point status flags and the state of open files.

Description

2 The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If **rand** is called before any calls to **srand** have been made, the same sequence shall be generated as when **srand** is first called with a seed value of 1.

3 The **srand** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **srand** function.

Returns

4 The **srand** function returns no value.

5 **EXAMPLE** The following functions define a portable implementation of **rand** and **srand**.

```
    static unsigned long int next = 1;

    int rand(void)   //  RAND_MAX assumed to be 32767
    {
          next = next * 1103515245 + 12345;
          return (unsigned int)(next/65536) % 32768;
    }

    void srand(unsigned int seed)
    {
          next = seed;
    }
```

7.22.3 Storage management functions

1 ~~The order and contiguity of storage allocated by successive calls to the~~ If the allocation succeeds, the pointer to a storage instance returned by a call to **aligned_alloc**, **calloc**, **malloc**, ~~and~~ or **realloc** ~~functions is unspecified. The pointer returned if the allocation succeeds~~ is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested. It may then be used to access such an object or an array of such objects in the ~~space~~ storage instance allocated (until the ~~space~~ storage instance is explicitly deallocated). The lifetime of an allocated ~~object~~ storage instance extends from the allocation until the deallocation. Each such allocation shall yield a pointer to ~~an object~~ a storage instance that is disjoint from any other ~~object~~storage instance. The pointer returned points to the start ~~(lowest byte address)~~ address of the allocated ~~space~~storage instance. If the ~~space~~ storage instance cannot be allocated, a null pointer is returned. If the size of the ~~space~~ storage instance requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the ~~behavior is as if the size were some nonzero value, except that~~ address of a storage instance of size zero is returned. For the latter, the returned pointer shall not be used to access an object.

2 For purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only ~~memory locations~~ storage instances accessible through their arguments and not other static duration storage instances. These functions may, however, visibly modify the storage instance that they allocate or deallocate. Calls to these functions that allocate or deallocate storage instances in a particular region of ~~memory~~ the address space shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.[321)]

7.22.3.1 The aligned_alloc function

Synopsis

[321)] This means that an implementation may only reuse a valid address that is computed from an allocated storage instance for a different allocated storage instance if the calls to allocate and deallocate the storage instances synchronize.

1
```
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```

Description

2 The **aligned_alloc** function allocates ~~space for an object~~ a storage instance whose alignment is specified by alignment, whose size is specified by size, and whose ~~value is indeterminate~~byte values are unspecified. If the value of alignment is not a valid alignment supported by the implementation the function shall fail by returning a null pointer.

Returns

3 The **aligned_alloc** function returns either a null pointer or a pointer to the allocated ~~space~~storage instance.

7.22.3.2 The calloc function

Synopsis

1
```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

2 The **calloc** function allocates ~~space~~ a storage instance for an array of nmemb objects, each of whose size is size. The ~~space~~ storage instance is initialized to all bits zero.[322)]

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated ~~space~~storage instance.

7.22.3.3 The free function

Synopsis

1
```
#include <stdlib.h>
void free(void *ptr);
```

Description

2 The **free** function causes the ~~space~~ storage instance pointed to by ptr to be deallocated, that is, made available for further ~~allocation.~~use.[323)] If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a ~~memory~~ storage management function, or if the ~~space~~ storage instance has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

7.22.3.4 The malloc function

Synopsis

1
```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

2 The **malloc** function allocates ~~space for an object~~ a storage instance whose size is specified by size and whose ~~value is indeterminate~~byte values are unspecified.

Returns

3 The **malloc** function returns either a null pointer or a pointer to the allocated ~~space~~storage instance.

322) Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

323) That means that the implementation may reuse the address range of the storage instance (determined by ptr and its size) for any storage instance whose instantiation synchronizes with the call.

7.22.3.5 The `realloc` function

Synopsis

1
```
        #include <stdlib.h>
        void *realloc(void *ptr, size_t size);
```

Description

2 The `realloc` function deallocates the old ~~object~~ storage instance pointed to by `ptr` and returns a pointer to a new ~~object~~ storage instance that has the size specified by `size`. The ~~contents of the new object shall be the same as that~~ bytes of the old ~~object prior to deallocation,~~ storage instance up to the lesser of the new and old sizes ~~.~~ are copied as if by `memcpy` to the initial bytes of the new storage instance. Any bytes in the new ~~object~~ storage instance beyond the size of the old object have ~~indeterminate~~ unspecified values.

3 If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by a ~~memory~~ storage management function, or if the ~~space~~ storage instance has been deallocated by a call to the `free` or `realloc` function, the behavior is undefined. If `size` is nonzero and ~~memory for the new object is not~~ no storage instance is allocated, the old ~~object~~ storage instance is not deallocated. If `size` is zero and ~~memory for the new object is not~~ no storage instance is allocated, it is implementation-defined whether the old ~~object~~ storage instance is deallocated. If the old ~~object~~ storage instance is not deallocated, ~~its value~~ it shall be unchanged.

Returns

4 The `realloc` function returns a pointer to the new ~~object~~ storage instance (which may have the same value as a pointer to the old ~~object~~storage instance), or a null pointer if ~~the new object has not~~ no new storage instance has been allocated.

5 NOTE If a call to `realloc` is successful, the initial part of the new storage instance represents objects with same value and effective type as the initial part of the old storage instance, if any. Nevertheless, the new storage instance has to be considered to be different from the old one:

- Even if both storage instances have the same address, all pointers to the old storage instance (stored within or outside the storage instance) are invalid because that storage instance ceases to exist.
- Copies of objects in the new storage instance that have hidden state and need explicit initialization (such as variable argument lists, atomic objects, mutexes, or condition variables) are in an unspecified state.
- Resources reserved for the original objects in the old storage instance that have hidden state and need destruction (such as variable argument lists, mutexes or condition variables) may be squandered.

7.22.4 Communication with the environment

7.22.4.1 The `abort` function

Synopsis

1
```
        #include <stdlib.h>
        _Noreturn void abort(void);
```

Description

2 The `abort` function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call `raise(SIGABRT)`.

Returns

3 The `abort` function does not return to its caller.

7.22.4.2 The `atexit` function

Synopsis

1

Annex J
(informative)
Portability issues

1 This annex collects some information about portability that appears in this document.

J.1 Unspecified behavior

1 The following are unspecified:

— The manner and timing of static initialization (5.1.2).

— The termination status returned to the hosted environment if the return type of `main` is not compatible with int (5.1.2.2.3).

— The values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).

— The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).

— The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).

— The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).

— The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).

— How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).

— Many aspects of the representations of types (6.2.6).

— The relative order of any two storage instances in the address space (6.2.6.1).

— The value of padding bytes when storing values in structures or unions (6.2.6.1).

— The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).

— The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).

— The values of any padding bits in integer representations (6.2.6.2).

— Whether certain operators can generate negative zeros and whether a negative zero becomes a normal zero when stored in an object (6.2.6.2).

— Whether two string literals result in distinct arrays (6.4.5).

— The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, &&, `||`, `?:`, and comma operators (6.5).

— The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).

— The order of side effects among compound literal initialization list expressions (6.5.2.5).

— The order in which the operands of an assignment operator are evaluated (6.5.16).

— The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).

— Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

— Whether or not a size expression is evaluated when it is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator (6.7.6.2).

— The order in which any side effects occur among the initialization list expressions in an initializer (6.7.9).

— ~~The layout of storage for function parameters (6.9.1).~~ When a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token and the next preprocessing token from the source file is a `(`, and the fully expanded replacement of that macro ends with the name of the first macro and the next preprocessing token from the source file is again a `(`, whether that is considered a nested replacement (6.10.3).

— The order in which **#** and **##** operations are evaluated during macro substitution (6.10.3.2, 6.10.3.3).

— The line number of a preprocessing token, in particular **__LINE__**, that spans multiple physical lines (6.10.4).

— The line number of a preprocessing directive that spans multiple physical lines (6.10.4).

— The line number of a macro invocation that spans multiple physical or logical lines (6.10.4).

— The line number following a directive of the form **#line __LINE__** *new-line* (6.10.4).

— The state of the floating-point status flags when execution passes from a part of the program translated with **FENV_ACCESS** "off" to a part translated with **FENV_ACCESS** "on" (7.6.1).

— The order in which **feraiseexcept** raises floating-point exceptions, except as stated in F.8.6 (7.6.4.3).

— Whether **math_errhandling** is a macro or an identifier with external linkage (7.12).

— The results of the **frexp** functions when the specified value is not a floating-point number (7.12.6.4).

— The numeric result of the **ilogb** functions when the correct value is outside the range of the return type (7.12.6.5, F.10.3.5).

— The result of rounding when the value is out of range (7.12.9.5, 7.12.9.7, F.10.6.5).

— The value stored by the **remquo** functions in the object pointed to by `quo` when `y` is zero (7.12.10.3).

— Whether a comparison macro argument that is represented in a format wider than its semantic type is converted to the semantic type (7.12.17).

— Whether **setjmp** is a macro or an identifier with external linkage (7.13).

— Whether **va_copy** and **va_end** are macros or identifiers with external linkage (7.16.1).

— The hexadecimal digit before the decimal point when a non-normalized floating-point number is printed with an `a` or `A` conversion specifier (7.21.6.1, 7.29.2.1).

— The value of the file position indicator after a successful call to the **ungetc** function for a text stream, or the **ungetwc** function for any stream, until all pushed-back characters are read or discarded (7.21.7.10, 7.29.3.10).

— The details of the value stored by the **fgetpos** function (7.21.9.1).

— The details of the value returned by the **ftell** function for a text stream (7.21.9.4).

— Whether the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, and `wcstold` functions convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (7.22.1.5, 7.29.4.1.1).

— ~~The order and contiguity of storage allocated by successive calls to the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions (??). The amount of storage allocated by a successful~~ If a call to the `calloc`, `malloc`, `realloc`, or `aligned_alloc` function ~~when~~ requesting 0 bytes ~~was requested (??~~fails or returns a storage instance of size zero (7.22.3).

— Whether a call to the `atexit` function that does not happen before the `exit` function is called will succeed (7.22.4.2).

— Whether a call to the `at_quick_exit` function that does not happen before the `quick_exit` function is called will succeed (7.22.4.3).

— Which of two elements that compare as equal is matched by the `bsearch` function (7.22.5.1).

— The order of two elements that compare as equal in an array sorted by the `qsort` function (7.22.5.2).

— The order in which destructors are invoked by `thrd_exit` (7.26.5.5).

— Whether calling `tss_delete` on a key while another thread is executing destructors affects the number of invocations of the destructors associated with the key on that thread (7.26.6.2).

— The encoding of the calendar time returned by the `time` function (7.27.2.4).

— The characters stored by the `strftime` or `wcsftime` function if any of the time values being converted is outside the normal range (7.27.3.5, 7.29.5.1).

— Whether an encoding error occurs if a `wchar_t` value that does not correspond to a member of the extended character set appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by `wcrtomb` (7.29.1).

— The conversion state after an encoding error occurs (7.29.6.3.2, 7.29.6.3.3, 7.29.6.4.1, 7.29.6.4.2,

— The resulting value when the "invalid" floating-point exception is raised during IEC 60559 floating to integer conversion (F.4).

— Whether conversion of non-integer IEC 60559 floating values to integer raises the "inexact" floating-point exception (F.4).

— Whether or when library functions in `<math.h>` raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.10).

— Whether or when library functions in `<math.h>` raise an undeserved "underflow" floating-point exception in an IEC 60559 conformant implementation (F.10).

— The exponent value stored by `frexp` for a NaN or infinity (F.10.3.4).

— The numeric result returned by the `lrint`, `llrint`, `lround`, and `llround` functions if the rounded value is outside the range of the return type (F.10.6.5, F.10.6.7).

— The sign of one part of the `complex` result of several math functions for certain special cases in IEC 60559 compatible implementations (G.6.1.1, G.6.2.2, G.6.2.3, G.6.2.4, G.6.2.5, G.6.2.6, G.6.3.1, G.6.4.2).

— The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than **INT_MAX** (7.21.6.1, 7.29.2.1).

— The number of input items assigned by a formatted input function is greater than **INT_MAX** (7.21.6.2, 7.29.2.2).

— The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.21.6.2, 7.29.2.2).

— A `c`, `s`, or `[` conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is `s` or `[`) (7.21.6.2, 7.29.2.2).

— A `c`, `s`, or `[` conversion specifier with an `l` qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.21.6.2, 7.29.2.2).

— The input item for a `%p` conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.21.6.2, 7.29.2.2).

— The **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, **vsscanf**, **vfwprintf**, **vfwscanf**, **vswprintf**, **vswscanf**, **vwprintf**, or **vwscanf** function is called with an improperly initialized **va_list** argument, or the argument is used (other than in an invocation of **va_end**) after the function returns (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

— The contents of the array supplied in a call to the **fgets** or **fgetws** function are used after a read error occurred (7.21.7.2, 7.29.3.2).

— The file position indicator for a binary stream is used after a call to the **ungetc** function where its value was zero before the call (7.21.7.10).

— The file position indicator for a stream is used after an error occurred during a call to the **fread** or **fwrite** function (7.21.8.1, 7.21.8.2).

— A partial element read by a call to the **fread** function is used (7.21.8.1).

— The **fseek** function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the **ftell** function on a stream associated with the same file or `whence` is not **SEEK_SET** (7.21.9.2).

— The **fsetpos** function is called to set a position that was not returned by a previous successful call to the **fgetpos** function on a stream associated with the same file (7.21.9.3).

— A non-null pointer returned by a call to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** function with a zero requested size is used to access an object (7.22.3).

— The value of a pointer that refers to ~~space~~ a storage instance deallocated by a call to the **free** or **realloc** function is used (7.22.3).

— The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by a ~~memory~~ storage management function, or the ~~space~~ storage instance has been deallocated by a call to **free** or **realloc** (7.22.3.3, 7.22.3.5).

— The value of the object allocated by the **malloc** function is used (7.22.3.4).

— The values of any bytes in a new object allocated by the **realloc** function beyond the size of the old object are used (7.22.3.5).

— The program calls the **exit** or **quick_exit** function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

— The effect if a file with the new name exists prior to a call to the `rename` function (7.21.4.2).

— Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

— Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).

— The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

— The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).

— The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).

— The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.2).

— The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

— The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.22.1.5, 7.29.4.1.1).

— Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.22.1.5, 7.29.4.1.1).

— The meaning of any d-char or d-wchar sequence in a string representing a NaN that is converted by the `strtod32`, `strtod64`, `strtod128`, `wcstod32`, `wcstod64`, or `wcstod128` function (7.22.1.6, 7.29.4.1.2).

— Whether or not the `strtod32`, `strtod64`, `strtod128`, `wcstod32`, `wcstod64`, or `wcstod128` function sets `errno` to `ERANGE` when underflow occurs (7.22.1.6, 7.29.4.1.2).

— Whether the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions return a null pointer or a pointer to ~~an allocated object~~ a storage instance when the size requested is zero (7.22.3).

— Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.22.4.1, 7.22.4.5).

— The termination status returned to the host environment by the `abort`, `exit`, `_Exit`, or `quick_exit` function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

— The value returned by the `system` function when its argument is not a null pointer (7.22.4.8).

— The range and precision of times representable in `clock_t` and `time_t` (7.27).

— The local time zone and Daylight Saving Time (7.27.1).

— The era for the `clock` function (7.27.2.1).

— The `TIME_UTC` epoch (7.27.2.5).

— The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the `"C"` locale (7.27.3.5, 7.29.5.1).

— Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).