ISO/IEC JTC 1/SC 22/WG14 **N 2328**

December 9, 2018 **v 1**

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.

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`/`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", "memory", and (maximal)"object" describing basically all the same thing.
Especially the use of the term "object" is unfortunate and produces a lot of confusion. There is a footnote

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

So it seems implied that all objects have a type. Also objects can have subobjects, e.g the members of a structure object are themselves objects.

2. FIX TERMINOLOGY

We should better make clearer distinctions between terms:

— A data storage facility that is allocated (or instantiated) and that by itself bares no type, is a different concept than the object that is represented by it.
— An object in the abstract state machine is a different concept than the storage (memory, address, space ...) that is used to represent it. An object should always have type, storage (memory, address, space ...) has not. Some objects have no known address (register variables).
— A memory location (address) that is used to identify an object or storage space is something else than the object or storage space itself. This is explicitly mentioned by the standard: over time, the same address can be reused by several "object instances" (automatic storage duration), "space" (allocated storage duration), and will most likely also occur for thread local objects.

We propose to add the term *storage instance*, as being a "*maximal region of data storage*" in the execution environment that is created when either an allocation is encountered or an object instance is created.
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 ensure a 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.

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 C17. *Beware*, that these pages are *not contiguous.*

3.1. Text additions

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

***now 3.19**:.* A definition of the term with two 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 'unsigned char' array. A footnote clarify the absence of any induced positioning between any storage instances.

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 are completely reformulated. 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.
***6.2.6.1, p3**:.* Object representations can now be introduced a bit more clearly.
***6.5, p18**:.* Clarify the extend of a flexible array member.
***7.22.3**:.* Mostly just a consequent use of the new terminology. Clarification that `realloc` does a byte-wise copy of the initial part. (And thus implicitly preserves effective type.)

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**

a 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.

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**

unspecified value where each implementation documents how the choice is made

3.20.2

1 **indeterminate value**

either an unspecified value or a trap representation

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.[33] An object exists, has a constant address,[34] if any, and retains its last-stored value throughout its lifetime.[35] 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.[36] 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*.[37] 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

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

[34]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.

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

[36]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.

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

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 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[51] of size m shall behave as a byte array of type **unsigned char**[m]. No life storage instances shall overlap in any way, and the relative position in memory of different storage instances is unspecified.[52]

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.[53]

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 **unsigned char**$*$ 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; this 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.[54] 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.[55] 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.[56] 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.

[51] All storage instances that do not originate from an object definition with **register** storage class are addressable.

[52] This means that the relative positioning between storage instances and the objects they represent cannot be deduced from syntactical 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).

[53] 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 $2^{\texttt{CHAR\_BIT}} - 1$.

[54] 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.

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

[56] 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.

6.5 Expressions

1 An *expression* is a sequence of operators and operands that specifies computation of a value,[89] or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

2 If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.[90]

3 The grouping of operators and operands is indicated by the syntax.[91] Except as specified later, side effects and value computations of subexpressions are unsequenced.[92]

4 Some operators (the unary operator `~`, and the binary operators `<<`, `>>`, `&`, `^`, and `|`, collectively described as *bitwise operators*) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[93] If a value is stored into an object ~~having no declared type~~ with allocated storage duration through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object ~~having no declared type~~ with allocated storage duration using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object ~~having no declared type~~with allocated storage duration, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[94]

— a type compatible with the effective type of the object,

[89] Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1).

[90] This paragraph renders undefined statement expressions such as

```
    i = ++i + 1;
    a[i++] = i;
```

while allowing

```
    i = i + 1;
    a[i] = i;
```

[91] The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary `+` operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses `()` (6.5.1), subscripting brackets `[]` (6.5.2.1), function-call parentheses `()` (6.5.2.2), and the conditional operator `?:` (6.5.15).

Within each major subclause, the operators have the same precedence. Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

[92] In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations.

[93] ~~Allocated objects have~~ An object with allocated storage duration has no declaration and thus no declared type.

[94] The intent of this list is to specify those circumstances in which an object can or cannot be aliased.

incomplete until immediately after the } that terminates the list, and complete thereafter.

9 A member of a structure or union may have any complete object type other than a variably modified type.[129] 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*;[130] its width is preceded by a colon.

10 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[131] 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**.

11 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.

12 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[132] 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.

13 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.

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

15 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.

16 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.

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

18 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 access that element or to generate a pointer one past it.

[129] 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.

[130] 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.

[131] 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.

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

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 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,[169] 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.

8 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.

9 Each parameter has automatic storage duration; its identifier is an lvalue.[170] ~~The layout of the storage for parameters is unspecified.~~[171]

10 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.)

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

12 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.

13 **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

```
{ return a  >  b ? a: b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

[168] 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
```

[169] See “future language directions” (6.11.7).

[170] A parameter identifier cannot be redeclared in the function body except in an enclosed block.

[171] As any object with automatic storage duration, each parameter gives rise to its own storage instance. Thus the relative layout of parameters in memory is unspecified.

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[257] 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[258] 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
}
```

[257] 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.

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

```
      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 instances allocated by successive calls to the **aligned_alloc**, **calloc**, **malloc**, and **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 then 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 extends from the allocation of the storage instance 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) 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 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 (identified by its address and size) shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.[306)]

7.22.3.1 The aligned_alloc function

Synopsis

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~~ byte values are indeterminate. 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

306)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.

```
        #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.[307]

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.[308]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~~ byte values are indeterminate.

Returns

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

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~~ bytes of the new ~~object shall be the same as that of the old object prior to deallocation,~~ storage instance up to the lesser of the new and old sizes ~~.~~ have the same value as the bytes in the same positions of the old storage instance.[309] Any bytes in the new ~~object~~ storage instance beyond the size of the old object have indeterminate values.

[307]Note that this need not be the same as the representation of floating-point zero or a null pointer constant.
[308]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.
[309]Thus this initial part behaves as if it were copied by `memcpy`. In particular, 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.

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.

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
```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

2 The **atexit** function registers the function pointed to by func, to be called without arguments at normal program termination.[310] It is unspecified whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

4 The **atexit** function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the **at_quick_exit** function (7.22.4.3), the **exit** function (7.22.4.4).

7.22.4.3 The at_quick_exit function

Synopsis

1
```
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
```

[310] The **atexit** function registrations are distinct from the **at_quick_exit** registrations, so applications might need to call both registration functions with the same argument.

— 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 relative layout of storage instances 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 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 (7.6.2.3).

— 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 (7.12.14).

— 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 (7.22.1.3, 7.29.4.1.1).

— 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~~ storage 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 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).

— During the call to a function registered with the **`atexit`** or **`at_quick_exit`** function, a call is made to the **`longjmp`** function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7).

— The string set up by the **`getenv`** or **`strerror`** function is modified by the program (7.22.4.6, 7.24.6.2).

— A signal is raised while the **`quick_exit`** function is executing (7.22.4.7).

— A command is executed through the **`system`** function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

— Whether the last line of a text stream requires a terminating new-line character (7.21.2).

— Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

— The number of null characters that may be appended to data written to a binary stream (7.21.2).

— Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

— Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

— The characteristics of file buffering (7.21.3).

— Whether a zero-length file actually exists (7.21.3).

— The rules for composing valid file names (7.21.3).

— Whether the same file can be simultaneously open multiple times (7.21.3).

— The nature and choice of encodings used for multibyte characters in files (7.21.3).

— The effect of the **remove** function on an open file (7.21.4.1).

— 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 (7.22.1.3, 7.29.4.1.1).

— Whether or not the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, or **wcstold** function sets **errno** to **ERANGE** when underflow occurs (7.22.1.3, 7.29.4.1.1).

— Whether the **calloc**, **malloc**, **realloc**, and **aligned_alloc** functions return a null pointer or a pointer to ~~an allocated object~~ storage 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).