Basic lambdas for C

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14
WG 21, SG 22

N2892 v5
P2303R4

2021-12-25

Basic lambdas for C
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

We propose the inclusion of simple lambda expressions into the C standard. We build on a slightly restricted syntax of that feature in C++. In particular, they only have immutable value captures, fully specified parameter types, and, based on N2891, the return type is inferred from **return** statements. This is part of a series of papers for the improvement of type-generic programming in C for which the rationale is given in N2890. Follow-up papers N2894 and N2893 extend this feature with **auto** parameter types and default capture strategies, respectively.

Changes:

v5/R4. this document
— integrate "lvalue captures" from N2737 into the base proposal and name them "identifier captures".
— remove default captures and shortcuts from this proposal (moved to N2893)
— remove option for grammar disambiguation (not relevant without shortcuts)
— Add a brief discussion about prior art.
— make access of register variables as identifier captures undefined
— make conversion of closures to function pointers undefined (accommodates possible introduction of wide function pointers, see N2862)
— attach attributes to the lambda type (and derived prototype) instead of the lambda value
— remove the constraints for indirect captures
v4/R3.
— Add a newly found ambiguity of the grammar to *Caveats* (IV) concerning array element designators.
— add a rule for consecutive `[ [` to avoid lexical conflicts with attributes
— a minor grammar change for function literals
— also insist that outer objects must be accessible (and not only visible) to be possible implicit captures
— add two notes to provide models for direct execution of lambda expressions
— make primary expressions transparent for lambda expression operands
— insist that a lambda type and its visibility depends on the lexical position of a lambda expression in the program
— use the correct terminology of blocks instead of scope in some places
v3/R2. integrating feedback from different sources
— Add a section *Caveats* (IV) that describes possible implementation difficulties.
— Wording changes:
  — add two notes in the concepts clause that relate the terms of scope and linkage to lambda expressions and captures
  — make lambda values copyable by assignment
  — better describe how a converted function literal would be specified as a static function
  — only require that converted-to function pointers are compatible
  — **setjmp** and lambdas, second take
v2/R1. integrating feedback from the WG14 reflector
— add function literals to the RHS of assignment and cast if the target type is a function pointer
— make it clear that lambda objects can only be formed by **auto** definitions
— cleanup of the relationship between lambdas and VM types
— be more precise on the sequencing of lambda evaluations and function calls
— affect the attributes of a lambda expression to the lambda value
— integrate `<stdarg.h>` and lambdas
— integrate `<setjmp.h>` and lambdas
— integrate lambdas with the rest of the library clause

I. MOTIVATION

In N2890 it is argued that the features presented in this paper are useful in a more general context, namely for the improvement of type-generic programming in C. First, we will try to motivate the introduction of lambdas as a stand-alone addition to C (I.1) and then show the type-generic features that can already be accomplished with the combination of lambdas and type inference (**auto** and **typeof**) (I.3).

When programming in C we are often confronted with the need of specifying small functional units that

— are to be reused in several places
— are to be passed as argument to another function
— need a fine control of data in- and outflow.

I.1. Lambdas for the specification of reusable blobs of code

Usual functional macros have several shortcomings that usually make life for programmers relatively difficult:

(1) Arguments may be evaluated several times, even for side effects.
(2) The sequencing property of a macro call may be different than for a function call. In particular, the evaluation of arguments is not clearly sequenced before the evaluation of the body.
(3) There can be uncontrolled interaction between the surrounding scope of a macro call and the executed body of the macro.

In their simplest form lambdas already help to overcome these.

```
1  #define minDouble(X, Y)                                  \
2  [](double x, double y) { return (x < y) ? x : y; } \
3    ((X),                                                  \
4     (Y))
```

In this example, the subexpressions of the macro body, one per source line, are all executed unsequenced before the actual function call. So if we call the macro with expressions that contain identifiers x and y their use is not mixed up with the parameter names of the lambda.

```
1  double x = 9.0;
2  double y = 1.5;
3  double z = minDouble(x+y, x-y);
```

If we want to be more generic and not depend on the type of the parameters, we can use captures for a general strategy to freeze the argument values of a macro and apply possible side effects exactly once.

```
1  #define min(X, Y)                                        \
2  /* capture clause */                                     \
3  [xMin = (X),                                            \
4   yMin = (Y)]                                            \
5   /* empty parameter list */                             \
6               (void) {                                   \
```

```
7       /* start of function body */              \
8       return (xMin < yMin) ? xMin : yMin;       \
9     }                                           \
10    /* end of lambda expression */              \
11    /* function call */                         \
12    ()
```

Here the assignment expressions for the captures `xMin` and `yMin` are the first expressions that are evaluated one after the other when such a lambda expression is met. Then this parameterless lambda is called directly with the empty `()` in the last line and the result value is determined.

Unfortunately, this technique is not a complete solution to problem (3) above. To ensure the independence of the evaluations of the macro arguments we need capture names that are unique. But if we are able to ensure that, they get not mixed up with other identifiers that might be part of the macro arguments when that is called, and the evaluation of the return expression of the lambda is then independent of these and has no additional side effects. For a discussion of more general type-generic lambdas see below (I.3) and N2894.

I.2. Function literals and function pointers

Function pointer are an important tool in C to apply generic functionality (such as sorting) to a variety of contexts. When for example sorting strings we might be interested in different sort orders, for example depending on the treated language.

The smallest unit currently is the specification of a function, that is a top-level named entity with identified parameters for input and output. Current `C` provides several mechanisms to ease the specification of such small functions:

— The possibility to distinguish internal and external linkage via a specification with **`static`** (or not).
— The possibility to add function definitions to header files and thus to make the definitions and not only the interface declaration available across translation units via the **`inline`** mechanism.
— The possibility to add additional properties to functions via the attribute mechanism.

All these mechanisms are relatively rigid:

(1) They require a naming convention for the function.
(2) They require a specification far away and ahead of the first use.
(3) They treat all information that is passed from the caller to the function as equally important.

As an example, take the task of specifying a comparison function for strings to **`qsort`**. There is already such a function, **`strcmp`**, in the `C` library that is almost fit for the task, only that its prototype is missing an indirection. The semantically correct comparison function could look something like this:

```
1   int strComp(char* const* a, char* const* b) {
2     return strcmp(*a, *b);
3   }
```

Although probably for most existing ABI its call interface could be used as such (**char*** **const*** and **void const*** have the same representation) the use of it in the following call is a constraint violation:

```
#define NUMEL 256
char* stringArray[NUMEL] = { "hei", "you", ... };

  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        strComp); // mismatch, constraint violation
  ...
```

The reflex of some C programmers will perhaps be to paint over this by using a cast:

```
  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        (int(*)(void const*, void const*))strComp); // UB
  ...
```

This does not only make the code barely readable, but also just introduces undefined behavior instead of a constraint violation. On the other hand, on many platforms the behavior of this code may indeed be well defined, because finally the ABI of `strComp` is the right one. Unfortunately there is no way for the programmer to know that for all possible target platforms.

So the "official" strategy in C is to invent yet another wrapper:

```
int strCompV(void const* a, void const* b) {
  return strComp(a, b);
}

  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        strCompV); // OK
  ...
```

This strategy has the disadvantages (1) and (2), but on most platforms it will also miss optimization opportunities:

— Since `strCompV` is specified as a function its address must be unique. The caller cannot inspect **qsort**, it cannot know if `strCompV` and `strComp` must have different addresses. Thus we are forcing the creation of a function that only consists of code duplication.
— If the two functions are found in two different translation units, `strCompV` will just consist of a tail call to `strComp` and thus create a useless indirection for every call within **qsort**.

C++'s lambda feature that we propose to integrate into C allows the following simple specification:

```
  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        [](void const* a, void const* b){
          return strComp(a, b);
```

```
5         });
6   ...
```

By such a specification of a lambda we do not only avoid (1) and (2), but we also leave it to the discretion of the implementation if this produces the a new function with a different address or if the tail call is optimized at the call site and the address of `strComp` is used instead.

Altogether, the improvements that we want to gain with this feature are:

— Similar to compound literals, avoid useless naming conventions for functions with a local scope (anonymous functions).
— Avoid to declare and define small functions far from their use.
— Allow the compiler to reuse functions that have the same functionality and ABI.
— Split interface specifications for such small functions into an invariant part (captures) and into a variable part (parameters).
— Strictly control the in- and outflow of data into specific functional units.
— Provide more optimization opportunities to the compiler, for example better tail call elimination or JIT compilation of code snippets for fixed run-time values.

I.3. Type-generic features

WG14 has already voted favorable to introduce the **typeof** and the **auto** features for type inference. Adding lambdas to this mix already forms a powerful toolset for type-generic programming. A type-generic sort macro for real types and pointer types already shows many of the possibilities:

```
1  #define SORT(X, N)                                                    \
2    [_Cap1 = &((X)[0]), _Cap2 = (N)](void) { /* fix arguments */ \
3      auto start = _Cap1;                /* claim desired name */ \
4      auto numel = _Cap2;                /* claim desired name */ \
5      typedef typeof(start[0]) base;           /* deduce type */ \
6      int (*comp)(void const*, void const*)                          \
7          = [](void const*restrict a, void const*restrict b){       \
8                  base A = *(base const*){ a };                      \
9                  base B = *(base const*){ b };                      \
10                 return (A < B) ? -1 : ((B < A) ? +1 : 0);          \
11         };                                                         \
12     qsort(start, numel, sizeof(base), comp);                       \
13   } ()
```

— The evaluation of the macro parameters in the capture of the outer lambda guarantees that they are evaluated at most once.
— Their type is inferred as it would for an **auto const** variable.
— Actual **auto** declarations provide variables with types and names as desired.
— Using an outer capture without default (see N2893), guarantees that the body of the capture will not use any local variable of the calling context in an unexpected way.
— Using **typeof** on the first element ensures that the inferred type has the correct qualification and size.

— The inner lambda is converted to a function pointer, over which the implementation has full control: they may synthesize it newly or reuse another one (with same representation for `base`) as long as the observable behavior is the same.
— No pollution of the global scope (or even the linker) with a name for a function that would only be used once.
— The inner lambda only uses static type information from outer scopes, namely the type `base`, a local type of the outer lambda.
— The correctness of the conversions to `void*` from `base*` and back to `base const*` from `void const*` can be checked easily.
— The predefined `<` operator of type `base` is used for comparison. If `base` is not a real or pointer type, the compilation fails.
— Once all required information is collected the sorting itself uses the standard `qsort` facility, but now with a type safe encapsulation.

Two disadvantages of this approach remain:

— To avoid name clashes with the argument expressions of the macro, generic capture names have to be invented and the desired application names can then only be claimed in a second step.
— No specialization of a pointer to sort function can be easily generated.

These issues will be addressed in the follow-up paper N2894.

II. PRIOR ART

N2890 also shows that there is a lot of prior art in this domain that covers a substantial part of the market. Nevertheless, there is no single solution that has emerged that would dominate, and so if such a feature is wanted for C, WG14 would have to decide which way to follow. WG14 already expressed a preference for C++'s lambdas instead of gcc's nested functions. We propose here a version of lambdas that provides most of the functionality of existing approaches; in particular it covers the possibility to capture values (as by default done by Objective C's block extension) and to capture identifiers (as by gcc's nested functions and compound expressions).

Because any of these forms of captures provoke strong allergic reactions in parts of the community, for this paper we only propose *explicit* captures, that is, that either explicitly evaluate an expression (in the form `id = expr`) or explicitly list an identifier that extends into the scope of the lambda (in the form `&id`).

By this strategy we intend to propose migration paths for user code that uses one or another form of these features, without introducing implicit evaluations (by abbreviating `id = id` to `id`) or default captures (provided by `=` for default value captures and by `&` for identifier captures).

III. DESIGN CHOICES

III.1. Expression versus function definition

Currently, the `C` standard imposes to use named callbacks for small functional units that would be used by `C` library functions such as `atexit` or `qsort`. Where inventing a name is already an unnecessary burden to the programming of small one-use functionalities, the distance between definition and use is a real design problem and can make it difficult to enforce consistency between a callback and a call. Already for the `C` library itself this is a

real problem, because function arguments are even reinterpreted (transiting through **void const***) by a callback to **qsort**, for example. The situation is even worse, if input data for the function is only implicitly provided by access to global variables as for **atexit**.

Nested functions improve that situation only marginally: definition and use are still dissociated, and access to variables from surrounding scopes can still be used within the local function. In many cases the situation can even be worse than for normal functions, because variables from outside that are accessed by nested functions may have automatic storage duration. Thus, nested functions may access objects that are already dead when they are called, making the behavior of the execution undefined.

For these reasons we opted for an expression syntax referred to as *lambda*. This particular choice not withstanding we think that it should still be *possible* to name a local functionality if need be, and to reuse it in several places of the same program. Therefore, lambdas still allow to manipulate *lambda values*, the results of a lambda expresssion, and in particular that these values are assigned to objects of lambda type.

III.2. Capture model

For the possible visibility of types and objects inside the body of a lambda, the simplest is to apply the existing scope model. This is what is chosen here (consistently with C++) for all use of types and objects that do not need an evaluation.

— All visible types can be used, if otherwise permitted, as type name in within **alignof**, **alignas**, **sizeof** or **typeof** expressions, type definitions, generic choice expressions, casts or compound literals, as long as they do not lead to an evaluation of a variably modified type.
— All visible objects can be used within the controlling expression of **_Generic**, within **alignof** expressions, and, if they do not have a variably modified type, within **sizeof** or **typeof** expressions.

In contrast to that and as we have discussed in N2890, there are four possible design choices for the *access* of automatic variables that are visible at the point of the evaluation of a lambda expression. We don't think that there is any "natural" choice among these, but that for a given lambda the choice has to depend on several criteria, some of which are general (such as personal preferences or coding styles) and some of which are special (such as a subsequent modification of the object or optimization questions).

As a consequence, we favor a solution that leaves the principal decision if a capture is a value capture or an identifier capture to the programmer of the lambda; it is only they who can appreciate the different criteria. We think that the choice of explicit specification of value captures as provided by C++ lambdas is preferable to the implicit use of value captures for all automatic variables as in Objective C's blocks, or of identifier captures as for gcc's compound expression or nested functions.[1]

III.3. Call sequence

As for all papers in this series, we intend not to impose ABI changes to implementations. We chose a specification for a call sequence for lambdas that either uses an existing function call ABI or encapsulates all calls to lambdas within a given translation unit.

[1] These different possibilities have been discussed in N2890.

For function literals, that is lambdas that have no captures, we impose that they should be convertible to function pointers with a compatible prototype. Such a lambda can be rewritten to a static function with an auxiliary name which then is used in place of the lambda expression itself.

For closures, that is lambdas with captures, the situation is a bit more complicated. Where some implementations, building for example upon `gcc`'s nested functions, may prefer to use the same calling sequence as for functions, others may want to evaluate captures directly in place and use an extended ABI to call a derived function interface or pass information for the captures implicitly in a special register.

Therefore, our proposal just adds lambda values to the possibilities of the postfix expression (LHS) of a function call, and imposes no further restrictions how this feature is to be implemented.

III.4. Interoperability

The case that objects with lambda type can be defined and may have external linkage, could imply that such lambda objects are made visible between different translation units. If that would be possible, implementations would be forced to extend ABIs with the specification of lambda types, and platforms that have several interoperable implementations would have to agree on such ABI.

To require such an ABI specifiction would have several disadvantages:

— A cross-implementation effort for an ABI extension would incur a certain burden for implementations.
— Many different ABI are possible, in particular special cases have a certain potential for optimization. Fixing an ABI too early, forces implementations to give stability guarantees for the interface.

For our proposal here, we expect that most lambda expressions that appear in file scope will be function literals. Since function literals can be converted to function pointers, no special syntax is needed to make their functionalities available to other translation units.

Because there are no objects with automatic storage duration in file scope, the only captures that can be formed in file scope are those that are derived from expressions, and these expression must have a value that can be determined at translation time. We think that it should be possible to define most such captures as lambda-local unmutable objects with static storage duration, and thus, in general such lambdas are better formulated as function literals.

To be accessible in another translation unit a closure expression that is evaluated in block scope, would have to be assigned to a global variable of lambda type. We inhibit this by not specifying a declaration syntax for lambdas. Thereby the only possibility to declare an object of lambda type is to use **`auto`**, and thus each such declaration must also be a definition such that the full specification of the lambda expression is visible. But then, no translation unit can declare an object of lambda value with external linkage that is not already a definition.

Note, that N2862 independently makes a proposal that introduces wide functions and pointers to them. This establishes an ABI that could be used for calling lambdas across different translation units.

III.5. Invariability

Since lambdas will often concern small functional units, our intent is that implementations use all the means available to optimize them, as long as the security of the execution can be guaranteed. Therefore we will enforce that lambda values, once they are stored in an object, will be known to never change. This will inhibit, e.g, that implementation specific functions or jump targets will change between calls to the same lambda value, or that any lambda value can escape to a context where its originating lambda expression is not known.

III.6. Recursion

Since there is no syntax to forward-declare a lambda and they can only be assigned to a lambda value that stems from the same lambda expression, a lambda cannot refer to itself (same lambda value and type), neither directly nor indirectly by calling other functions or lambdas. The only possibility is for function literals, when they are converted and assigned to function pointers. Such a function pointer can then be used directly or indirectly as any other function pointer, also by the function literal expression that gave rise to its conversion.

```
1  // file scope definition
2  static int (*comp)(void const*, void const*) = 0;
3  ...
4  int main(void) {
5    ...
6    comp = [](void const* A, void const* B){
7      if (something) {
8        return 0;
9      } else {
10       return comp(B, A);
11     }
12   }
13   ...
14 }
```

Such examples for function literals are a bit contrived, and will probably not be very common.

In contrast to that, closures cannot be called recursively because they don't even convert to function pointers. This is a conscious decision for this paper, because we don't want to constrain implementations in the way(s) they reserve the storage that is necessary to hold captures, and how they implement captures in general. For example, closures that return **void** can be implement relatively simple as-if by adding some small state, an entry label, one return label per call, and some switched or computed **goto** statements.

As a consequence, the maximum storage that is needed for the captures of a given closure can be computed at translation time, and no additional mechanism to handle dynamic storage is necessary.

III.7. Variable argument lists

Although permitted, lambdas with variable argument list are not completely implemented by the major C++ compilers. This seems to indicate that there is not much need for them, and to simplify we have left them out of this specification. If need be, they could be added

later with a separate paper by using *parameter-type-list* instead of *parameter-list* in the definition of *parameter-clause*

This not withstanding, lambdas may have parameters of type **`va_list`** (`stdarg.h`). This can be useful for small functional units that process variable argument lists of functions.

III.8. Variably modified (VM) types

All VM types, not only VLA, have a hidden state that keeps track of the size or sizes of the current object or the object it points to. Even if such objects may have static storage duration (see e.g 6.7.6.2 p10), their state may have automatic storage duration, and so their use from a lambda is not easily modeled. Therefore the use of an outer object with VM type is completely forbidden with the body of a lambda.

III.9. Lexical ambiguities

The new grammar as proposed has two new lexical ambiguities, see also Section IV.1, below.

— A start sequence `[` *identifier* `=` *expression* `]` may be an array element designator idexed by an assignment expression (`6.7.12 p1`) or a capture clause (`6.5.2.6 p1`).
— A start sequence `[ [` *identifier* may be the start of an attribute specifier (`6.7.15.1` p1) or the start of several other constructs that allow an expression within an array bound (`6.7.8.2 p1` and `6.7.9 p1`) or array subscript (`6.5.2 p1`).

The first may be resolved immediately after a token sequence as indicated above has been scanned. It only requires limited lookahead for resolution, although the occurrence of commas may complicate parsing. We don't think this ambiguity needs otherwise to be resolved normatively. WG14 could add a rule that gives priority to the designator reading and force lambda expressions that are used in initializers to be surrounded by parenthesis, but this should be proposed in a different paper.

In this basic version as presented here, the second ambiguity only needs bounded lookahead, because for an attribute the next character after *identifier* can only be an opening parenthesis (for the attribute argument), a closing bracket (for the termination of the attribute) or a comma (for a subsequent attribute name). All of these are not valid for lambdas as presented here.

This becomes more complicated if we allow identifier shortcuts for value captures as are proposed in N2893. Therefore, that paper then also proposes an option for syntax ambiguation.

IV. CAVEATS FOR IMPLEMENTORS

IV.1. Syntax

While at the time of their introduction into C++ the lambda feature caused no syntax ambiguity, unfortunately C's VLA and C++' **`constexpr`** evaluations of array sizes now make the construct ambiguous in both languages.

With C17 the lambda feature interacts in a subtle way concerning array element designators in initializers. The problem is that in C an initializer of an array element can be an expression or a designator. With lambdas we have expressions that start with a `[` and so this can be taken for either the start of an initializer or a designator. E.g `[something` could be valid for both and only the next token would decide how this is to be interpreted. If it is a `=` it must

start a capture list (designators don't have assignment expressions), otherwise it must be a designator.

Additionally, lambdas as presented here add an interaction between the attribute syntax and array declarations, see above. Since for this proposal here we don't have identifier shortcuts for value captures, so far this only adds bounded lookahead.

IV.2. Visibility of non-captures

The lambda concept allows to refer to outer identifiers even if they are not captured, as long as they are not evaluated. This can for example be the case for local type definitions (**typedef**) or the use of variable names in **typeof** or **sizeof** expression. This specification takes care that none of these identifiers have VM types, and so all their accessible features are known at translation time. Nevertheless, textually lifting a lambda and all the features of which it depends outside of its defining function may be challenging even for a function literal.

V. SYNTAX AND TERMINOLOGY

For all proposed wording see Section X.

V.1. Lambda expressions

Since it is the most flexible and expressive, we propose to adopt C++ syntax for lambdas, 6.5.2.6 p1, as a new form of postfix expression (6.5.2 p1) introducing the terms *lambda expression*, *capture clause*, *capture list*, *capture list element*, *value capture*, *identifier capture*, *capture* and *parameter clause*.

We make some exceptions from that C++ syntax for the scope of this paper:

(1) We omit the possibility to specify the return type of a lambda. The corresponding C++ syntax

```
-> return-type
```

reuses the **->** token in an unexpected way, and is not strictly necessary if we have **auto** return. Additionally, even in C++ the implications on visibility of the return type and evaluation order are not yet completely settled for this construct. If WG14 wishes so, this feature could be added in the future as a general function return type syntax.

(2) We omit the possibility to specify all value captures as mutable. The C++ syntax introduces a keyword, mutable, that would be new to C. We don't see enough gain that would justify the introduction of a new keyword.

(3) For the simplicity of this proposal we omit default captures and shortcuts for value captures. A follow-up paper, N2893, takes care of these. The introduction of lvalue aliases (C++'s references) is not currently planned.

(4) We omit the possibility for the parameter list to end in a ... token.

As this syntax leaves the parameter clause as optional, 6.5.2.6 p7 fixes the semantics for this case to be equivalent to an empty parameter list, and also introduces the terminology of *function literal* (no captures) and *closure* (any capture).

The terminology for *lambda values* and *lambda types* and their *prototype* is introduced with the other type categories in 6.2.5 p20, and then later specified in the clause for lambda expressions, 6.5.2.6 p11.

V.2. Adjustments to other constructs

With the introduction of lambda expressions, functions bodies can now be nested and several standard constructs become ambiguous. Therefore it is necessary to adjust the definitions of these constructs and relate them to the nearest other constructs to which they could refer. This ensures that their use remains unique and well defined, and that no jumps across boundaries of function bodies are introduced.

— For labels we enforce that they are anchored within the nearest function body in which they appear:
  — Function scope as the scope for labels must only extend to the innermost function body in which a label is found and such function scopes are *not* nested (`6.2.1 p3`).
  — Case labels must be found within a corresponding **`switch`** statement of their innermost function body (`6.8.1 p2`).

— **`continue`** and **`break`** statements must match to a loop or **`switch`** construct that is found in the innermost function body that contains them (`6.8.6.2 p1` and `6.8.6.3 p1`).

— A **`return`** statement also has to be associated to the innermost function body. It has to return a value, if any, according to the type of that function body. Also, if its function body is associated to a lambda, it only has to terminate the corresponding call to the lambda, and not the surrounding function (`6.8.6.4 p3`).

— We allow function literals to be operand of simple assignment (`6.5.16.1 p1`) and cast (`6.5.4 p2`) when the target type is a function pointer.

Another property is to know how a lambda expressions integrate into the semantics categories that are induced by the grammar. Similar as for other categories (for example being an lvalue expression) we emphasize that primary expressions transmit the category to be a lambda expression, `6.5.1 p5` and `6.5.1.1 p4`.

VI. SEMANTICS

The principal semantic of lambda expressions themselves is described in `6.2.5.6 p6`. Namely, it describes how lambda expressions are similar to functions concerning the scope of visibility and the lifetime of captures and parameters.

Captures are handled in three paragraphs. The main feature is the description of the evaluation that provides values for value captures, `6.5.2.6 p7`. It stipulates that their values are determined at the point of evaluation of the lambda expression (basically in order of declaration), that the value undergoes lvalue, array-to-pointer or function-to-pointer conversion if necessary, and that the type of the capture then is the type of the expression *after* that conversion, that is without any qualification or atomic derivation, and, that it gains a **`const`** qualification. Additionally, we insist that the so-determined value of a value capture will and cannot not change by any means and is the same during all evaluations during all calls to the same lambda value. Paragraph `6.5.2.6 p8` then describes the order of evaluations and sequencing properties of value captures.

Paragraph `6.5.2.6 p9` specifies the semantics of identifier captures, namely that they follow the usual lexical rules for access to variables, that it is undefined if they may be **`register`**, and that modifications to them are synchronized via the happens-before relation.

The other specifications for lambda expressions are then their use in different contexts.

— Function literals may be converted to function pointers, 6.3.2.1 p5. For these this is easily possible because they have exactly the same functionality as functions: all additional caller information is transferred by arguments to the call. Thus the existing function ABI can be used to call a function literal, and the translator has in fact all information to provide such a call interface.
— As postfix expression within function calls they can take the place that previously only had function pointers, 6.5.2.2. If we would not provide the possibility of captures, the corresponding function literals could all first be converted to function literals (see above) and called then. But since we don't want to impose how lambda-specific capture information is transferred during a call and to guarantee the properties specified in III.3 above, we just add lambdas to the possibilities of the postfix expression that describes the called function.[2]

VII. LIBRARY

The impact on the library clause is relatively small. It mostly concerns an update for the terminology, because the calling context may be a function or a lambda and a callback feature that is referred by a function pointer may indicate an ordinary function or a function literal. Such rectifications concern <setjmp.h>, <signal.h>, <stdarg.h>, <stdlib.h> and <thread.h>. The impacted library functions or macros are

_Exit	**call_once**	**quick_exit**	**va_end**
at_quick_exit	**exit**	**signal**	
atexit	**longjmp**	**thrd_create**	
bsearch	**qsort**	**tss_create**	

VIII. CONSTRAINTS AND REQUIREMENTS

— Identifier captures are introduced to handle objects of automatic storage duration, all other categories of objects and functions are to use other mechanisms of access within lambdas. Therefore, we constrain identifier captures to names of objects of automatic storage duration (6.5.2.6 p2) and limit the evaluation of other such objects from a surrounding scope to the initialization of value captures (6.5.2.6 p3). All such evaluations thus take place during the evaluation of the lambda expression itself, not during a subsequent call to the lambda value.
— Unfortunately such a restriction for objects of automatic storage duration is not sufficient to avoid the implicit access of hidden dynamic state from within a lambda. The reason is that there are some rare forms of objects of VM type that have static storage duration, for which even the use in **sizeof** or similar constructs would constitute an evaluation. These are just exotic artifacts in the language without much use cases or justification. We just forbid them by a constraint for this proposal (also 6.5.2.6 p3), but they could be added in later a stage if need be.
— Calling a closure needs additional information, namely the transfer of per-instance specific values for value captures and context information for identifier captures. Converting closures to function pointers is not defined by the text, so doing so in a program is implicitly undefined. This UB allows implementations to extend this feature and allow such conversions for some or all closure types. For example implementations that use gcc's nested functions could continue to allow a conversion of such function names to function pointers.
— A **switch** label should not enable control flow that jumps from the controlling expression of the **switch** into a lambda. The corresponding property is syntactic and can be checked at translation time. Therefore we formulate this as a constraint in 6.8.1 p2.

[2] A similar addition for function designators could also be made, see [Gustedt 2016].

— Labels should not be used to bypass the calling sequence (capture and parameter instantiation) and jump into a lambda. Therefore we constrain the visibility scope of labels to the surrounding function body, `6.2.1 p3`. With these constraints, no **`goto`** statement can be formed that jumps into or out of a lambda or into a different function.
— Similarly, all jump statements other than **`return`** should never attempt to jump into or out of the nearest enclosing function body. To ensure this we add an explicit constraint as `6.8.6 p2`, and in `6.8.6.2 p1` and `6.8.6.3 p1`.
— According to III.5 we don't want lambda values to be modified. If they were specified from scratch, this would probably be reflected in both, a constraint and a requirement. But since we want to be able to leave the possibility that lambda values are implemented as function pointers (in particular for function literals) we cannot make this a requirement. Therefore, we only introduce a requirement (`6.5.2.6 p10` last sentence) and recommended practice for applications to use a **`const`** qualification and for implementations to diagnose modifications when possible (`6.5.2.6 p11`).
— There is no direct syntax to declare lambda types, and so objects of lambda type can only be declared (and defined) through type inference. The necessary adjustments to that feature are integrated to the constraints of `6.7.11 p3` and `p4`.

IX. QUESTION FOR WG14

In the March 2021 session, WG14 has already voted in favor of integrating the lambda feature into C23 along the lines as described here.

Does WG14 want to integrate the changes as specified in N2892 into C23?

Ackowledgments

Many thanks go to and to many other WG14ners for the discussions, especially to Joseph Myers for his very detailed review and feedback.

References

Jens Gustedt. 2016. *The register overhaul*. Technical Report N2067. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2067.pdf.

Jens Gustedt. 2021a. *Function literals and value closures*. Technical Report N2892. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2892.pdf.

Jens Gustedt. 2021b. *Improve type generic programming*. Technical Report N2890. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2890.pdf.

Jens Gustedt. 2021c. *Lvalue closures*. Technical Report N2737. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2737.pdf.

Jens Gustedt. 2021d. *Type-generic lambdas*. Technical Report N2894. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2894.pdf.

Jens Gustedt. 2021e. *Type inference for variable definitions and function return*. Technical Report N2891. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2891.pdf.

X. PROPOSED WORDING

The proposed text is given as diff against N2891.

— Additions to the text are marked as shown.
— Deletions of text are marked as ~~shown~~.

6. Language

6.1 Notation

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that

$$\{ \ expression_{opt} \ \}$$

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

3 A summary of the language syntax is given in Annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

1 An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)

3 A label name is the only kind of identifier that has *function scope*. It can be used (in a **goto** statement) ~~anywhere~~ in the function body in which it appears, and is declared implicitly by its syntactic appearance (followed by a : and a statement). Each function body has a function scope that is separate from the function scope of any other function body. In particular, a label is visible in exactly one function scope (the innermost function body in which it appears) and distinct function bodies may use the same identifier to designate different labels.[29)]

4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator.[30)] If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

[29)] As a consequence, it is not possible to specify a **goto** statement that jumps into or out of a lambda or into another function.

[30)] Identifiers that are defined in the parameter list of a lambda expression do not have prototype scope, but a scope that comprises the whole body of the lambda.

5 Unless explicitly stated otherwise, where this document uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

6 Two identifiers have the *same scope* if and only if their scopes terminate at the same point.

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. An identifier that has an underspecified definition and that designates an object, has a scope that starts at the end of its initializer and from that point extends to the whole translation unit (for file scope identifiers) or to the whole block (for block scope identifiers); if the same identifier declares another entity with a scope that encloses the current block, that declaration is hidden as soon as the inner declarator is met.[31)] An identifier that designates a function with an underspecified definition has a scope that starts after the lexically first **`return`** statement in its function body or at the end of the function body if there is no such **`return`**, and from that point extends to the whole translation unit. Any other identifier has scope that begins just after the completion of its declarator.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

9 **NOTE** Properties of the feature to which an identifier refers are not necessarily uniformly available within its whole scope of visibility. Examples are identifiers of objects or functions with an incomplete type that is only completed in a subscope of its visibility, labels that are only valid targets of **`goto`** statements when the jump does not cross the scope of a VLA, identifiers of objects to which the access is restricted in specific contexts such as signal handlers or lambda expressions, or library features such as **`setjmp`** where the use is restricted to a specific subset of the grammar.

Forward references: declarations (6.7), function calls (6.5.2.2), lambda expressions (6.5.2.6), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.3), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements and blocks (6.8).

6.2.2 Linkages of identifiers

1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[32)] There are three kinds of linkage: external, internal, and none.

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.

3 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **`static`**, the identifier has internal linkage.[33)]

4 For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible,[34)] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

5 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **`extern`**. If the declaration of an identifier for an object has file scope and no storage-class specifier or only the specifier **`auto`**, its linkage is external.

6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier **`extern`**.

[31)] That means, that the outer declaration is not visible for the initializer.

[32)] There is no linkage between different identifiers.

[33)] A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.1.

[34)] As specified in 6.2.1, the later declaration might hide the prior declaration.

7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

8 **NOTE** Internal and external linkage is used to access objects or functions that have a lifetime of the whole program execution. It is therefore usually determined before the execution of a program starts. For variables with a lifetime that is not the whole program execution and that are accessed from lambda expressions an additional mechanism called identifier capture is available that dynamically provides the access to the current instance of such a variable within the active function call that defines it.

Forward references: storage durations of objects (6.2.4), declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

— *label names* (disambiguated by the syntax of the label declaration and use);

— the *tags* of structures, unions, and enumerations (disambiguated by following any[35] of the keywords **`struct`**, **`union`**, or **`enum`**);

— the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator);

— all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

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 of objects

1 An object has a *storage duration* that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.22.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[36] 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.

3 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 lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with the storage-class specifier **`_Thread_local`** has *thread storage duration*. Its 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 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 object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*, as do 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.

---

[35] There is only one name space for tags even though three are possible.

[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.

20 Any number of *derived types* can be constructed from the object and function types, as follows:

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*". The construction of an array type from an element type is called "array type derivation".

— A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

— A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

— A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". The construction of a function type from a return type is called "function type derivation".

— A *lambda type* is a complete object type that describes the value of a lambda expression. A lambda type is characterized but not determined by a return type that is inferred from the function body of the lambda expression, and by the number, order, and type of parameters that are expected for function calls, and by the lexical position of the lambda expressions in the program. The function type that has the same return type and list of parameter types as the lambda is called the *prototype* of the lambda. A lambda type has no syntax derivation[50] and the lexical position of the originating lambda expression determines its scope of visibility. Objects of such a type shall only be defined as a capture (of another lambda expression) or by an underspecified declaration for which the lambda type is inferred.[51] An object of lambda type shall only be modified by simple assignment (6.5.16.1).

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.

— An *atomic type* describes the type designated by the construct **_Atomic**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[52]

22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

[50] Not even a **typeof** type specifier with lambda type can be formed. So there is no syntax to make a lambda type a choice in a generic selection other than **default**

[51] Another possibility to create an object that has an effective lambda type is to copy a lambda value into allocated storage via simple assignment.

[52] Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

complex result value is a positive zero or an unsigned zero.

2 When a value of complex type is converted to a real type other than **_Bool**,[68] the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

6.3.1.8 Usual arithmetic conversions

1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a *common real type* for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the *usual arithmetic conversions*:

> First, if the corresponding real type of either operand is **long double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **long double**.
>
> Otherwise, if the corresponding real type of either operand is **double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **double**.
>
> Otherwise, if the corresponding real type of either operand is **float**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **float**.[69]
>
> Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
> > If both operands have the same type, then no further conversion is needed.
> >
> > Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
> >
> > Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
> >
> > Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
> >
> > Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

2 The values of floating operands and of the results of floating expressions may be represented in greater range and precision than that required by the type; the types are not changed thereby.[70]

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, function designators and lambdas

1 An *lvalue* is an expression (with an object type other than **void**) that potentially designates an object;[71] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate

---

[68] See 6.3.1.2.

[69] For example, addition of a **double _Complex** and a **float** entails just the conversion of the **float** operand to **double** (and yields a **double _Complex** result).

[70] The cast and assignment operators are still required to remove extra range and precision.

[71] The name "lvalue" comes originally from the assignment expression `E1 = E2`, in which the left operand `E1` is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called "rvalue" is in this document described as the "value of an expression".

An obvious example of an lvalue is an identifier of an object. As a further example, if `E` is a unary expression that is a pointer to an object, `*E` is an lvalue that designates the object to which `E` points.

the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the **typeof** specifier, the **sizeof** operator, the unary & operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined. If the lvalue designates an object of automatic storage duration that could have been declared with the **register** storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

3 Except when it is the operand of the **typeof** specifier, the unary **sizeof** operator, or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

4 A *function designator* is an expression that has function type. Except when it is the operand of the **typeof** specifier, the **sizeof** operator,[72] or the unary & operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

5 A function literal with a type "lambda with prototype `P`" can be converted implicitly or explicitly to an expression that has type "pointer to `Q`", where `Q` is a function type that is compatible with `P`.[73] The function pointer value behaves as if a function `F` of type `P` with internal linkage, a unique name, and the same parameter list and function body as for λ, where uses of identifiers from enclosing blocks in expressions that are not evaluated are replaced by proper types or values, had been defined in the translation unit, and the function pointer had been formed by function-to-pointer conversion of that function. The only difference is that the function pointer needs not necessarily to be distinct from any other compatible function pointer that provides the same observable behavior.

Forward references: lambda expressions (6.5.2.6) address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions `<stddef.h>` (7.19), **typeof** specifier 6.7.9, initialization (6.7.10), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and **_Alignof** operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 void

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

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.

---

[72] Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraints in 6.5.3.4.

[73] It follows that lambdas of different type cannot be assigned to each other. Thus, in the conversion of a function literal to a function pointer, the prototype of the originating lambda expression can be assumed to be known, and a diagnostic can be issued if the prototypes do not aggree.

— a type that is the signed or unsigned type corresponding to the effective type of the object,

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

— a character type.

8 A floating expression may be *contracted*, that is, evaluated as though it were a single operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[98] The `FP_CONTRACT` pragma in `<math.h>` provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.[99]

Forward references: the `FP_CONTRACT` pragma (7.12.2), copying functions (7.24.2).

6.5.1 Primary expressions

Syntax

1 *primary-expression:*
identifier
constant
string-literal
(*expression* **)**
generic-selection

Semantics

2 An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[100]

3 A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.

4 A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.

5 A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, a lambda expression, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, a lambda expression, or a void expression.

6 A generic selection is a primary expression. Its type and value depend on the selected generic association, as detailed in the following subclause.

Forward references: declarations (6.7).

6.5.1.1 Generic selection

Syntax

1 *generic-selection:*
_Generic (*assignment-expression* **,** *generic-assoc-list* **)**
generic-assoc-list:
generic-association
generic-assoc-list **,** *generic-association*
generic-association:
type-name **:** *assignment-expression*

[98] The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the final operation is rounded to the format determined by the expression evaluation method. A contracted expression might also omit the raising of floating-point exceptions.

[99] This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

[100] Thus, an undeclared identifier is a violation of the syntax.

`default :` *assignment-expression*

Constraints

2 A generic selection shall have no more than one **`default`** generic association. The type name in a generic association shall specify a complete object type other than a variably modified type. No two generic associations in the same generic selection shall specify compatible types. The type of the controlling expression is the type of the expression as if it had undergone an lvalue conversion,[101)] array to pointer conversion, or function to pointer conversion. That type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no **`default`** generic association, its controlling expression shall have type compatible with exactly one of the types named in its generic association list.

Semantics

3 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **`default`** generic association. None of the expressions from any other generic association of the generic selection is evaluated.

4 The type and value of a generic selection are identical to those of its result expression. It is an lvalue, a function designator, a lambda expression, or a void expression if its result expression is, respectively, an lvalue, a function designator, a lambda expression, or a void expression. A generic selection that is the operand of a **`typeof`** specification behaves as if the selected assignment expression had been the operand.

5 **EXAMPLE** The **`cbrt`** type-generic macro could be implemented as follows:

```
#define cbrt(X) _Generic((X),                   \
                         long double: cbrtl,    \
                         default: cbrt,         \
                         float: cbrtf           \
                         )(X)
```

6.5.2 Postfix operators

Syntax

1 *postfix-expression:*
primary-expression
postfix-expression **`[`** *expression* **`]`**
postfix-expression **`(`** *argument-expression-list*$_{\text{opt}}$ **`)`**
postfix-expression **`.`** *identifier*
postfix-expression **`->`** *identifier*
postfix-expression **`++`**
postfix-expression **`-`**
`(` *type-name* **`)`** **`{`** *initializer-list* **`}`**
`(` *type-name* **`)`** **`{`** *initializer-list* **`,`** **`}`**
lambda-expression

argument-expression-list:
assignment-expression
argument-expression-list **`,`** *assignment-expression*

[101)] An lvalue conversion drops type qualifiers.

6.5.2.1 Array subscripting

Constraints

1 One of the expressions shall have type "pointer to complete object *type*", the other expression shall have integer type, and the result has type "*type*".

Semantics

2 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2` -th element of `E1` (counting from zero).

3 Successive subscript operators designate an element of a multidimensional array object. If `E` is an n-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$, then `E` (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \cdots \times k$. If the unary $*$ operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n-1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

4 **EXAMPLE** Consider the array object defined by the declaration

```
int x[3][5];
```

Here `x`

is a 3×5 array of

`int` s; more precisely, `x` is an array of three element objects, each of which is an array of five **`int`** s. In the expression `x[i]`, which is equivalent to `(*((x)+(i)))`, `x` is first converted to a pointer to the initial array of five **`int`** s. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five **`int`** objects. The results are added and indirection is applied to yield an array of five **`int`** s. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the **`int`** s, so `x[i][j]` yields an **`int`**.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.6.2).

6.5.2.2 Function calls

Constraints

1 The ~~expression that denotes the called function~~postfix expression[102)] shall have ~~type~~ lambda type or pointer to function type, returning **`void`** or returning a complete object type other than an array type.

2 If the ~~expression that denotes the called function has a type that~~ postfix expression is a lambda or if the type of the function includes a prototype, the number of arguments shall agree with the number of parameters of the function or lambda type. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

3 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function or lambda. The list of expressions specifies the arguments to the function or lambda.

4 An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[103)]

5 If the expression that denotes the called function has lambda type or type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **`void`**.

[102)] Most often, this is the result of converting an identifier that is a function designator.

[103)] A function or lambda can change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function or lambda can then change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

6 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **`float`** are promoted to **`double`**. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

— one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;

— both types are pointers to qualified or unqualified versions of a character type or **`void`**.

7 If the expression that denotes the called function is a lambda or is a function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

8 No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

9 If the lambda or function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called lambda or function, the behavior is undefined.

10 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function or lambda is indeterminately sequenced with respect to the execution of the called function.[104)]

11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions or lambdas.

12 **EXAMPLE** In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

Forward references: function declarators (including prototypes) (6.7.6.3), function definitions (6.9.1), the **`return`** statement (6.8.6.4), simple assignment (6.5.16.1).

6.5.2.3 Structure and union members

Constraints

1 The first operand of the `.` operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

2 The first operand of the`->` operator shall have type “pointer to atomic, qualified, or unqualified structure” or “pointer to atomic, qualified, or unqualified union”, and the second operand shall name a member of the type pointed to.

Semantics

3 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member,[105)] and is an lvalue if the first expression is

[104)] In other words, function executions do not “interleave” with each other.

[105)] If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new

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 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 (6.7.10).

6.5.2.6 Lambda expressions

Syntax

1 *lambda-expression:*
capture-clause *parameter-clause*$_{\text{opt}}$ *attribute-specifier-sequence*$_{\text{opt}}$ *function-body*

capture-clause:
[*capture-list*$_{\text{opt}}$ **]**

capture-list:
capture-list-element
capture-list **,** *capture-list-element*

capture-list-element:
value-capture
identifier-capture

value-capture:
capture **=** *assignment-expression*

identifier-capture:
& *capture*

capture:
identifier

parameter-clause:
(*parameter-list*$_{\text{opt}}$ **)**

Constraints

2 An identifier shall appear at most once; either as a capture or as a parameter name in the parameter list. The identifier of an identifier capture shall designate an object of automatic storage duration that is defined in a scope that surrounds the lambda expression.

3 Within the lambda expression, identifiers (including captures and parameters of the lambda) shall be used according to the usual scoping rules, but outside the assignment expression of a value capture the following exceptions apply to identifiers that are declared in a block that strictly encloses the lambda expression and that are not identifier captures:

— Objects or type definitions with variably modified type shall not be used.

— Objects with automatic storage duration shall not be evaluated.[112)]

4 The function body shall be such that a return type *type* according to the rules in 6.8.6.4 can be inferred.

Semantics

5 The optional attribute specifier sequence in a lambda expression appertains to the resulting lambda type and to its function prototype. If the parameter clause is omitted, a clause of the form `()` is assumed. A lambda expression without any capture is called a *function literal expression* , otherwise it is called a *closure expression* . A lambda value originating from a function literal expression is called a *function literal* , otherwise it is called a *closure* .

6 Similar to a function definition, a lambda expression forms a single block that comprises all of its parts. Each capture and parameter has a scope of visibility that starts immediately after its definition is completed and extends to the end of the function body. In particular, captures and parameters are visible throughout the whole function body, unless they are redeclared in a depending block within that function body. Value captures and parameters have automatic storage duration; in each function call to the formed lambda value, a new instance of each value capture and parameter is created and initialized in order of declaration and has a lifetime until the end of the call, only that the addresses of value captures are not necessarily unique.

7 The assignment expression E in the definition of a value capture determines a value E_0 with type T_0, which is E after possible lvalue, array-to-pointer or function-to-pointer conversion. The type of the capture is T_0 **`const`** and its value is E_0 for all evaluations in all function calls to the lambda value. If, within the function body, the address of the capture or one of its members is taken, either explicitly by applying a unary & operator or by an array to pointer conversion,[113)] and that address is used to modify the underlying object, the behavior is undefined.

8 The evaluation of the assignment expressions of value captures takes place during each evaluation of the lambda expression. The evaluations for the value captures are sequenced in order of declaration; an earlier capture may occur within an assignment expression of a later one. The evaluation of a lambda expression is sequenced before any use of the resulting lambda value. For each call to a lambda value, value captures (with type and value as determined during the evaluation of the lambda expression) and then parameter types and values are determined in order

[112)] Identifiers of visible automatic objects that are not captures and that do not have a VM type, may still be used if they are not evaluated, for example in **`sizeof`** expressions, in **`typeof`** specifiers (if they are not lambdas themselves) or as controlling expression of a generic primary expression.

[113)] The capture does not have array type, but if it has a union or structure type, one of its members may have such a type.

of declaration. Value captures and earlier parameters may occur within the declaration of a later one.

9 The object of automatic storage duration of the surrounding scope that corresponds to an identifier capture shall be visible within the function body according to the usual scoping rules and shall be accessible within the function body throughout each call to the lambda. If the definition of the object uses the **register** storage class, the behavior is undefined. Access to the object within a call to the lambda follows the happens-before relation, in particular modifications to the object that happen before the call are visible within the call, and modifications to the object within the call are visible for all evaluations that happen after the call.[114)]

10 For each lambda expression, the return type *type* is inferred as indicated in the constraints. A lambda expression λ has an unspecified lambda type `L` that is the same for every evaluation of λ. As a result of the expression, a value of type `L` is formed that identifies λ and the specific set of values of the identifiers in the capture clause for the evaluation, if any. This is called a *lambda value* . It is unspecified, whether two lambda expressions λ and κ share the same lambda type even if they are lexically equal but appear at different points of the program. Objects of lambda type shall not be modified other then by simple assignment.

11 **NOTE 1** A direct function call to a function literal expression can be modeled by first performing a conversion of the function literal to a function pointer and then calling that function pointer.

12 **NOTE 2** A direct function call to a closure expression with parameters

```
[ captures ] (decl1, ..., decln) {
  block-item-list
}(arg1, ..., argn)
```

can be modeled with a such a call to a closure expression without parameters

```
[ _ArgCap1 = arg1, ..., _ArgCapn = argn, captures ] (void) {
  decl1 = _ArgCap1;
  ...
  decln = _ArgCapn;
  block-item-list
}()
```

where $\texttt{\_ArgCap}_1, \ldots, \texttt{\_ArgCap}_n$ are new identifiers that are unique for the translation unit. This equivalence uses the fact that the evaluation of the argument expressions $\texttt{arg}_1, \ldots, \texttt{arg}_n$ and the original closure expression as a whole can be evaluated without sequencing constraints before the actual function call operation. In particular, side effects that occur during the evaluation of any of the arguments or the capture list will not effect one another. This not withstanding, side effects that have an influence about the evaluation of captures in the specified capture list or that determine the type of parameters occur sequenced as specified in the original closure expression.

Recommended practice

13 Implementations are encouraged to diagnose any attempt to modify a lambda type object other than by assignment.

14 **EXAMPLE 1** The usual scoping rules extend to lambda expressions; the concept of captures only restricts which identifiers may be evaluated or not.

```
#include <stdio.h>
static long v;
int main(void) {
  [     ](void){ printf("%ld\n", v); }();               // valid, prints 0
  [     ](void){ printf("%zu\n", sizeof v); }();        // valid, prints sizeof(long)
  int v = 5;
  [     ](void){ printf("%d\n", v); }();                // invalid
  [     ](void){ extern long v; printf("%ld\n", v; }(); // valid, prints 0

  auto const λ = [v = v](void){ printf("%d\n", v); };  // freeze and shadow v
```

[114)] That is, evaluation of the identifier results in the same lvalue with the same type and address as for the scope surrounding the lambda. In particular, it is possible that the value of such an object becomes indeterminate after a call to **longjmp**, see 7.13.2.1.

```
  [&v   ](void){ v = 7; printf("%d\n", v); }();         // valid, prints 7
  λ();                                                  // valid, prints 5
  [v = v](void){ printf("%d\n", v); }();                // valid, prints 7
  [v = v](void){ printf("%zu\n", sizeof v); }();        // valid, prints sizeof(int)
  [     ](void){ printf("%zu\n", sizeof v); }();        // valid, prints sizeof(int)

}
```

15 **EXAMPLE 2** The following uses a function literal as a comparison function argument for **qsort**.

```
#define SORTFUNC(TYPE) [](size_t nmemb, TYPE A[nmemb]) {                  \
  qsort(A, nmemb, sizeof(A[0]),                                           \
        [](void const* x, void const* y){          /* comparison lambda  */ \
          TYPE X = *(TYPE const*)x;                                       \
          TYPE Y = *(TYPE const*)y;                                       \
          return (X < Y) ? -1 : ((X > Y) ? 1 : 0); /* return of type int */ \
        }                                                                 \
        );                                                                \
  return A;                                                               \
  }
  ...
  long C[5] = { 4, 3, 2, 1, 0, };
  SORTFUNC(long)(5, C);                          // lambda → (pointer →) function call
  ...
  auto const sortDouble = SORTFUNC(double);     // lambda value → lambda object
  double* (*sF)(size_t nmemb, double[nmemb]) = sortDouble;   // conversion
  ...
  double* ap = sortDouble(4, (double[]){ 5, 8.9, 0.1, 99, });
  double B[27] = { /* some values ... */ };
  sF(27, B);                                     // reuses the same function
  ...
  double* (*sG)(size_t nmemb, double[nmemb]) = SORTFUNC(double); // conversion
```

This code evaluates the macro `SORTFUNC` twice, therefore in total four lambda expressions are formed.

The function literals of the "comparison lambdas" are not operands of a function call expression, and so by conversion a pointer to function is formed and passed to the corresponding call of **qsort**. Since the respective captures are empty, the effect is as if to define two comparison functions, that could equally well be implemented as **static** functions with auxiliary names and these names could be used to pass the function pointers to **qsort**.

The outer lambdas are again without capture. In the first case, for **long**, the lambda value is subject to a function call, and it is unspecified if the function call uses a specific lambda type or directly uses a function pointer. For the second, a copy of the lambda value is stored in the variable `sortDouble` and then converted to a function pointer `sF`. Other than for the difference in the function arguments, the effect of calling the lambda value (for the compound literal) or the function pointer (for array `B`) is the same.

For optimization purposes, an implementation may fold lambda values that are expanded at different points of the program such that effectively only one function is generated. For example here the function pointers `sF` and `sG` may or may not be equal.

16 **EXAMPLE 3**

```
void matmult(size_t k, size_t n, size_t m,
             double const A[k][n], double const B[n][m], double const C[k][m]) {
  // dot product with stride of m for B
  // ensure constant propagation of n and m
  auto const λ0 = [ν=n, μ=m](double const x[ν], double const B[ν][μ], size_t m0) {
    double ret = 0.0;
    for (size_t i = 0; i < ν; ++i) {
      ret += x[i]*B[i][m0];
    }
    return ret;
  };
  // vector matrix product
```

```
  // ensure constant propagation of n and m, and accessibility of λ0
  auto const λ1 = [ν=n, μ=m, &λ0](double const x[ν], double const B[ν][μ],
                                  double res[m]) {
    for (size_t m0 = 0; m0 < μ; ++m0) {
      res[m0] = λ0(x, B, m0);
    }
  };
  for (size_t k0 = 0; k0 < k; ++k0) {
    double const (*Ap)[l] = A[k0];
    double (*Cp)[m] = C[k0];
    λ1(*Ap, B, *Cp);
  }
}
```

This function evaluates two closures; `λ0` has a return type of **`double`**, `λ1` of **`void`**. Both lambda values serve repeatedly as first operand to function evaluation but the evaluation of the captures is only done once for each of the closures. For the purpose of optimization, an implementation could generate copies of the underlying functions for each evaluation of such a closure such that the values of the captures ν and μ are replaced on a machine instruction level.

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.

Semantics

2 The value of the operand of the prefix `++` operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`. See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

3 The prefix `--` operator is analogous to the prefix `++` operator, except that the value of the operand is decremented.

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.3.2 Address and indirection operators

Constraints

1 The operand of the unary & operator shall be either a function designator, the result of a `[]` or unary `*` operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **`register`** storage-class specifier.

2 The operand of the unary `*` operator shall have pointer type.

a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[116] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is **size_t**, defined in <stddef.h> (and other headers).

6 **EXAMPLE 1** A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the alloc function presumably ensures that its return value is aligned suitably for conversion to a pointer to **double**.

7 **EXAMPLE 2** Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

8 **EXAMPLE 3** In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>

size_t fsize3(int n)
{
      char b[n+3];       // variable length array
      return sizeof b;   // execution time sizeof
}

int main()
{
      size_t size;
      size = fsize3(10); // fsize3 returns 13
      return 0;
}
```

Forward references: common definitions <stddef.h> (7.19), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.7), array declarators (6.7.6.2).

6.5.4 Cast operators

Syntax

1 *cast-expression:*

unary-expression
(*type-name* **)** *cast-expression*

Constraints

2 Unless the type name specifies a void type, the type name shall specify atomic, qualified, or unqualified scalar type, and the operand shall have scalar type, or, the type name shall specify an atomic, qualified, or unqualified pointer to function with prototype, and the operand is a function literal such that a conversion (6.3.2.1) from the function literal to the function pointer type is defined.

3 Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

4 A pointer type shall not be converted to any floating type. A floating type shall not be converted to

[116] When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the adjusted (pointer) type (see 6.9.1).

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold:[125)]

— the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type;

— the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;

— the left operand has the unqualified version of the lambda type of the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of **`void`**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand is an atomic, qualified, or unqualified pointer to function with a prototype, the right operand is a function literal, and the prototypes of the function pointer and of the function literal shall be such that a conversion from the function literal to the function pointer type is defined;

— the left operand is an atomic, qualified, or unqualified pointer, and the right is a null pointer constant; or

— the left operand has type atomic, qualified, or unqualified **`_Bool`**, and the right is a pointer.

Semantics

2 In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

4 **EXAMPLE 1** In the program fragment

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
      /* ... */
```

the **`int`** value returned by the function could be truncated when stored in the **`char`**, and then converted back to **`int`** width prior to the comparison. In an implementation in which "plain" **`char`** has the same range of values as **`unsigned char`** (and **`char`** is narrower than **`int`**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable `c` would be declared as **`int`**.

5 **EXAMPLE 2** In the fragment:

```
char c;
int i;
long l;

l = (c = i);
```

[125)] The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **`const`** but not **`volatile`** from the type **`int volatile * const`**).

the value of `i` is converted to the type of the assignment expression `c = i`, that is, **`char`** type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, **`long int`** type.

6 **EXAMPLE 3** Consider the fragment:

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;       // constraint violation
*cpp = &c;      // valid
*p = 0;         // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object `c`.

7 **EXAMPLE 4** Lambda types can be assigned in a portable way, only if both lambda types originate from the same lambda expression.

```
auto λ = [s = 0](){ puts("hello"); };
auto κ = [s = 0](){ puts("hello"); };
κ = λ;                                          // invalid, different types
auto λp = (false ? &λ : malloc(sizeof(λ))); // pointer to lambda
*λp = λ;                                        // valid, same type
(*λp)();                                        // valid, prints ``hello''
```

6.5.16.2 Compound assignment

Constraints

1 For the operators `+=` and `-=` only, either the left operand shall be an atomic, qualified, or unqualified pointer to a complete object type, and the right shall have integer type; or the left operand shall have atomic, qualified, or unqualified arithmetic type, and the right shall have arithmetic type.

2 For the other operators, the left operand shall have atomic, qualified, or unqualified arithmetic type, and (considering the type the left operand would have after lvalue conversion) each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.

Semantics

3 A *compound assignment* of the form `E1` *op*`=` `E2` is equivalent to the simple assignment expression `E1 = E1` *op* `(E2)`, except that the lvalue `E1` is evaluated only once, and with respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. If `E1` has an atomic type, compound assignment is a read-modify-write operation with **`memory_order_seq_cst`** memory order semantics.

4 **NOTE** Where a pointer to an atomic object can be formed and `E1` and `E2` have integer type, this is equivalent to the following code sequence where *T1* is the type of `E1` and *T2* is the type of `E2`:

```
T1 *addr = &E1;
T2 val = (E2);
T1 old = *addr;
T1 new;
do {
      new = old op val;
} while (!atomic_compare_exchange_strong(addr, &old, new));
```

with `new` being the result of the operation.

If `E1` or `E2` has floating type, then exceptional conditions or floating-point exceptions encountered during discarded evaluations of `new` would also be discarded in order to satisfy the equivalence of `E1` *op*`=` `E2` and `E1 = E1` *op* `(E2)`. For example, if Annex F is in effect, the floating types involved have IEC 60559 formats, and **`FLT_EVAL_METHOD`** is 0, the equivalent code would be:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
```

pointer:
`*` *type-qualifier-list*$_{\text{opt}}$
`*` *type-qualifier-list*$_{\text{opt}}$ *pointer*
type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier
parameter-type-list:
parameter-list
parameter-list `, ...`
parameter-list:
parameter-declaration
parameter-list `,` *parameter-declaration*
parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{\text{opt}}$
identifier-list:
identifier
identifier-list `,` *identifier*

Semantics

2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

3 A *full declarator* is a declarator that is not part of another declarator. If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified.

4 In the following subclauses, consider a declaration

`T D1`

where `T` contains the declaration specifiers that specify a type *T* (such as **`int`**) and `D1` is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

5 If, in the declaration "`T D1`", `D1` has the form

identifier

then the type specified for *ident* is *T*.

6 If, in the declaration "`T D1`", `D1` has the form

`(` `D` `)`

then *ident* has the type specified by the declaration "`T D`". Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

Implementation limits

7 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or **`void`** type, either directly or via one or more **`typedef`** s.

Forward references: array declarators (6.7.6.2), type definitions (6.7.8)~~.~~, type inference (6.7.11).

6.7.6.1 Pointer declarators

Semantics

1 If, in the declaration "`T D1`", `D1` has the form

`*` *type-qualifier-list*$_{\text{opt}}$ `D`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the

declare a typedef name `t` with type **`signed int`**, a typedef name `plain` with type **`int`**, and a structure with three bit-field members, one named `t` that contains values in the range $[0, 15]$, an unnamed const-qualified bit-field which (if it could be accessed) would contain values in either the range $[-15, +15]$ or $[-16, +15]$, and one named `r` that contains values in one of the ranges $[0, 31]$, $[-15, +15]$, or $[-16, +15]$. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **`unsigned`** is a type specifier (which forces `t` to be the name of a structure member), while **`const`** is a type qualifier (which modifies `t` which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function `f` is declared with type "function returning **`signed int`** with one unnamed parameter with type pointer to function returning **`signed int`** with one unnamed parameter with type **`signed int`**", and an identifier `t` with type **`long int`**.

7 **EXAMPLE 4** On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **`signal`** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

8 **EXAMPLE 5** If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
void copyt(int n)
{
      typedef int B[n];   // B is n ints, n evaluated now
      n += 1;
      B a;                // a is n ints, n without += 1
      int b[n];           // a and b are different sizes
      for (int i = 1; i  <  n; i++)
            a[i-1] = b[i];
}
```

6.7.9 `typeof` specifier

Syntax

1 *typeof-specifier:*

`typeof` **`(`** *type-name* **`)`**
`typeof` **`(`** *expression* **`)`**

Constraints

2 The type name or expression shall be valid and have a function or object type, but not a lambda type. No new type declaration shall be formed by the type name or expression themselves.[160)]

Semantics

3 A **`typeof`** specifier can be used in places where other type specifiers are used to declare or define objects, members or functions. It stands in for the unmodified type of the type name or expression, even where the expression cannot be used for type inference of its type (opaque types, function types, array types), where a type-qualification should not be dropped, or where an identifier may only be accessed for its type without evaluating it (within lambda expressions).

4 If it does not have a variably modified (VM) type, the type name or expression is not evaluated. For VM types, the same rules for evaluation as for **`sizeof`** expressions apply. Analogous to **`typedef`**, a

[160)] This could for example happen if the expression contained the forward declaration of a tag type, such as in `(struct newStruct*)0` where `struct newStruct` has not yet been declared, or if it uses a compound literal that declares a new structure or union type in its *type-name* component.

```
{
      struct S l = { 1, .t = x, .t.l = 41, };
}
```

The value of `l.t.k` is 42, because implicit initialization does not override explicit initialization.

37 **EXAMPLE 13** Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

38 In the above, if **MAX** is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

39 **EXAMPLE 14** Any member of a union can be initialized:

```
union { /* ... */ } u = {.any_member = 42 };
```

Forward references: common definitions <stddef.h> (7.19).

6.7.11 Type inference

Constraints

1 An underspecified declaration shall contain the storage class specifier **auto**.

2 For an identifier that is declared but not defined by an underspecified declaration, a prior definition shall be be visible. For an underspecified declaration which is not the declaration of a parameter, an init-declarator corresponding to the definition of an object shall have of one of the forms

declarator **=** *assignment-expression*
declarator **= {** *assignment-expression* **}**
declarator **= {** *assignment-expression* **, }**

such that the declarator does not declare an array.[166] If the assignment expression has lambda type, the declaration shall only declare one object and shall only consist of storage class specifiers, qualifiers, the identifier that is to be declared, and the initializer.

3 ~~Prior~~ Unless it is the definition of an object with an assignment expression of lambda type as above, prior to an underspecified declaration there shall exist a **typeof** specifier *type* that if used to replace the **auto** specifier makes the adjusted declaration a valid declaration.[167] If it is also the definition of a function the return type shall be determined from **return** statements (or the lack thereof) as specified in 6.9.1. Otherwise, *type* shall be such that for all defined objects the assignment expression in the corresponding init-declarator, after possible lvalue, array-to-pointer or function-to-pointer conversion, has the non-atomic, unqualified type of the declared object.

Description

4 ~~In~~ Although there is no syntax derivation to form declarators of lambda type, a value λ of lambda type L can be used as assignment expression to initialize an underspecified object declaration and as the return value of an underspecified function. The inferred type then is L, possibly qualified, and the visibility of L extends to the visibility scope of the declared object or function. Otherwise, in an underspecified declaration the type of the declared identifiers is the type after the declaration would have been adjusted by a choice for *type* as described in the constraints. If the declaration is also an object definition, each assignment expressions that is used to determine the type and initial value of an object is evaluated exactly once each time the declaration is met.

5 **NOTE 1** Because of the relatively complex syntax and semantics of type specifiers, the requirements for *type* use a **typeof** specifier. For an underspecified declaration

```
auto x = v;
```

[166] The scope rules as described in 6.2.1 also prohibit the use of the identifier of the declarator within the assignment expression.

[167] The qualification of the type of an lvalue that is the assignment expression, or the fact that it is atomic, can never be used to infer such a property of the type of the defined object.

6.8 Statements and blocks

Syntax

1 *statement:*

labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression, nor part of a declarator or abstract declarator. There is also an implicit full expression in which the non-constant size expressions for a variably modified type are evaluated; within that full expression, the evaluation of different size expressions are unsequenced with respect to one another. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

5 **NOTE** Each of the following is a full expression:

- a full declarator for a variably modified type,
- an initializer that is not part of a compound literal,
- the expression in an expression statement,
- the controlling expression of a selection statement (**`if`** or **`switch`**),
- the controlling expression of a **`while`** or **`do`** statement,
- each of the (optional) expressions of a **`for`** statement,
- the (optional) expression in a **`return`** statement.

While a constant expression satisfies the definition of a full expression, evaluating it does not depend on nor produce any side effects, so the sequencing implications of being a full expression are not relevant to a constant expression.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 *labeled-statement:*

identifier **`:`** *statement*
`case` *constant-expression* **`:`** *statement*
`default` **`:`** *statement*

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement ~~.~~ that is associated with the same function body as the statement to which the label is attached.[168)] Further constraints on such labels are discussed under the **`switch`** statement.

[168)] Thus, a label that appears within a lambda expression may only be associated to a switch statement within the body of the lambda.

6.8.5.3 The `for` statement

1 The statement

```
for (clause-1; expression-2; expression-3) statement
```

behaves as follows: The expression *expression-2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression-3* is evaluated as a void expression after each execution of the loop body. If *clause-1* is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[174)]

2 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant.

6.8.6 Jump statements

Syntax

1 *jump-statement:*

goto *identifier* **;**
continue ;
break ;
return *expression*$_{\text{opt}}$ **;**

Constraints

2 No jump statement other than **return** shall have a target that is found in another function body.[175)]

Semantics

3 A jump statement causes an unconditional jump to another place.

6.8.6.1 The `goto` statement

Constraints

1 The identifier in a **goto** statement shall name a label located somewhere in the enclosing function body. A **goto** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.[176)]

Semantics

2 A **goto** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

3 **EXAMPLE 1** It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **continue** statements, for example.)

```
/* ... */
goto first_time;
for (;;) {
```

[174)] Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

[175)] Thus jump statements other than **return** may not jump between different functions or cross the boundaries of a lambda expression, that is, they may not jump into or out of the function body of a lambda. Other features such as signals (7.14) and long jumps (7.13) may delegate control to points of the program that do not fall under these constraints.

[176)] The visibility of labels is restricted such that a **goto** statement that jumps into or out of a different function body, even if it is nested within a lambda, is a constraint violation.

```
        // determine next operation
        /* ... */
        if (need to reinitialize) {
              // reinitialize-only code
              /* ... */
        first_time:
              // general initialization code
              /* ... */
              continue;
        }
        // handle other operations
        /* ... */
  }
```

4 **EXAMPLE 2** A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
  goto lab3;              // invalid:  going INTO scope of VLA.
  {
        double a[n];
        a[j] = 4.4;
  lab3:
        a[j] = 3.3;
        goto lab4;        // valid:  going WITHIN scope of VLA.
        a[j] = 5.5;
  lab4:
        a[j] = 6.6;
  }
  goto lab4;              // invalid:  going INTO scope of VLA.
```

6.8.6.2 The **continue** statement

Constraints

1 A **continue** statement shall appear only in or as a loop body ~~.~~ that is associated to the same function body.[177)]

Semantics

2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```
while (/* ...  */) {
   /* ...  */
   continue;
   /* ...  */
contin:;
}
```

```
do {
   /* ...  */
   continue;
   /* ...  */
contin:;
} while (/* ...  */);
```

```
for (/* ...  */) {
   /* ...  */
   continue;
   /* ...  */
contin:;
}
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto** `contin`;.[178)]

6.8.6.3 The **break** statement

Constraints

1 A **break** statement shall appear only in or as a switch body or loop body ~~.~~ that is associated to the same function body.[179)]

[177)] Thus a **continue** statement by itself may not be used to terminate the execution of the body of a lambda expresssion.
[178)] Following the `contin`: label is a null statement.
[179)] Thus a **break** statement by itself may not be used terminate the execution of the body of a lambda expresssion.

Semantics

2 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

6.8.6.4 The `return` statement

Constraints

1 A **return** statement with an expression shall not appear in a function body whose return type is **void**. A **return** statement without an expression shall only appear in a function body whose return type is **void**.

2 For a function ~~that has~~ body that corresponds to an underspecified definition of a function or to a lambda, all **return** statements shall provide expressions with a consistent type or none at all. That is, if any **return** statement has an expression, all **return** statements shall have an expression (after lvalue, array-to-pointer or function-to-pointer conversion) with the same type; otherwise all **return** expressions shall have no expression.

Semantics

3 A **return** statement is associated to the innermost function body in which appears. It evaluates the expression, if any, terminates the execution of ~~the~~ that function body and returns control to ~~the caller~~its caller; if it has an expression, the value of the expression is returned to the caller as the value of the function call expression. A function body may have any number of **return** statements.

4 If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[180)]

5 For a lambda or for a function that has an underspecified definition, the return type is determined by the lexically first **return** statement, if any, that is associated to the function body and is specified as the type of that expression, if any, after lvalue, array-to-pointer, function-to-pointer conversion, or as **void** if there is no expression.

6 **EXAMPLE** In:

```
struct s { double i; } f(void);
union {
      struct {
            int f1;
            struct s f2;
      } u1;
      struct {
            struct s f3;
            int f4;
      } u2;
} g;

struct s f(void)
{
      return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

[180)] The **return** statement is not an assignment. The overlap restriction of 6.5.16.1 does not apply to the case of function return. The representation of floating-point values can have wider range or precision than implied by the type; a cast can be used to remove this extra range and precision.

— 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 non-modifiable storage 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.[208)] 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.[209)]

— 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.[210)]

— 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.

4 The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration.[211)]

5 Unless explicitly stated otherwise in the detailed descriptions that follow, library functions shall prevent data races as follows: A library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments. A library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly

[208)] 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.

[209)] Such macros might not contain the sequence points that the corresponding function calls do. Nevertheless, it is recommended that implementations provide the same sequencing properties as for a function call, by, for example, wrapping the macro expansion in a suitable lambda expression.

[210)] 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.

[211)] Thus, a signal handler cannot, in general, call standard library functions.

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 body 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[273)] 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 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 or lambda 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.

[273)] Such a function contains the call to **setjmp** either directly or within a set of nested lambdas. All local variables of the function and the nested lambdas that have been modified between the corresponding calls to **setjmp** and **longjmp** function are affected.

7.14 Signal handling <signal.h>

1 The header <signal.h> declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

2 The type defined is

```
sig_atomic_t
```

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

3 The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the **signal** function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type **int** and distinct values that are the signal numbers, each corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the **abort** function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

4 An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter,[274)] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

7.14.1 Specify signal handling

7.14.1.1 The signal function

Synopsis

1

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Description

2 The **signal** function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is **SIG_DFL**, default handling for that signal will occur. If the value of `func` is **SIG_IGN**, the signal will be ignored. Otherwise, `func` shall point to a function or shall be the result of a conversion of a function literal to a function pointer. The function or lambda value is then to be called when that signal occurs~~.~~, An invocation of such a function or function literal because of a signal, or (recursively) of any further functions or lambdas called by that invocation (other than functions in the standard library),[275)] is called a *signal handler*.

[274)] See "future library directions" (7.31.7). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

[275)] This includes functions called indirectly via standard library functions (e.g., a **SIGABRT** handler called via the **abort** function).

3 When a signal occurs and `func` points to a function,[276] it is implementation-defined whether the equivalent of `signal(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of **SIGILL**, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of `sig` is **SIGFPE**, **SIGILL**, **SIGSEGV**, or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

4 If the signal occurs as the result of calling the **abort** or **raise** function, the signal handler shall not call the **raise** function.

5 If the signal occurs other than as the result of calling the **abort** or **raise** function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as **volatile sig_atomic_t**, or the signal handler calls any function in the standard library other than

— the **abort** function,

— the **_Exit** function,

— the **quick_exit** function,

— the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,

— the **atomic_is_lock_free** function with any atomic argument, or

— the **signal** function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the **signal** function results in a **SIG_ERR** return, the value of **errno** is indeterminate.[277]

6 At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

7 Use of this function in a multi-threaded program results in undefined behavior. The implementation shall behave as if no library function calls the **signal** function.

Returns

8 If the request can be honored, the **signal** function returns the value of `func` for the most recent successful call to **signal** for the specified signal `sig`. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Forward references: the **abort** function (7.22.4.1), the **exit** function (7.22.4.4), the **_Exit** function (7.22.4.5), the **quick_exit** function (7.22.4.7).

7.14.2 Send signal

7.14.2.1 The raise function

Synopsis

1
```
#include <signal.h>
int raise(int sig);
```

[276] Or, equivalently, it is the result of a conversion of a function literal to a function pointer.
[277] If any signal is generated by an asynchronous signal handler, the behavior is undefined.

7.16 Variable arguments <stdarg.h>

1 The header <stdarg.h> declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

2 A function may be called with a variable number of arguments of varying types. As described in 6.9.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

3 The type declared is

```
va_list
```

which is a complete object type suitable for holding information needed by the macros **va_start**, **va_arg**, **va_end**, and **va_copy**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as ap in this subclause) having type **va_list**. The object ap may be passed as an argument to another function ~~; if that function~~ call; if the called function or lambda invokes the **va_arg** macro with parameter ap, the value of ap in the calling function or lambda is indeterminate and shall be passed to the **va_end** macro prior to any further reference to ap.[278)]

4 **NOTE** Because the ... parameter syntax is not valid for lambda expressions, these macros can never be applied directly to process a variable list of arguments to the call of a lambda. In contrast to that, the type **va_list** itself can be a parameter type of a lambda expression to process the argument list of a function.

7.16.1 Variable argument list access macros

1 The **va_start** and **va_arg** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **va_copy** and **va_end** are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the **va_start** and **va_copy** macros shall be matched by a corresponding invocation of the **va_end** macro in the same function or lambda expression.

7.16.1.1 The va_arg macro

Synopsis

1
```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

2 The **va_arg** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter ap shall have been initialized by the **va_start** or **va_copy** macro (without an intervening invocation of the **va_end** macro for the same ap). Each invocation of the **va_arg** macro modifies ap so that the values of successive arguments are returned in turn. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

— one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;

— one type is pointer to **void** and the other is a pointer to a character type.

Returns

3 The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

[278)] It is permitted to create a pointer to a **va_list** and pass that pointer to another function or lambda, in which case the ~~original~~ calling function or lambda can make further use of the original list after the other function returns.

7.16.1.2 The `va_copy` macro

Synopsis

1
```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Description

2 The **`va_copy`** macro initializes `dest` as a copy of `src`, as if the **`va_start`** macro had been applied to `dest` followed by the same sequence of uses of the **`va_arg`** macro as had previously been used to reach the present state of `src`. Neither the **`va_copy`** nor **`va_start`** macro shall be invoked to reinitialize `dest` without an intervening invocation of the **`va_end`** macro for the same `dest`.

Returns

3 The **`va_copy`** macro returns no value.

7.16.1.3 The `va_end` macro

Synopsis

1
```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

2 The **`va_end`** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the **`va_start`** macro, or the function or lambda expression containing the expansion of the **`va_copy`** macro, that initialized the **`va_list`** `ap`. The **`va_end`** macro may modify `ap` so that it is no longer usable (without being reinitialized by the **`va_start`** or **`va_copy`** macro). If there is no corresponding invocation of the **`va_start`** or **`va_copy`** macro, or if the **`va_end`** macro is not invoked before the return, the behavior is undefined.

Returns

3 The **`va_end`** macro returns no value.

7.16.1.4 The `va_start` macro

Synopsis

1
```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Description

2 The **`va_start`** macro shall be invoked before any access to the unnamed arguments.

3 The **`va_start`** macro initializes `ap` for subsequent use by the **`va_arg`** and **`va_end`** macros. Neither the **`va_start`** nor **`va_copy`** macro shall be invoked to reinitialize `ap` without an intervening invocation of the **`va_end`** macro for the same `ap`.

4 The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If the parameter *parmN* is declared with the **`register`** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

Returns

5 The **`va_start`** macro returns no value.

6 **EXAMPLE 1** The function `f1` gathers into an array a list of arguments that are pointers to strings (but not more than `MAXARGS` arguments), then passes the array as a single argument to function `f2`. The number of pointers is specified by the first argument to `f1`.

Returns

4 The **realloc** function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object has not 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 or function literal pointed to by func, to be called without arguments at normal program termination.[323] 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~~function pointers.

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));
```

Description

2 The **at_quick_exit** function registers the function or function literal pointed to by func, to be called without arguments should **quick_exit** be called.[324] It is unspecified whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed.

[323] 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.

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

Environmental limits

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

Returns

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

Forward references: the **quick_exit** function (7.22.4.7).

7.22.4.4 The exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void exit(int status);
```

Description

2 The **exit** function causes normal program termination to occur. No ~~functions~~ function pointers registered by the **at_quick_exit** function are called. If a program calls the **exit** function more than once, or calls the **quick_exit** function in addition to the **exit** function, the behavior is undefined.

3 First, all ~~functions~~ function pointers registered by the **atexit** function are called, in the reverse order of their registration,[325)] except that a function pointer is called after any previously registered ~~functions~~ function pointers that had already been called at the time it was registered. If, during the call to any such function or function literal, a call to the **longjmp** function is made that would terminate the call to the registered function or function literal, the behavior is undefined.

4 Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **tmpfile** function are removed.

5 Finally, control is returned to the host environment. If the value of `status` is zero or **EXIT_SUCCESS**, an implementation-defined form of the status *successful termination* is returned. If the value of `status` is **EXIT_FAILURE**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

6 The **exit** function cannot return to its caller.

7.22.4.5 The _Exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void _Exit(int status);
```

Description

2 The **_Exit** function causes normal program termination to occur and control to be returned to the host environment. No ~~functions~~ function pointers registered by the **atexit** function, the **at_quick_exit** function, or signal handlers registered by the **signal** function are called. The status returned to the host environment is determined in the same way as for the **exit** function (7.22.4.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

Returns

3 The **_Exit** function cannot return to its caller.

7.22.4.6 The getenv function

Synopsis

1
```
#include <stdlib.h>
```

[325)] Each function is called as many times as it was registered, and in the correct order with respect to other registered ~~functions~~function pointers.

```
char *getenv(const char *name);
```

Description

2 The **getenv** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by name. The set of environment names and the method for altering the environment list are implementation-defined. The **getenv** function need not avoid data races with other threads of execution that modify the environment list.[326)]

3 The implementation shall behave as if no library function calls the **getenv** function.

Returns

4 The **getenv** function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **getenv** function. If the specified name cannot be found, a null pointer is returned.

7.22.4.7 The quick_exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void quick_exit(int status);
```

Description

2 The **quick_exit** function causes normal program termination to occur. No ~~functions~~ function pointers registered by the **atexit** function or signal handlers registered by the **signal** function are called. If a program calls the **quick_exit** function more than once, or calls the **exit** function in addition to the **quick_exit** function, the behavior is undefined. If a signal is raised while the **quick_exit** function is executing, the behavior is undefined.

3 The **quick_exit** function first calls all ~~functions~~ function pointers registered by the **at_quick_exit** function, in the reverse order of their registration,[327)] except that a function pointer is called after any previously registered ~~functions~~ function pointers that had already been called at the time it was registered. If, during the call to any such function or function literal, a call to the **longjmp** function is made that would terminate the call to the registered function pointer, the behavior is undefined.

4 Then control is returned to the host environment by means of the function call **_Exit**(status).

Returns

5 The **quick_exit** function cannot return to its caller.

7.22.4.8 The system function

Synopsis

1
```
#include <stdlib.h>
int system(const char *string);
```

Description

2 If string is a null pointer, the **system** function determines whether the host environment has a *command processor*. If string is not a null pointer, the **system** function passes the string pointed to by string to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling **system** to behave in a non-conforming manner or to terminate.

Returns

3 If the argument is a null pointer, the **system** function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the **system** function does return, it returns an

[326)] Many implementations provide non-standard functions that modify the environment list.

[327)] Each function pointer is called as many times as it was registered, and in the correct order with respect to other registered ~~functions~~function pointers.

implementation-defined value.

7.22.5 Searching and sorting utilities

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

2 The implementation shall ensure that the second argument of the comparison function or function literal (when called from **`bsearch`**), or both arguments (when called from **`qsort`**), are pointers to elements of the array.[328] The first argument when called from **`bsearch`** shall equal `key`.

3 The comparison function or function literal shall not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function or function literal, but shall not alter the contents of any individual element.

4 When the same objects (consisting of `size` bytes, irrespective of their current positions in the array) are passed more than once to the comparison function or function literal, the results shall be consistent with one another. That is, for **`qsort`** they shall define a total ordering on the array, and for **`bsearch`** the same object shall always compare the same way with the key.

5 A sequence point occurs immediately before and immediately after each call to the comparison function or function literal, and also between any call to the comparison function or function literal and any movement of the objects passed as arguments to that call.

7.22.5.1 The `bsearch` function

Synopsis

1
```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
      size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Description

2 The **`bsearch`** function searches an array of `nmemb` objects, the initial element of which is pointed to by `base`, for an element that matches the object pointed to by `key`. The size of each element of the array is specified by `size`.

3 The comparison function or function literal pointed to by `compar` is called with two arguments that point to the `key` object and to an array element, in that order. ~~The function~~ A function call shall return an integer less than, equal to, or greater than zero if the `key` object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the `key` object, in that order.[329]

Returns

4 The **`bsearch`** function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

---

[328] That is, if the value passed is `p`, then the following expressions are always nonzero:

```
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p  <  (char *)base + nmemb * size
```

[329] In practice, the entire array is sorted according to the comparison function.

7.22.5.2 The `qsort` function

Synopsis

1
```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Description

2 The **`qsort`** function sorts an array of `nmemb` objects, the initial element of which is pointed to by `base`. The size of each object is specified by `size`.

3 The contents of the array are sorted into ascending order according to a comparison function or function literal pointed to by `compar`, which is called with two arguments that point to the objects being compared. ~~The function~~ A function call shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

4 If two elements compare as equal, their order in the resulting sorted array is unspecified.

Returns

5 The **`qsort`** function returns no value.

7.22.6 Integer arithmetic functions

7.22.6.1 The `abs`, `labs` and `dlabs` functions

Synopsis

1
```
#include <stdlib.h>
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
```

Description

2 The **`abs`**, **`labs`**, and **`llabs`** functions compute the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.[330)]

Returns

3 The **`abs`**, **`labs`**, and **`llabs`**, functions return the absolute value.

7.22.6.2 The `div`, `ldiv`, and `lldiv` functions

Synopsis

1
```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Description

2 The **`div`**, **`ldiv`**, and **`lldiv`**, functions compute `numer/denom` and `numer%denom` in a single operation.

Returns

3 The **`div`**, **`ldiv`**, and **`lldiv`** functions return a structure of type **`div_t`**, **`ldiv_t`**, and **`lldiv_t`**, respectively, comprising both the quotient and the remainder. The structures shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has the same type as the arguments `numer` and `denom`. If either part of the result cannot be represented, the behavior is undefined.

[330)] The absolute value of the most negative number cannot be represented in two's complement.

(6.5.1.1) *generic-selection:*
_Generic **(** *assignment-expression* **,** *generic-assoc-list* **)**

(6.5.1.1) *generic-assoc-list:*
generic-association
generic-assoc-list **,** *generic-association*

(6.5.1.1) *generic-association:*
type-name **:** *assignment-expression*
default **:** *assignment-expression*

(6.5.2) *postfix-expression:*
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list*$_{\text{opt}}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **-**
(*type-name* **)** **{** *initializer-list* **}**
(*type-name* **)** **{** *initializer-list* **,** **}**
lambda-expression

(6.5.2) *argument-expression-list:*
assignment-expression
argument-expression-list **,** *assignment-expression*

(6.5.2.6) *lambda-expression:*
capture-clause *parameter-clause*$_{\text{opt}}$ *attribute-specifier-sequence*$_{\text{opt}}$ *function-body*

(6.5.2.6) *capture-clause:*
[*capture-list*$_{\text{opt}}$ **]**

(6.5.2.6) *capture-list:*
capture-list-element
capture-list **,** *capture-list-element*

(6.5.2.6) *capture-list-element:*
value-capture
identifier-capture

(6.5.2.6) *value-capture:*
capture **=** *assignment-expression*

(6.5.2.6) *identifier-capture:*
& *capture*

(6.5.2.6) *capture:*
identifier

(6.5.2.6) *parameter-clause:*
(*parameter-list*$_{\text{opt}}$ **)**

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