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

N2638 v1
P2304R0

2021-1-20

Improve type generic programming
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

C already has a variaty of interfaces for type-generic programming, but lacks a systematic approach that provides type safety, strong ecapsulation and general usability. This paper is a summary paper for a series that provides improvements through

N2632. type inference for variable definitions (**auto** feature) and function return
N2633. function literals and value closures
N2634. type-generic lambdas (with **auto** parameters)
N2635. lvalue closures (pseudo-references for captures)

The aim is to have a complete set of features that allows to easily specify and reuse type-generic code that can equally be used by applications or by library implementors. All this by remaining faithful to C's efficient approach of static types and automatic (stack) allocation of local variables, by avoiding superfluous indirections and object aliasing, and by forcing no changes to existing ABI.

Contents

I. INTRODUCTION

With the exception of type casts and pointer conversions to and from **void***, C is a programming language with a relatively rigid type system that can provide very useful diagnostics during compilation, if expected and presented types don't match. This rigidity can be, on the other hand, quite constraining when programming general features or algorithms that potentially can apply to a whole set of types, be they pre-defined by the C standard or provided by applications.

This is probably the main reason, why C has no well established general purpose libraries for algorithmic extensions; the interfaces (**bsearch** and **qsort**) that the C library provides are quite rudimentary. By using pointer conversions to **void*** they circumvent exactly the type safety that would be critical for a safe and secure usage of such generic features.

To our knowledge, libraries that provide type-generic features only have a relatively restricted market penetration. In general, they are tedious to implement and to maintain and the interfaces they provide to their users may place quite a burden of consistency checks to these users.

On the other hand, some extensions in C implementations and in related programming languages have emerged that provide type-genericity in a much more comfortable way. At the same time these extensions improve the type-safety of the interfaces and libraries that are coded with them.

An important feature that is proposed here, again, are lambdas. WG14 had talked about them already at several occasions [Garst 2010; Crowl 2010; Hedquist 2016; Garst 2016; Gustedt 2020b], and one reason why their integration in one form or another did not find consensus in 2010 seems to be that, at that time, it had been considered to be too late for C11. An important data point for lambdas is also that within C++ that feature has much evolved since C++11; they have become an important feature in every-day code (not only for C++ but many other programming languages) and their usability has much improved. Thus we think that it is time to reconsider them for integration into C23, which is our first opportunity to add such a new feature to C since C11.

The goal of this paper is to provide an argumentation to integrate some of the existing extensions into the C programming language, such that we can provide interfaces that

— are type and qualifier safe;
— are comfortable to use as if they were just simple functions;
— are comfortable to implement without excessive case analysis.

It provides the introduction to four other papers that introduce different aspects of such a future approach for type-generic programming in C. Most of the features already have been proposed in [Gustedt 2020b] and the intent of these four papers is to make concrete proposals to WG14 for the addition of these features, namely

(1) type inference for variable definitions and function returns,
(2) simple lambdas,
(3) type-generic lambdas,
(4) lvalue closures.

Additionally, we also anticipate that the **typeof** feature as proposed by a fifth paper [Meneide 2020], should be integrated into C.

This paper is organized as follows. Below, Section II, we will briefly present these five papers in subsections of their own. In Section III, we will discuss in more detail the 8 features in the C standard that already provide type-genericity. Section IV then discusses the major problems that current type-generic programming in C faces and the missing properties that we would like to achieve with the proposed extensions. Then, Section V introduces the extension that could close the gaps and shows examples of type-generic code using them, and Section VI provides the combinations of all wordings that are proposed by the four papers in the series.

II. A LEVELED SPECIFICATION OF NEW FEATURES

In the following we briefly present the five papers that should be proposed for C23. The first (Section II.1) and the fifth (Section II.5) handle two forms of type inference. The first uses inference from evaluated expressions that undergo lvalue conversion, array-to-pointer and function-to-pointer decay. The fifth uses direct inference from a wider range of features, namely identifiers, type names and expressions, without performing any form of conversion. These two papers should each be independent from all the others, with the notable thematic connection about type inference between them.

The second paper, Section II.2, introduces a simple version of C++'s lambda feature. In its proposed form it builds on II.1 for (lack of) the specification of return types, but this dependency could be circumvented by adding additional C++ syntax for the specification of return types.

Paper II.3 builds on II.1 and II.2 to provide quite powerful type-generic lambdas.

As an extension of the features proposed in [Gustedt 2020b], paper II.4 builds on II.2 to provide full access to automatic variables from within a lambda.

II.1. Type inference for variable definitions (auto feature) and function return

C's declaration syntax currently already allows to omit the type in a variable definition, as long as the variable is initialized and a storage initializer (such as **auto** or **static**) disambiguates the construct from an assignment. In previous versions of C the interpretation of such a definition had been to attribute the type **int**; in current C this is a constraint violation. We will propose to align C with C++, here, and to change this such the type of the variable is inferred the type from the initializer expression. In a second alignment with C++ we will also propose to extend this notion of **auto** type inference to function return types, namely such that such a return type can be deduced from **return** statements or be **void** if there is none.

II.2. Simple lambdas: function literals and value closures

Since 2011, C++ has a very useful feature called lambdas. These are function-like expressions that can be defined and called at the same point of a program. The simple lambdas that are introduced in this paper are of two kind. We call the first *function literals*, that are lambdas that interact with their context only via the arguments to a call, no automatic variables of the context can be evaluated within the function body. If they are not used in a function call such function literals can be converted to function pointers with the corresponding prototype. The concept is extended with *value closures*, namely lambdas that can access all or part of their context, but by evaluating automatic variables (in a so-called *capture*) at the same point where the lambda as a whole is evaluated. The return type of any such lambda is not provided by the interface specification but it is deduced from the arguments to a possible call.

II.3. Type-generic lambdas (with auto parameters)

Type-generic lambdas extend the lambda feature such that the parameter types can use the **auto** feature and thus be underspecified. This allows lambdas to be a much more general tool and eases the programming of type-generic features. The concrete types of the **auto** parameters for a specific instance of such a lambda are deduced either from the arguments if the lambda is used in a function call, or from the target type of a lambda-to-function-pointer conversion.

II.4. Lvalue closures (pseudo-references for captures)

This paper also introduces C++'s syntax to access automatic variables from within the body of a lambda and eventually modify the variable. C does not have the concept of references to which this feature refers in C++, and the intent of this paper is not to introduce references into C. Therefore we introduce the feature as *lvalue capture* (in contrast to value capture) and just refer to the identifiers that name automatic variables and to the possible lvalue conversion while calling the lambda.

II.5. Type inference from identifiers, value expressions and type expressions

Our hope is that the attempts to integrate gcc's **typeof** extension will be successful. We think that a **typeof** operator that has similar syntactic properties as the **sizeof** and **alignof** operators and that maintains all type properties such as qualification and derivation (atomic, array, pointer or function) could be quite useful for type-generic programming and its type safety.

III. EXISTING TYPE-GENERIC FEATURES IN C

Type-generic features are so deeply integrated into C that most programmers are probably not even aware of there omnipresence. Below we identify eight different features that do indeed provide type-genericity, ranging from simple features, such as operators that work for multiple types, to complicated programmable features, such as generic primary expressions (**_Generic**).

The following discussion is not meant to cover all aspects of existing type-generic features, but to raise awareness for their omnipresence, for their relative complexity, and for their possible defects.

III.1. Operators

The first type-generic feature of C are operators. For example the binary operators == and != are defined for all wide integer types (**signed**, **long**, **long long** and their unsigned variants), for all floating types (**float**, **double**, **long double** and their complex variants) and for pointer types, see Tab. I for more details.

Table I. Permitted types for binary operators that require equal types

		floating		pointer		
				object		
operator	wide integer	real	complex	complete	**void**	function
==, !=	×	×	×	×	×	×
-	×	×	×	×		
+, *, /	×	×	×			
<, <=, >=, >	×	×		×		
%, ^, &, |	×					

Thus, expressions of the form a*b+c are by themselves already type-generic and the programmer does not have to be aware of the particular type of any of the operands. In addition, if the types of the operands do not agree, there is a complicated set of conversions (see below) that enforces equal types for all these operations. Other binary operators (namely shift

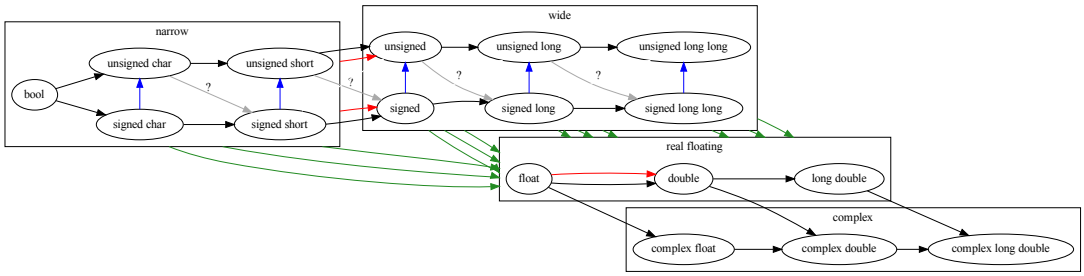

Fig. 1. Upward conversion of arithmetic types. Black arrows conserve values, red arrows may occur for integer promotion or default argument conversion, blue arrows are reduction modulo 2^N, well-definition of grey arrows depends on the platform, green arrows may loose precision

operators, object pointer addition, array subscripting) can even deal with different operand types, even without conversion.

III.2. Default promotions and conversions

If operands for the operators in Tab. I don't agree, or if they are even types for which these operands are not supported (narrow integer types such as **bool**, **char** or **short**) a complicated set of so-called promotion and conversion rules are set in motion. See Fig. 1 for an overview.

III.2.1. Conversions. Whenever an arithmetic argument to a function or the LHS of an assignment or initialization has not the requested type of the corresponding parameter, there is a whole rule set that provides a conversion from the argument type to the parameter type.

```
1   printf("result␣is:␣%g\n", cosf(1));
```

Here, the **cosf** function has a **float** parameter and so the **int** argument 1 is first converted to `1.0f`.

Figure 1 shows the *upward* conversions that are put in place by C. These kind of conversions help to avoid to write several versions of the same function and allow to use such a function, to a certain extend, with several argument types.

III.2.2. Promotion and default argument conversion. In the above example, the result of **cosf** is **float**, too, but **printf** as a variadic function cannot handle a **float**. So that value is converted to **double** before being printed.

Generally, there are certain types of numbers that are not used for arithmetic operators or for certain types of function calls, but are always replaced by a wider type. These mechanisms are called *promotion* (for integer types) or *default argument conversion* (for floating point).

III.2.3. Default arithmetic conversion. To determine the target type of an arithmetic operation, these concepts are taken on a second level. *Default arithmetic conversion* determines a common "super" type for binary arithmetic operators. For example, an operation `-1 + 1U` first performs the minus operation to provide a **signed int** of value -1, then (for arithmetic

conversion) converts that value to an **unsigned int** (with value **UINT_MAX**) and performs the addition. The result is an **unsigned int** of value 0.

III.3. Macros

C's preprocessor has a powerful macro feature that is designed to replace identifiers (so-called object macros) and pseudo-function calls by other token sequences. Together with default arithmetic promotions it can be used to provide type-generic programming for several categories of tasks:

— type-generic expressions
— type-generic declarations and definitions
— type-generic statements that are not expressions

III.3.1. Macros for type-generic expressions. A typically type-generic macro has an arithmetic expression that is evaluated and that uses default arithmetic conversion to determine a target type. For example the following macro computes a grey value from three color channels:

```
1   #define GREY(R, G, B) (((R) + (G) + (B))/3)
```

It can be used for any type that would be used to represent colors. If used with **unsigned char** the result would typically be **int**, for **float** values the result would also be **float**.

Naming conventions, here for structure members `r`, `g`, and `b`, can also help to write type generic macros.

```
1  #define red(P) (P.r)
2  #define green(P) (P.g)
3  #define blue(P) (P.b)
4  #define grey(P) (GREY(P.r, P.g, P.b))
```

III.3.2. Macros for declarations and definitions. Type defitions that then can use the above macros can also be provided by macros.

```
1  #define declareColor(N) typedef struct N N
2
3  declareColor(color8);
4  declareColor(color64);
5  declareColor(colorF);
6  declareColor(colorD);
7
8
9  #define defineColor(N, T) struct N { T r; T g; T b; }
10
11 defineColor(color8, uint8_t);
12 defineColor(color64, uint64_t);
13 defineColor(colorF, float);
14 defineColor(colorD, double);
```

III.3.3. Macros placeable as statements. Macros can also be used to group together several statements for which no value return is expected. Unfortunately, coding properly with this

technique usually has to trade in some uglyness and maintenance suffering. The following presents common practice for generic macro programming in C that can be used for any structure type T that has a **mtx_t** member mut and a data member that is assignment compatible with BASE.

```
#define dataCondStore(T, BASE, P, E, D)      \
  do {                                       \
    T* _pr_p = (P);                          \
    BASE _pr_expected = (E);                 \
    BASE _pr_desired = (D);                  \
    bool _pr_c;                              \
    do {                                     \
      mtx_lock(&_pr_p->mtx);                 \
      _pr_c = (_pr_p->data == _pr_expected); \
      if (_pr_c) _pr_p->data = _pr_desired;  \
      mtx_unlock(&_pr_p->mtx);               \
    }  while(!_pr_c);                        \
  } while (false)
```

Coded like that, the macro has several advantages:

— It can syntactically be used in the same places as a **void** function. This is achieved by the crude outer **do** ... **while**(**false**) loop.
— Macro parameters are evaluated at most once. This is achieved by declaring auxiliary variables to evaluate and hold the values of the macro arguments. Note that the definition of these auxiliary variables needs knowledge about the types T and BASE.
— Some additional auxiliary variables (here _pr_c) can be bound to the scope of the macro.

Additionally, a naming convention for local variables is used as to minimize possible naming conflicts with identifiers that might already be defined in the context where the macro is used. Nevertheless, such a naming convention is not fool proof. In particular, if the use of several such macros is nested, surprising interactions between them may occur.

III.4. Variadic functions

Above we also have seen another C standard tool for type-generic interfaces, variadic functions such as **printf**:

```
int printf(char const format[static 1], ...);
```

The ... denotes an arbitrary list of arguments that can be passed to the function, and it is mostly up to a convention between the implementor and the user how many and what type of arguments a call to the function may receive. There are notable exceptions, though, because with the ... notation all arguments that are narrow integers or are **float** are converted, see Figure 1.

For such interfaces in the C standard library modern compilers can usually check the arguments against the format string. In contrast to that, user specified functions remain usually unchecked and can present serious safety problems.

III.5. function pointers

Function pointers allow to handle algorithms that can be made dependent of another function. For example, here is a generic function that computes an approximation of the derivative of function `func` in point `x`:

```
typedef double math_f(double);

inline double tangent5(math_f* func, double x, double ε) {
  double h = ε * x;
  return (-func(x + 2*h) +8*func(x + h)
          -8*func(x - h) +func(x - 2*h))/(12*h);
}
```

III.6. void pointers

The C library itself has some interfaces that use function pointers for type-genericity, namely **bsearch** and **qsort** receive a function pointer to the following function type

```
typedef int compar_t(void const*, void const*);
```

with the understanding that the pointer parameters of such a function represent pointers to the same object type `BASE`, depending on the function, and that the return value is less than, equal to, or greater than `0` if the first argument compares less than, equivalent to, or greater than the second argument.

```
int comparDouble(void const* A, void const* B){
  double const* a = A;
  double const* b = B;
  return (*a < *b) ? -1 : ((*a == *b) ? 0 : +1);
}

double tabd[] = { 1, 4, 2, 3, };
qsort(tab, sizeof tabd[0], sizeof tab/sizeof tabd[0],
    comparDouble);
```

This uses the fact that data pointers can be converted forth and back to **void** pointers, as long as the target qualification is respected. The advantage is that such a comparison (and thus search or sorting) interface can then be written quickly. The disadvantage is that guaranteeing type safety is solely the job of the user.

III.7. Type-generic C library functions

C gained its first explicit type-generic library interface with the introduction of `<tgmath.h>` in C99. The idea here is that a functionality such as the cosine should be presented to the user as a single interface, a type-generic macro **cos**, instead of the three functions **cos**, **cosf** and **cosl** for **double**, **float** or **long double** arguments, respectively.

At least for such one-argument functions the expectation seems to be clear, that such a functionality should return a value of the same type as the argument. In a sense, such type-generic macros are just the extension of C's operators (which are type-generic) to a set of well specified and understood functions. An important property here is that each of

the type-generic macros in <tgmath.h> represents a finite set of functions in <math.h> or <complex.h>. Many implementations implemented these macros by just choosing a function pointer by inspecting the size of the argument, using the fact that their representations of the argument types all had different sizes.

Then, C11 gained a whole new set of type-generic functions in <stdatomic.h>. The difficulty here is that there is a possibly unbounded number of atomic types, some of which with equal size but different semantics, and so the type-generic interfaces cannot simply rely on the argument size to map to a finite set of functions. Implementations generally have to rely on language extensions to implement these interfaces.

III.8. `_Generic` primary expressions

C11 introduced a new feature, generic primary expressions, that was primarily meant to implement type generic macros similar to those in <tgmath.h>, that is to perform a choice of a limited set of possibilities, guided by the type of an input expression. By that our example for **cos** from above could be implemented as follows:

```
1 #define cos(X)                          \
2   _Generic((X),                         \
3     float: cosf,                        \
4     long double: cosl,                  \
5     default: cos)(X)
```

That is a **_Generic** expression is used to choose a function pointer that is then applied to the argument X. Note that here **_Generic** only uses X for its type and does not evaluate it, that the result type of the **_Generic** is the type of the chosen expression, and, that the library function **cos** can be used within the macro, because C macros are not recursive. Thus, this technique allows an "overload" of some sort of the function **cos** with the macro **cos**. Another implementation could be as follows:

```
1 #define cos(X)                           \
2   _Generic((X),                          \
3     float: cosf((float)X),               \
4     long double: cosl((long double)X), \
5     default: cos((double)X))
```

By this, **cosf** and **cosl** themselves could even be macros and the compiler would not have to use the corresponding function pointers.

The concept of generic primary expressions goes much further than for switching between different function pointers. For example, the following can do a conversion of a pointer value P according to the type of an additional argument X.

```
1 #define getOrderCP(X, P)                                  \
2   _Generic((X),                                           \
3     float: (float const*)(P),                             \
4     double: (double const*)(P),                           \
5     long double: (long double const*)(P),                 \
6     unsigned: (unsigned const*)(P),                       \
7     unsigned long: (unsigned long const*)(P),             \
8     ... /* other ordered arithmetic types */ ...          \
9     )
```

Still, the important concepts are the same: `X` is only used for its type, and the type of the expression itself corresponds to the type of the choosen expression.

IV. MISSING FEATURES

IV.1. Temporary variables of inferred type

One of the most important restrictions for type-generic statements above (III.3.3) was that the macro needed arguments that encoded the types for which the macro was evaluated. This not only inconvenient for the user of these macros but also an important source of errors. If the user chooses the wrong type, implicit conversions can impede on the correctness of the macro. For our example `dataCondStore` a wrong choice of the type `BASE` **`float`** instead of **`double`** could for example have the effect that the equality test never triggers, and thus that the inner loop never terminates.

In accordance with C's syntax for declarations and in extension of its semantics, C++ has a feature that allows to infer the type of a variable from its initializer expression.

```
1  auto y = cos(x);
```

This eases the use of type-generic functions because now the return value **and** type can be captured in an auxiliary variable, without necessarily having the type of the argument, here `x`, at hand. This can become even more interesting if the return type of type-generic functions is just an aggregation of several values for which the type itself is just an artefact:

```
1  #define div(X, Y)            \
2    _Generix((X)+(Y),          \
3             int: div,         \
4             long: ldiv,       \
5             long long: lldiv) \
6             ((X), (Y))
7
8    auto res = div(38484848448, 448484844);  // int or long?
9    auto a = b * res.quot + res.rem;
```

Used in the macro from III.3.3, this can easily remove the need for the specification of the types `T` and `BASE`:

```
1  #define dataCondStoreTG(P, E, D)                \
2    do {                                          \
3      auto* _pr_p = (P);                          \
4      auto _pr_expected = (E);                    \
5      auto _pr_desired = (D);                     \
6      bool _pr_c;                                 \
7      do {                                        \
8        mtx_lock(&_pr_p->mtx);                    \
9        _pr_c = (_pr_p->data == _pr_expected);    \
10       if (_pr_c) _pr_p->data = _pr_desired;     \
11       mtx_unlock(&_pr_p->mtx);                  \
12     }  while(!_pr_c);                           \
13   } while (false)
```

IV.2. Controlled encapsulation

Even as presented now, the macro `dataCondStoreTG` has a serious flaw that is not as apparent as it should be. The assignment of the values of `E` and `D` to `_pr_expected` and `_pr_desired` is not independent. This is, because `D` itself may be an expression that contains a reference to an identifier `_pr_expected`, and thus the intended evaluation of `D` (before even entering the macro) is never performed, but a completely different value (depending on `E`) is used instead.

```
1   dataCondStoreTG(P, 4, 3*_pr_expected);
```

The result of the macro then depends on the order of specification of the variables `_pr_expected` and `_pr_desired`. This kind of interaction is the main reason why we had to chose these ugly names with a `_pr_` prefix in the first place: they reduce the probability of interaction between the code inside the macro and its caller.

C++ has a feature that is called *lambda*. In its simplest form (that we call *function literal*) it provides just the possibility to specify an anonymous function that only interacts with its context via parameters:

```
1  auto const dataCondStoreλDD =
2    [](DD *p, double expected, double desired) {
3      bool c;
4      do {
5        mtx_lock(&p->mtx);
6        c = (p->data == expected);
7        if (c) p->data = desired;
8        mtx_unlock(&p->mtx);
9      } while(!c);
10   };
11
12   dataCondStoreλDD(pDD, 0.5, 0.7);
```

Here, we may now chose "decent" variable and parameter names, because we know that they will not interact with a calling context.

When we combine lambdas with the **auto** feature for the parameters, this tool becomes even more powerful, because now we have in fact a way to describe a type-generic functionality without having to worry about the particular types of the arguments nor of an uncontrolled interaction with the calling environment.

```
1  #define dataCondStoreλ                          \
2    [](auto *p, auto expected, auto desired) { \
3      bool c;                                   \
4      do {                                      \
5        mtx_lock(&p->mtx);                      \
6        c = (p->data == expected);              \
7        if (c) p->data = desired;               \
8        mtx_unlock(&p->mtx);                    \
9      } while(!c);                              \
```

```
10    }
11
12    dataCondStoreλ(pDD, 0.5, 0.7);
13    dataCondStoreλ(pFF, 0.1f, 0);
```

IV.3. Controlled constant propagation

The above form of lambdas for function literals is introduced by an empty pair of brackets `[]` to indicate that the lambda does not access to any automatic variables from the calling context. More general forms of lambdas called *closures* are available in C++ that provide access to the calling context.

The idea is that the body of a closure may use identifiers that are *free*, that is that don't have a definition that is provided by the lambda itself but by the calling context. C++ has a strict policy here, that such free variables must be explicitly named within the brackets, or that the bracket should have a `=` token to allow any such free variables to appear. For example a lambda expression as in the following

```
1  auto const tangent5λ = [ε](math_f* func, double x) {
2    double h = ε * x;
3    return (-func(x + 2*h) +8*func(x + h)
4            -8*func(x - h) +func(x - 2*h))/(12*h);
5  };
```

captures the value ε from the environment and freezes it for any use of the `tangent5λ` closure to the value at the point of evaluation of the lambda (and not the call).

An even more extended form of this allows the assignment of any expression to the free variables:

```
1  #define TANGENT5(F, E) [func = (F), ε = (E)](double x) { \
2    double h = ε * x;                                      \
3    return (-func(x + 2*h) +8*func(x + h)                  \
4            -8*func(x - h) +func(x - 2*h))/(12*h);         \
5  }
6
7  int main(int argc, char* argv[static argc+1]) {
8    auto const f0 = argc > 1 ? &sin : &cos; // function pointer
9    auto const f1 = TANGENT5(f0, 0x1E-12);  // lambda value
10   auto const f2 = TANGENT5(f1, 0x1E-12);  // lambda value
11   auto const f3 = TANGENT5(f2, 0x1E-12);  // lambda value
12
13   for (double x = 0.01; x < 4; x += 0.5) {
14     printf("%g␣%g␣%g␣%g\n", f0(x), f1(x), f2(x), f3(x));
15   }
16 }
```

Here, three lambdas are evaluated and assigned to **auto** variables `f1`, `f2` and `f3`, respectively. By that technique, the compiler is free to optimize the code in the body of the lambda with respect to the possible values of `func` and ε, and then to use these optimized versions within the **for** loop as indicated.

IV.4. Automatic instantiation of function pointers

Library programmers often need a seamless tool to describe and implement a generic feature, and, from time to time, they need the possibility to instantiate a function pointer for a certain set of function arguments from there. **_Generic** provides the complete opposite of that: previously unrelated specialized function pointers are stitched together into one feature.

C++'s lambda model allows to provide such a more practical tool, namely it allows to instantiate function pointers from all function literals.

```
1  auto const sortDouble =
2    // function literal
3    [](size_t len, double const ar[static len]) {
4        // function pointer
5        int (*comp)(void const*, void const*) =
6          // function literal
7          [](void const* A, void const* B){
8            double const* a = A;
9            double const* b = B;
10           // returns -1, 0, or +1, an int
11           return (*a < *b) ? -1 : ((*a == *b) ? 0 : +1);
12         };
13         qsort(ar, sizeof ar[0], len, comp);
14       );
15       // no return statement, void
16   };
17
18 double tabd[] = { 1, 4, 2, 3, };
19 sortDouble(sizeof tab/sizeof tabd[0], tabd);
```

That is, all lambdas without capture can be converted implicitly or explicitly to a function pointer with a prototype that is compatible with the parameter and return types of the lambda. If such an attempt is made and the parameter types are not compatible, an error (constraint violation) occurs and the compilation should abort. In the above example the inner lambda has two parameters of type **void const*** and its **return** expression has type **int**. Thus its lambda type is convertible to the function pointer type as indicated.

Such a conversion to a function pointer can be done implicitly as above, in an intialization, assignment or by passing a lambda as an argument to a function call. It can also come from an explicit conversion, that is a cast operator.

IV.5. Automatic instantiation of specializations

When the parameters of a lambda use the **auto** feature, we have a *type-generic* lambda, that is a lambda that can receive different types of parameters. When such a lambda is used, the underspecified parameter types must be completed, such that the compiler can instantiate code that has all types fixed at compile time.

If there are no captures, one possibility to determine the parameter types is to assign such a type-generic lambda to a function pointer:

```
1 #define TANGENT5TG(auto* func, auto x, auto ε) {  \
2   auto h = ε * x;                                 \
```

```
  return (-func(x + 2*h) +8*func(x + h)           \
          -8*func(x - h) +func(x - 2*h))/(12*h);  \
}

typedef double math_f(double);
typedef float mathf_f(float);
typedef long double mathl_f(long double);


double (*tangent5)(math_f*, double, double) = TANGENT5TG;
float (*tangent5f)(mathf_f*, float, float) = TANGENT5TG;
long double (*tangent5l)(mathl_f*, long double, long double) =
   TANGENT5TG;
```

Here, again, such a conversion to a function pointer can only be formed if the parameter and return types can be made consistent.

The following shows how an inner lambda can even be made type-generic, such that it synthesizes a function pointer on the fly, whenever the outer lambda is instantiated:

```
#define sortOrder                                          \
    [](size_t len, auto const ar[static len]) {            \
       qsort(ar, sizeof ar[0], len,                        \
          [](void const* A, void const* B){                \
            auto const* a = getOrderCP(ar[0], A);          \
            auto const* B = getOrderCP(ar[0], B);          \
            return (*a < *b) ? -1 : ((*a == *b) ? 0 : +1); \
          }                                                \
       );                                                  \
    }

  void (*sortd)(size_t len, double const ar[static len])
    = sortOrder;
  void (*sortu)(size_t len, unsigned const ar[static len])
    = sortOrder;

  double tabd[] = { 1, 4, 2, 3, };
  // semantically equivalent
  sortOrder(sizeof tab/sizeof tabd[0], tabd);
  sortd(sizeof tab/sizeof tabd[0], tabd);

  unsigned tabu[] = { 1, 4, 2, 3, };
  // semantically equivalent
  sortOrder(sizeof tabu/sizeof tabu[0], tabu);
  sortu(sizeof tabu/sizeof tabu[0], tabu);
```

Here, we use the type-generic macro `getOrderCP` from above which does not evaluate its first argument, `ar[0]` in this case, but only uses it for its type. Remember that the visiblity rules for identifiers from outer scopes are the same as elsewhere, only the *access* to automatic variables is constrained or allowed by the capture clause. Thus, such a use for the type inside the inner lambda is allowed, and provides a lambda that is dependent on the type of `ar[0]`.

IV.6. Direct type inferrence

The possibility of inferring a type via the **auto** feature has the property that it is only possible for an expression that is evaluated in an initializer, and thus it first undergoes lvalue, array-to-pointer or function-to-pointer conversion before the type is determined. In particular, by this mechanism it is not possible to propagate qualifiers (including **_Atomic**) nor to conserve array dimensions.

C++ has the **decltype** operator and many C compilers have a **__typeof__** extension that fills this gap. For the following we assume a **typeof** operator that just captures the type of an expression or typename that is passed as an argument.

```
1  int i;
2  // an array of three int
3  typeof(i) iA[] = { 0, 8, 9, };
4
5  double A[4];
6  typedef typeof(A) typeA;
7  // equivalent definition
8  typedef double typeA[4];
9  // equivalent declaration
10 typeA A;
11 // equivalent declaration
12 typeof(double[4]) A;
13 // mutable array of 4 elements intialized to 0
14 typeof(A) dA = { 0 };
15 // immutable array of 4 elements
16 typeof(A) const cA = { 0, 1, 2, 3, };
17
18 // infer the type of a function
19 typeof(sin) cos;
20 // equivalent declaration
21 double cos(double);
22
23 // infer the type of a function pointer and initialize
24 typeof(sin)*const Δ = cos;
25 // equivalent definition
26 auto*const Δ = cos;
27 // equivalent definition
28 const auto Δ = cos;
29 // equivalent definition
30 double (*const Δ)(double) = cos;
```

In particular, for every declared identifier `id` with external linkage (that is not also thread local) the following redundant declaration can be placed anywhere where a declaration is allowed.

```
1  extern typeof(id) id;
```

A **typeof** operator can be used everywhere where an **typedef** identifier can be used. It can not only applied to type expressions and identifiers as above, but also to any valid expression:

```c
#define sortOrder                                                  \
    [](size_t len, auto const ar[static len]) {                    \
       qsort(ar, sizeof ar[0], len,                                \
          [](void const* A, void const* B){                        \
            typeof(ar[0])* a = A;                                  \
            typeof(ar[0])* b = B;                                  \
            return (*a < *b) ? -1 : ((*a == *b) ? 0 : +1); \
          }                                                        \
       );                                                          \
    }

  void (*sortd)(size_t len, double const ar[static len])
    = sortOrder;
  void (*sortu)(size_t len, unsigned const ar[static len])
    = sortOrder;

  double tabd[] = { 1, 4, 2, 3, };
  // semantically equivalent
  sortOrder(sizeof tab/sizeof tabd[0], tabd);
  sortd(sizeof tab/sizeof tabd[0], tabd);

  unsigned tabu[] = { 1, 4, 2, 3, };
  // semantically equivalent
  sortOrder(sizeof tabu/sizeof tabu[0], tabu);
  sortu(sizeof tabu/sizeof tabu[0], tabu);
```

By that we are now able to remove the call to `getOrderCP` from the inner lambda expression. The result is a macro `sortOrder` that can be used to sort any array as long as the elements that can be compared with the `<` operator. The only external reference that remains is the `C` library function **`qsort`**. That macro can be used to instantiate a function pointer or it can be used directly in a function call.

V. COMMON EXTENSIONS IN C IMPLEMENTATIONS AND IN OTHER RELATED PROGRAMMING LANGUAGES

In the following we are interested in features that extend current C for type-genericity but with one important restriction:

Features that are proposed imply no ABI changes.

In particular, with the proposed changes we do not intend

— to change the ABI for function pointers,
— to introduce linkage incompatibilities such as mangling,
— to modify the life-time of automatic objects, or
— to introduce other managed storage that is different from automatic storage.

There are a lot of features in the field that would need one or several points from the above, such as `C++`'s `template` functions or functor classes, Objective `C`'s `__block` storage specifiers, or `gcc`'s callable nested functions. All of these approaches have their merits, and this paper is not written to argue against their integration into `C`. We simply try first to look into the

features that can do without, such that they might be easily adopted by programmers that are used to our concepts and implemented more widely than they already are.

V.1. Type inferrence

Besides the possibility of functional expression, declaring parameters, variables and return values of inferred type is a crucial missing feature for an enhancement of standard C towards type-genericity. This allows to declare local, auxiliary, variables of a type that is deduced from parameters and to return dependent values and types from functional constructs.

We found several existing extensions in C or related languages that allow to infer a type from a given construct. They differ in the way derived type constructions (qualifiers, **_Atomic**, arrays or functions) influence the derived type: C++'s **auto** feature and gcc's **auto_type**, C++'s **decltype**, and gcc's **typeof**.

V.1.1. **auto** *type inference.* This kind of type inference takes up an idea that already exists in C:

> *A type specification may only have incomplete information, and then is completed by an initializer.*

This is currently possible for array declarations where an incomplete specification of an array bound may be completed by an initializer:

```
double const A[] = { 5. 6, 7, };  // array of 3 elements
double const B[] = { [23] = 0, }; // array of 24 zeroes
```

In fact, the maximum index in the initializer determines the size of the array and thereby completes the array type.

auto type inference pushes this further, such that also the base type of an object definition can be inferred from the initializer:

```
auto b = B[0]; // this is double
auto a = A;    // this is double const*
```

Here, the initializer is considered to be an *expression*, thus all rules for evaluation of expressions apply. So, qualifiers and some type derivations are dropped. For example, b is **double**, the **const** is dropped, and A on the RHS undergoes array-to-pointer conversion and the inferred type for a is **double const*** and not **double const**[24].

Since in the places that are interesting here = can have the meaning of an assignment operator or of an initializer, constructs as the following could be ambiguous:

```
b = B[0];
a = A;
```

This ambiguity can occur as soon that an attempted declaration has no storage class, therefore C++ extends the use of the keyword **auto** and allows to place it in any declaration that is supposed to be completed by an initializer.

This feature is then extended even further into contexts that don't even have initializers:

— An **auto** declaration of a function return type infers the completed return type from a **return** expression, if there is any, or infers a type of **void**, if there is none.
— An **auto** declaration of a function or lambda parameter infers the completed parameter type from the argument to a function call or from the corresponding parameter in a function-pointer conversion.

V.1.2. The **typeof** *feature.* **typeof** is an extension that has been provided since a long time in multiple compilers. A **typeof** specifier is just a placeholder for a type, similar to a **typedef**. It reproduces the type "as-is" without dropping qualifiers and without decaying functions or arrays. With this feature not only qualifiers and atomics do not get dropped, but they can even be added.

It differs (and complements) the **auto** feature syntactically and semantically. Its general forms are

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

and these can be substituted at any place where a type name may occur. With the definitions of `A` and `B` as above

```
auto b = B[0];                 // this is double
auto a = A;                    // this is double const*
typeof(B[0]) β;                // this is double const
typeof(A) α;                   // this is double const[24]
typeof(double const[24]) γ; // same type
```

So here we see that the expressions `B[0]` and `A` do not undergo any conversion and so the qualifier and the array derivation remain in place.

There have been some inconsistencies for the type derivation strategies for this operator in the past, but it seems that recent compilers interpret types that are given as arguments as it is presented above.

V.1.3. The **decltype** *feature.* Since almost a decade C++ has introduced the **decltype** feature which in most aspects that concern the intersection with `C` is similar to **typeof**.

Conceptually, integration into C would be a bit more difficult than for **auto**. This is because for historic reasons `C++` here mixes several concepts in an unfortunate way: for some types of expressions **decltype** has a reference type for others it hasn't. The line of when it does this is not where we would expect it to be for `C`: most lvalues produce a reference type, but not all of them. In particular, direct identification of variables or functions (by identifier) or of structure or union members leads to direct types, without reference, but surrounding them with an expression that conserves their "lvalueness" adds a reference to the type of the **decltype** specification.

It is quite unusual for `C` to have the type of an expression depend on surrounding `()`, but unfortunately that ship has sailed in `C++`. Therefore we prefer that a new operator **typeof** be introduced into both languages that clarifies these aspects and that is designed to have exactly the same properties in both.

V.2. Lambdas

As we have seen above, in C macros can serve for two important type-generic tasks, namely the specification of type-generic expressions and the specification of type-generic functions. But unfortunately they cannot, without extension, be used *in place* to specify functional units that use the whole expressiveness of the language to do their computation.

To illustrate that, consider the simple task of specifying a **max** feature that computes the maximum of two values x and y. In essence, we would like this to compute the expression

```
(x < y ? y : x)
```

regardless of the type of the two values x and y. As such this is not possible to specify this safely with a macro

```
#define BADMAX(X, Y) ((X) < (Y) ? (Y) : (X))
```

because such a macro always evaluates one of the argument twice; once in the comparison and a second time to evaluate the chosen value. As soon as we pass in argument expressions that have side effects (such as i++ or a function call) these effects could be produced twice and therefore result in surprising behavior for the unaware user of the interface.

Also, when we would mix signed and unsigned arguments, the above formula would not always compute the mathematical maximum value of the two arguments because a negative signed value could be converted to a large positive unsigned value.

Thus, already for a simple type-generic feature such as **max**, we would need the possibility to define local variables that only have the scope of the **max** expression, and for which we may somehow infer the type from the arguments that are passed to **max**.

In a slight abuse of terminology we will borough the term *lambda* from the domain of functional programming to describe a functional feature that is an expression with a *lambda value* of *lambda type*. Several proposals have already been discussed to integrate lambdas into C [Garst 2010; Crowl 2010; Hedquist 2016; Garst 2016].

Basically, a lambda value can be used in two ways

— It can be moved around as values of objects, that is assigned to variables or returned from functions.
— It can replace the function specifier in a function call expression.

In C++'s lambda notation (that we will propose to adopt below) a **max** feature can be implemented as follows

```
[](auto x, auto y) {
  if ((x < 0) != (y < 0)) {
    x = (x < 0) ? 0 : x;
    y = (y < 0) ? 0 : y;
  }
  return (x < y ? y : x);
}
```

That is, [] introduces a lambda expression, x and y are parameters to the lambda that have an underspecified type (indicated by **auto**) and a **return** statement in the body of the

lambda specifies a return value and, implicitly, a return type. The logic of the **if** statement is to capture the case where one of the two parameters is negative and the other is not, and then to replace the negative one with the value zero. Thereby the lambda never converts a negative signed value to a positive unsigned value.

Observe, that this lambda does not access any other identifier than its parameters.

Global identifiers are easy to handle by lambdas as they are handled by any traditional C function. For these there are two mechanism in play:

> *visibility.* This regulates which identifiers can be used and which type they have. In particular, visible identifiers can be used in some context (such as **sizeof** or **_Generic**) without being accessed.
> *linkage.* This regulates how the object or function behind an identifier is accessible. In particular, an object or function with internal linkage is expected to be instantiated in the same translation unit, and one with external linkage may refer to another, yet unknown, translation unit.

We will call a lambda as the above that does not access external identifiers other than global variables or functions a *function literal*. This term is chosen because such an expression can be used like other literals in C: all information for the lambda value is available at compilation time. Such function literal can be moved freely within the scope of the identifiers that are used.

V.2.1. Possible syntax. There are several possibilities to specify syntax for lambdas and below we will see three such specifications as they are currently implemented in the field:

— C++ lambdas,
— Objective C blocks,
— gcc's statement expressions.

A fourth syntax had been proposed by us in some discussions in WG14, namely to extend the notion of compound literals to function types. Syntactically this could be quite simple: for a compound literal where the type expression is a function specification, the brace-enclosed initializer would be expected to be a *function body*, just as for an ordinary function. The successful presence of gcc's statement expressions as an extension shows that such an addition could be added to C's syntax tree without much difficulties. But these two approaches also share the same insufficiencies, namely the semantic ambiguity how references to local variables of the enclosing function would resolve.

V.2.2. The design space for captures and closures. For an object `id` with automatic storage duration there is currently not much a distinction between the visibility of `id` and the possibility to access the object through `id`. For the current definition of the language this sufficient, but if lambdas are able to refer to identifiers that correspond to objects with automatic storage duration, things become more complicated. For example, we might want to execute a lambda that accesses a local variable `x` in a context where `x` is hidden by another variable with the same name. So lambdas that access local variables must use a different mechanism to do so.

We call lambdas that access identifiers of the context in which they are evaluated, *closures*, and the identifiers that are such accessed by a closure *captures*. Since lambdas are inherently expressions, within the context of C there are several possible interpretations of such a

capture. The design space for modeling the capture of local variables with existing `C` features can be described as follows:

(1) The identifier `id` of type τ is evaluated at the point of evaluation of the capture, and the value ν of type τ' that is determined is used in place throughout the whole lifetime of the closure. Such a capture is called a *value capture*. A closure that has only value captures is called a *value closure*.
If τ would be an array type it would not be copyable (there is no such thing as an array value in `C`) and thus it would not fit well in the scheme of a value capture. Therefore, generally array types (and maybe other, non-copyable, types) are not allowed as value captures.
A value capture can in principle be made visible with three different models as follows. They all have in common that `id` can never appear where a modifiable lvalue is required, such as the LHS of an assignment or as the operand of an increment.

rvalue capture. A value capture `id` can be presented as an "rvalue", that is as if it were defined as the result of an expression evaluation `(0,id)`. The address of a capture in this model cannot be taken. Although this might seem the most natural view for the evaluation of lambda expression in `C`, we are not aware of an implementation that that uses this model.

immutable capture. A value capture `id` is a lambda-local object of type τ'' that is initialized with ν, where τ'' is τ' with an additional **`const`**-qualification. The address of such a capture can be taken and, for example, be passed as argument to a function call. But nevertheless the underlying object cannot be modified.

mutable capture. A value capture `id` is a lambda-local object of type τ' that is initialized with ν. Such a capture behaves very similar to a function parameter that receives the same value as argument on each function call. Such an object is mutable during the execution of the closure, but all changes are lost as soon as control is returned to the calling context.

Note that because τ' is a type after an evaluation, in all these models qualification or atomicity of τ is dropped.

(2) Throughout the life-time of the closure, `id` refers to the same object that is visible by this name at the point of evaluation of the closure. Such a capture is called an *lvalue capture*. A closure that has at least one lvalue capture is called an *lvalue closure*. Since lvalue captures refer to objects, an lvalue closure cannot have a life-time that exceeds any of its lvalue captures. Since `id` is not evaluated at the same time as the lambda expression is formed, it has the same type τ inside the body of the lambda. No qualifiers are dropped, type derivations such as atomic or array are maintained.

V.2.3. C++ lambdas. `C++` lambdas are the most general existing extension and they also fit well into the constraints that we have set ourselves above, namely to be compatible with existing storage classes. Their syntactic form if we don't consider the possibility of adding attributes is

`[` *capture-list* **`]`** $\{$**`(`** *parameter-list* **`)`**$\}_{opt}$ **`mutable`**$_{opt}$ $\{$**`->`** *return-type*$\}_{opt}$ *function-body*

Identifiers with automatic storage duration are captured exclusively if they are listed in the *capture-list* or if a default capture is given. This is a list of captures, each of one the following forms

explicit	
`id`	immutable value capture
`id = `*expression*	immutable value capture with type and value of *expression*
`&id`	lvalue capture
`&id = `*lvalue-expression*	object alias
default	
	forbidden
`=`	immutable value capture
`&`	lvalue capture

If the optional keyword **`mutable`** is present, all captures that would otherwise be immutable value captures are mutable value captures, instead. If `->` *return-type* is present it describes the return type of the lambda; if not, the return type is deduced from a **`return`** statement if there is any, or it is **`void`** otherwise. The *object alias* feature introduces a `C++` reference variable. For `C`, these constructs would need some avoidable extension to the syntax and object semantic, so we will not use these parts of the syntax in the proposed addition to `C`.

The *parameter-list* can be a usual parameter list with the notable extension that the type of a parameter can be underspecified by using the **`auto`** feature, see below. A lambda that has at least one underspecified parameter is a *type-generic* lambda.

Lambda values can be used just as function designators as the left operand of a function call, and all rules for arguments to such a call and the rules to convert them transfers naturally to a lambda call.

When used outside the LHS of a function call expression, lambdas are just values of some object type that is not further specified. Such a lambda type has no declaration syntax, and so the only way to store a lambda value into an object is to use the **`auto`** feature:

```
1  auto const λ = [](double x){ return x+1; };
```

By these precautions, for any `C++` lambda the original expression that defined the value is always known. So the compiler will always master any aspects of the lambda, in particular which variables of the context are used as captures. If a lambda value leaves the scope of definition of any of its lvalue captures the compiler can print a diagnosis.

Function literals are special with respect to these aspects, since they do not have any captures. This is why these special lambdas allow for a third operation, they can be converted to a function pointer:

```
1  double (*λp)(double) = λ;
2  double (*κp)(double) = [](double x){ return x+1; };
```

V.2.4. Objective `C`'s blocks. Objective `C` [ObjectiveC 2014] has a highly evolved lambda feature that they call *block*, see also [Garst 2009; Garst 2016]. Their syntax is

$$\text{`^'}\ \textit{return-type}_{opt}\ \{\texttt{(}\ \textit{parameter-list}\ \texttt{)}\}_{opt}\ \textit{function-body}$$

Besides the obvious syntactic difference, blocks lack an important feature of C++ lambdas, namely the possibility to specify the policy for captures. If used without other specific extensions, an Objective C block has the same semantic as a C++ value closure, where any automatic variable in the surrounding context can be used as immutable value capture. Such a block can be equivalently defined with a C++ lambda as

[=] $\{$**(** *parameter-list* **)**$\}_{opt}$ $\{$**->** *return-type*$\}_{opt}$ *function-body*

and in particular the variants that omit the return type have a syntax that only differs on the token sequence that introduces the feature:

^ $\{$**(** *parameter-list* **)**$\}_{opt}$ *function-body*

[=] $\{$**(** *parameter-list* **)**$\}_{opt}$ *function-body*

An important difference arises though, when it comes to lvalue captures, where Objective C takes a completely different approach than C++. Here, the property if a capture is a value or an lvalue capture is attributed to the underlying variable itself, not to the closure that uses it.

A new storage class for managed storage is introduced, unfortunately also called `__block`; `__block` variables are always lvalue captures. Such variables have a lifetime that is prolonged even after their defining scope is left, as long as there is any living closure that refers to it. By this, blocks elegantly resolve the lifetime issues of lvalue closures in C++: by definition a block will never access a variable after its end-of-life. This elegance comes at the cost of introducing a new storage class with a substantial implementation cost, a certain runtime overhead, and a lack of expressiveness for the choice of the access model for each individual capture.

Because of this extension of the lifetime of lvalue captures, for Objective C it is also much easier to describe functors as variables of block type. The declaration syntax for these is similar to function pointers, but using a ^ token instead of *.

V.2.5. Statement expressions. Statement expressions are an intuitive extension first introduced by the gcc compiler framework. Their basic idea is to surround a compound statement with parenthesis and thereby to transform such a compound statement into an expression. The value of such an expression is the value of the last statement if that is an expression statement, or **void** if it is any other form of statement. With *statements* any list of C statements (including a terminating ; if necessary), the syntax

({ *statements expression*; **})**

is equivalent to the following function call with a C++ lvalue closure as the left operand

[&] (void) { *statements* **return** *expression*; **} ()**

V.2.6. Nested functions. Gcc and related compiler platforms also implement the feature of a *nested function*, that is a function that is declared inside the function body of another function. Obviously, because they are not expressions, nested functions are not lambdas, but we will see below how they can be effectively used to implement lambdas. On the other hand, since they cannot be forward-declared, lambda expressions don't allow for recursion, so nested functions clearly are more expressive.

Nested functions can also capture local variables of the surrounding scope. Because they are not expressions but definitions, the most natural semantic is that of lvalue captures the use of such variables, and this is the semantic that `gcc` applies.

Much as global standard `C` functions, nested functions decay into function pointers if they are used other than for the LHS of a function call. This is even for functions that need access to captures, and thus the ABI must be extended to make this possible. The `gcc` implementation does that by creating a so-called *trampoline* as an automatic object, namely as a small function that collects the local information that is necessary and then calls a conventional function to execute the specified function body. Doing so needs execute rights for the automatic storage in question, which is widely criticized because of its possible security impact. On the other hand, this approach is uncritical when it is used without captures, because then the result of the conversion is a simple, conventional, function pointer.

Provided we have an **`auto`** feature as presented in Section V.1.1 and a **`typeof`** feature as in Section V.1.2, the semantics of a wide variety of `C++` lambdas can be implemented with nested functions. For example, with the `shnell` source-to-source rewriting tool [Gustedt 2020a], we have implemented such a transformation as follows. For a value closure of the form

`[id0 =` *expr0*`, ...,` **`idk =`** *exprk*`]` **`(`** *parameter-list* **`)`** *function-body0*

a definition of a state type **`_Uniq_Struct`**, state variable **`_Uniq_Capt`** and a definition of a local function **`_Uniq_Func`** are placed inside the closest compound statement that contains the lambda expression:

```
struct _Unique_Struct {
  typeof(expr0) id0;
  ...
  typeof(exprk) idk;
} _Uniq_Capt;
auto _Uniq_Func( parameter-list ) function-body1
```

Here, *function-body1* is the same as *function-body0*, only that the contents is prefixed with definitions of the captures:

```
auto const id0 = _Uniq_Capt.id0;
...
auto const idk = _Uniq_Capt.idk;
```

The lambda expression itself then has to be replaced by an expression that evaluates all the expressions to be captured, followed by the name of the function:

`((_Uniq_Capt = (struct _Uniq_Struct){` *expr0, ..., exprk* **`}), _Uniq_Func)`**

Similarly to the above, value captures of the form **`idI`** (without expression) can just use **`idI`** for *exprI*.

Additionally, a `C++` lvalue closure that has either a default **`&`** token or individual lvalue captures **`&idI`** can be implemented by just removing these elements from the capture list. Then, the same restrictions for the lifetime of lvalue captures and lambda values applies to the rewritten code, and it is up to the programmer to verify this property.

Although this approach covers a wide range of C++ lambdas, such a rewriting strategy has some limits:

— The lambda expression cannot be used in all places that are valid for expression. This are for example an initializer for a variable that is not the first declared variable in a declaration or a controlling expression of a **for** loop.
— The default token **=** in the capture list is not implementable by such simple rewriting,
— The function body is not checked for an access of automatic variables that are not listed in the capture clause.

References

Lawrence Crowl. 2010. *Comparing Lambda in C Proposal N1451 and C++ FCD N3092*. Technical Report N1483. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1483.htm.

Blaine Garst. 2009. *Apple's extensions to C*. Technical Report N1370. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1370.pdf.

Blaine Garst. 2010. *Blocks proposal*. Technical Report N1451. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1451.pdf.

Blaine Garst. 2016. *A Closure for C*. Technical Report N2030. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2030.pdf.

Jens Gustedt. 2020a. *C source-to-source compiler enhancement from within*. Research Report RR-9375. INRIA. https://hal.inria.fr/hal-02998412

Jens Gustedt. 2020b. *A Common C/C++ Core Specification, rev. 2*. Technical Report N2522. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2522.pdf.

Barry Hedquist. 2016. *WG14 Minutes, Kona, HI, USA, 26-29 October, 2015*. Technical Report N2093. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2093.pdf.

JeanHeyd Meneide. 2020. *Not-So-Magic - typeof(...) in C*. Technical Report N2593. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2593.htm.

ObjectiveC 2014. Programming with Objective-C. Apple Inc., https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.

VI. PROPOSED WORDING

This is the proposed text for the whole series of papers. It is given as diff against C17. A factored diff for the specific concerns is provided with each individual paper.

— 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

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

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 declarator and that designates an object has a scope that starts at the end of its initializer; if the same identifier declares another entity in an surrounding scope, that declaration is hidden as soon as the inner declarator is met.[31)] An identifier that designates a function with an underspecified return type 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.

Forward references: declarations (6.7), function calls (6.5.2.2), 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`**.

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

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

[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.

— 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 an object type that describes the value of a lambda expression. A complete 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; 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 expression that has underspecified parameters has an incomplete lambda type that can be completed by function call arguments.

— 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*.[50)]

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.

24 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[51)] corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[52)] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

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

[51)] See 6.7.3 regarding qualified array and function types.

[52)] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

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`**.[67)]
>
> 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.[68)]

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;[69)] 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 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 **`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

[67)] 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).

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

[69)] 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.

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 **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 **sizeof** operator,[70] 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 Other than specified in the following, lambda types shall not be converted to any other object type. A function literal with a type "lambda with prototype *type*" can be converted implicitly or explicitly to an expression that has type "pointer to *type*".For a type-generic lambda, types of underspecified parameters shall first be completed according to the parameters of the target prototype; that is, for each underspecified parameter there shall be a type specifier as described in 6.7.10 such that the adjusted parameter type is compatible with the parameter type of the target function type. After that, the inferred return type of the thus completed lambda shall be compatible with the return type of the target prototype.[71] The function pointer value behaves as if a function with internal linkage with the appropriate prototype, a unique name, and the same function body as for λ had been specified in the translation unit and the function pointer had been formed by function-to-pointer conversion of that function. The only differences are that, if λ is not type-generic, the resulting function pointer is the same for the whole program execution whenever a conversion of λ is met[72] and 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), initialization (6.7.9), 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)~~.~~, type inference (6.7.10).

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.

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

[70] 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.

[71] 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.

[72] Thus a function literal that is not type-generic has properties that are similar to a function declared with **static** and **inline**. A possible implementation of the lambda type is to be the the function pointer type to which they convert.

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

`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,[100] 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, or a void expression if its result expression is, respectively, an lvalue, a function designator, or a void expression.

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*

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*".

[100] An lvalue conversion drops type qualifiers.

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[101] 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.[102]

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`**.

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

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

[102] 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.

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.[103)]

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,[104)] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

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

[104)] 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 type as described in 6.2.6 (a process sometimes called “type punning”). This might be a trap representation.

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 (6.7.9).

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-default
capture-list-element
capture-list **,** *capture-list-element*

capture-default:
=
&

capture-list-element:
value-capture
lvalue-capture

value-capture:
capture

capture **=** *assignment-expression*

lvalue-capture:
& *capture*

capture:
identifier

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

Constraints

2 A lambda expression shall not be operand of the unary & operator.[111)]

3 A capture that is listed in the capture list is an *explicit capture* . If the first element in the capture list is a capture default, `id` is the name of an object with automatic storage duration in a surrounding scope, `id` is used within the function body of the lambda without redeclaration and `id` is not an explicit capture or a parameter, the effect is as if `id` were a value capture (for an = token) or a reference capture (for an & token). Such a capture is an *implicit capture* . If the first element in the capture list is an = token, all other elements shall be lvalue captures; if it is an & token, all shall be value captures.

4 Value captures without assignment expression or lvalue captures shall be names of complete objects with automatic storage duration in a scope surrounding the lambda expression that are visible at the point of evaluation of the lambda expression. Additionally, value captures shall not have an array type. An identifier shall appear at most once; either as an explicit capture or as a parameter name in the parameter type list.

5 Within the function body, identifiers (including explicit and implicit captures, and parameters of the lambda) shall be used according to the usual scoping rules, but identifiers of a scope that includes the lambda expression, that are not lvalue captures and that are declared with automatic storage duration shall only be evaluated within the assignment expression of a value capture.[112)]

6 A closure that has an explicit or implicit lvalue capture `id` shall not be used as the expression of a **`return`** statement, unless that **`return`** statement is itself associated to another closure for which `id` is an lvalue capture that refers to the same object.[113)]

7 After determining the type of all captures and parameters the function body shall be such that a return type *type* according to the rules in 6.8.6.4 can be inferred. If the lambda occurs in a conversion to a function pointer, the inferred return type shall be compatible to the specified return type of the function pointer.

Semantics

8 If the parameter clause is omitted, a clause of the form ( ) is assumed. A lambda expression without capture list 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* . A closure that has an lvalue capture is called an *lvalue closure* , otherwise it is a *value closure* .

9 Similar to a function definition, a lambda expression forms a single block scope that comprises its

111) Objects with lambda type that can be operand of the unary & operator can be formed by type inference and initialization with a lambda value.

112) Identifiers of visible automatic objects that are not captures, may still be used if they are not evaluated, for example in **`sizeof`** expressions (if they are not VM types) or as controlling expression of a generic primary expression.

113) Since each closure expression may have a unique type, it is generally not possible to assign it to an object with lambda value or to a function pointer that is declared outside of its defining scope or to use it, even indirectly, through a pointer to lambda value. Therefore the present constraint inhibits the use of an lvalue closure outside of the widest enclosing scope of its defining closure expression in which all its lvalue captures are visible.

capture clause, its parameter clause and its function body. Each explicit 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. The scope of visibility of implicit captures is 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 address of value captures is not necessarily unique.

10 A lambda expression for which at least one parameter declaration in the parameter list has no type specifier is a *type-generic lambda* with an imcomplete lambda type. It shall only occur in a void expression, as the postfix expression of a function call or, if the capture clause is empty, in a conversion to a pointer to function with fully specified parameter types, see 6.3.2.1. For a void expression, it has no side effects and shall be ignored.

11 For a function call, the type of an argument (after lvalue, array-to-pointer or function-to-pointer conversion) to an underspecified parameter shall be such that it can be used to complete the type of that parameter analogous to 6.7.10, only that the inferred type for an parameter of array or function type is adjusted analogously to function declarators (6.7.6.3) to a possibly qualified object pointer type (for an array) or to a function pointer type (for a function) to match type of the argument. For a conversion of any arguments, the parameter types shall be those of the function type.

12 If a value capture `id` is defined without an assignment expression, the assignment expression is assumed to be `id` itself, referring to the object of automatic storage duration of the surrounding scope that exists according to the constraints.[114)]

13 The implicit or explicit 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 `id` or one of its members is taken, either explicitly by applying a unary `&` operator or by an array to pointer conversion,[115)] and that address is used to modify the underlying object, the behavior is undefined. The evaluation of `E` takes place during the evaluation of the lambda expression; for an explicit capture when the value capture is met and for an implicit capture at the beginning of the evaluation of the function body.

14 The object of automatic storage duration `id` of the surrounding scope that corresponds to an lvalue 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. 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.[116)]

15 For each lambda expression, the return type *type* is inferred as indicated in the constraints. A lambda expression λ that is not type-generic has an unspecified lambda type `L` that is the same for every evaluation of λ. If λ appears in a context that is not a function call, 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.

Recommended practice

16 To avoid their accidental modification, it is recommended that declarations of lambda type objects are **`const`** qualified. Whenever possible, implementations are encouraged to diagnose any attempt

[114)] The evaluation in rules in the next paragraph then stipulates that it is evaluated at the point of evaluation of the lambda expression, and that within the body of the lambda an unmutable **`auto`** object of the same name, value and type is made accesssible.

[115)] 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.

[116)] That is, lvalue conversion of `id` results in the same lvalue with the same type and address as for the scope surrounding the lambda.

to modify a lambda type object.

17 **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 var;
int main(void) {
  [   ](void){ printf("%ld\n", var); }();                  // valid, prints 0
  [var](void){ printf("%ld\n", var); }();                  // invalid, var is static

  int var = 5;

  auto const λ = [var](void){ printf("%d\n", var); };      // freeze var
  [&var](void){ var = 7; printf("%d\n", var); }();         // valid, prints 7
  λ();                                                     // valid, prints 5
  [ var](void){ printf("%d\n", var); }();                  // valid, prints 7
  [   ](void){ printf("%d\n", var); }();                   // invalid
  [ var](void){ printf("%zu\n", sizeof var); }();          // valid, prints sizeof(int)
  [   ](void){ printf("%zu\n", sizeof var); }();           // valid, prints sizeof(int)
  [   ](void){ extern long var; printf("%ld\n", var; }(); // valid, prints 0

}
```

18 **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.

19 **EXAMPLE 3** Consider the following type-generic function literal that computes the maximum value of two parameters `X` and `Y`.

```
#define MAXIMUM(X, Y)                                   \
    [](auto a, auto b){                                 \
      return (a < 0)                                    \
             ? ((b <  0) ? ((a < b) ? b : a) : b)       \
             : ((b >= 0) ? ((a < b) ? b : a) : a);      \
    }(X , Y)

    auto R = MAXIMUM(-1, -1U);
    auto S = MAXIMUM(-1U, -1L);
```

After preprocessing, the definition of `R` becomes

```
auto R = [](auto a, auto b){
  return (a < 0)
     ? ((b <  0) ? ((a < b) ? b : a) : b)
     : ((b >= 0) ? ((a < b) ? b : a) : a);
  }(-1, -1U);
```

To determine type and value of `R`, first the type of the parameters in the function call are inferred to be **`signed int`** and **`unsigned int`**, respectively. With this information, the type of the **`return`** expression becomes the common arithmetic type of the two, which is **`unsigned int`**. Thus the return type of the lambda is that type. The resulting lambda value is the first operand to the function call operator `()`. So `R` has the type **`unsigned int`** and a value of **`UINT_MAX`**.

For `S`, a similar deduction shows that the value still is **`UINT_MAX`** but the type could be **`unsigned int`** (if **`int`** and **`long`** have the same width) or **`long`** (if **`long`** is wider than **`int`**).

As long as they are integers, regardless of the specific type of the arguments, the type of the expression is always such that the mathematical maximum of the values fits. So `MAXIMUM` implements a type-generic maximum macro that is suitable for any combination of integer types.

20 **EXAMPLE 4**

```
void matmult(size_t k, size_t l, size_t m,
             double const A[k][l], double const B[l][m], double const C[k][m]) {
  // dot product with stride of m for B
  // ensure constant propagation of l and m
  auto const λδ = [l,m](double const v[l], double const B[l][m], size_t m0) {
    double ret = 0.0;
    for (size_t i = 0; i < l; ++i) {
      ret += v[i]*B[i][m0];
    }
    return ret;
  };
  // vector matrix product
  // ensure constant propagation of l and m, and accessibility of λδ
  auto const λμ = [l, m, &λδ](double const v[l], double const B[l][m], double res[m]) {
    for (size_t m0 = 0; m0 < m; ++m0) {
      res[m0] = λδ(v, B, m0);
    }
  };
  for (size_t k0 = 0; k0 < k; ++k0) {
    double const (*Ap)[l] = A[k0];
    double (*Cp)[m] = C[k0];
    λμ(*Ap, B, *Cp);
  }
}
```

This function evaluates two closures; $\lambda\delta$ has a return type of **`double`**, $\lambda\mu$ 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

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

typedef
extern
static
_Thread_local
auto
register

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that **_Thread_local** may appear with **static** or **extern** , and that **auto** may appear with all others but with **typedef** .[134)]

3 In the declaration of an object with block scope, if the declaration specifiers include **_Thread_local**, they shall also include either **static** or **extern**. If **_Thread_local** appears in any declaration of an object, it shall be present in every declaration of that object.

4 **_Thread_local** shall not appear in the declaration specifiers of a function declaration. **auto** shall only appear in the declaration specifiers of a function declaration if it is the declaration part of a function definition or if the corresponding function has already been defined.

Semantics

5 The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only; it is discussed in 6.7.8. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

6 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.[135)]

7 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

8 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

9 If **auto** appears with another storage-class specifier, or if it appears in a declaration at file scope it is ignored for the purpose of determining a storage class or linkage. It then only indicates that the declared type may be inferred from an initializer (for objects see 6.7.10), or from the function body (for functions see 6.8.6.4).

Forward references: type definitions (6.7.8), type inference (6.7.10), function definitions (6.9.1).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

void
char
short
int

[134)] See "future language directions" (6.11.5).

[135)] The implementation can treat any **register** declaration simply as an **auto** declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary & operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

long
float
double
signed
unsigned
_Bool
_Complex
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name

Constraints

2 ~~At~~ Unless stated otherwise, at least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— **void**
— **char**
— **signed char**
— **unsigned char**
— **short**, **signed short**, **short int**, or **signed short int**
— **unsigned short**, or **unsigned short int**
— **int**, **signed**, or **signed int**
— **unsigned**, or **unsigned int**
— **long**, **signed long**, **long int**, or **signed long int**
— **unsigned long**, or **unsigned long int**
— **long long**, **signed long long**, **long long int**, or **signed long long int**
— **unsigned long long**, or **unsigned long long int**
— **float**
— **double**
— **long double**
— **_Bool**
— **float _Complex**
— **double _Complex**
— **long double _Complex**
— atomic type specifier
— struct or union specifier
— enum specifier
— typedef name

3 The type specifier **_Complex** shall not be used if the implementation does not support complex types (see 6.10.8.3).

Semantics

4 Specifiers for structures, unions, enumerations, and atomic types are discussed in 6.7.2.1 through 6.7.2.4. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5. Declarations for which the type specifiers are inferred from initializers are discussed in 6.7.10.

5 Each of the comma-separated multisets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **`int`** designates the same type as **`signed int`** or the same type as **`unsigned int`**.

6 A declaration that contains no type specifier is said to be *underspecified*. Identifiers that are such declared have incomplete type. Their type can be completed by type inference from an intialization (for objects) or from **`return`** statements in a function body (for return types of functions).

Forward references: atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.8)~~.~~, type inference (6.7.10).

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
struct-or-union *identifier*$_{\text{opt}}$ **`{`** *struct-declaration-list* **`}`**
struct-or-union *identifier*
struct-or-union:
`struct`
`union`
struct-declaration-list:
struct-declaration
struct-declaration-list *struct-declaration*
struct-declaration:
specifier-qualifier-list *struct-declarator-list*$_{\text{opt}}$ **`;`**
static_assert-declaration
specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{\text{opt}}$
type-qualifier *specifier-qualifier-list*$_{\text{opt}}$
alignment-specifier *specifier-qualifier-list*$_{\text{opt}}$
struct-declarator-list:
struct-declarator
struct-declarator-list **`,`** *struct-declarator*
struct-declarator:
declarator
declarator$_{\text{opt}}$ **`:`** *constant-expression*

Constraints

2 A struct-declaration that does not declare an anonymous structure or anonymous union shall contain a struct-declarator-list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted.[136)] If the value is zero, the declaration shall have no declarator.

[136)] While the number of bits in a **`_Bool`** object is at least **`CHAR_BIT`**, the width (number of sign and value bits) of a **`_Bool`** can be just 1 bit.

```
                * type-qualifier-list_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_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 (6.7.10).

6.7.6.1 Pointer declarators

Semantics

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

```
* type-qualifier-list_opt D
```

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

```
    }
```

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

6.7.6.3 Function declarators (including prototypes)

Constraints

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

2 The only storage-class ~~specifier~~ specifiers that shall occur in a parameter declaration ~~is~~ are **`auto`** and **`register`**.

3 An identifier list in a function declarator that is not part of a definition of that function shall be empty. A parameter declaration without type specifier shall not be formed, unless it includes the storage class specifier **`auto`** and unless it appears in the parameter list of a lambda expression.

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

Semantics

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

`D (` *parameter-type-list* `)`

or

`D (` *identifier-list*$_{\text{opt}}$ `)`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning the unqualified version of *T*".

6 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

7 ~~A~~ After the declared types of all parameters have been determined in order of declaration, any declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword **`static`** also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

8 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

9 If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[158)]

10 The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

11 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

12 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types.

13 The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

14 An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function

[158)] The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.

```
struct S {
      int i;
      struct T t;
};

struct T x = {.l = 43, .k = 42, };

void f(void)
{
      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).

6.7.10 Type inference

Constraints

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

2 For an underspecified declaration of a function that is also a definition, the return type shall be completed as of 6.9.1. For an underspecified declaration of a function that is not a definition a prior definition of the declared function shall be visible.

3 An underspecified declaration of an object that is also a definition and that is not the declaration of a parameter shall be of one of the forms

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

such that the declarator does not declare an array.

4 For an underspecified declaration such that the assignment expression does not have lambda type there shall be a type specifier *type* that can be inserted in the declaration immediately after the last storage class specifier that makes the adjusted declaration a valid declaration and such that the assignment expression, after possible lvalue, array-to-pointer or function-to-pointer conversion, has the non-atomic, unqualified type of the declared object.[167] if the assignment expression has lambda type, the lambda type shall be complete and the declarator shall only consist of storage class specifiers, qualifiers and the identifier that is to be declared. A function declaration that is not a definition shall have a type that is compatible with the type of the corresponding definition.

Description

5 Although there is no syntax derivation to form declarators of lambda type, values of lambda type can be used as assignment expression and the inferred type is that lambda type, possibly qualified. Otherwise, provided the constraints above are respected, in an underspecified declaration the type

[167] For most assignment expressions of integer or floating point type, there are several types *type* that would make such a declaration valid. The second part of the constraint ensures that among these a unique type is determined that does not need further conversion to be a valid initializer for the object.

of the declared identifiers is the type after the declaration has been adjusted by *type*. The type of each identifier that declares an object is incomplete until the end of the assignment expression that initializes it.

6 **NOTE** The scope of the identifier for which the type is inferred only starts after the end of the initializer (6.2.1), so the assignment expression cannot use the identifier to refer to the object or function that is declared, for example to take its address. Any use of the identifier in the initializer is invalid, even if an entity with the same name exists in an outer scope.

```
   {
     double a = 7;
     double b = 9;
     {
       double b = b * b;    // error, RHS uses uninitialized variable
       printf("%g\n", a);   // valid, uses "a" from outer scope, prints 7
       auto a   = a * a;    // error, "a" from outer scope is already shadowed
     }
     {
       auto b   = a * a;    // valid, uses "a" from outer scope
       auto a   = b;        // valid, shadows "a" from outer scope
       ...
       printf("%g\n", a);   // valid, uses "a" from inner scope, prints 49
     }
     ...
   }
```

7 **EXAMPLE 1** Consider the following definitions:

```
static auto a = 3.5;
auto * p = &a;
```

They are interpreted as if they had been written as:

```
static auto double a = 3.5;
auto double * p = &a;
```

which again is equivalent to

```
static double a = 3.5;
double * p = &a;
```

So effectively `a` is a **`double`** and `p` is a **`double`**`*`.

8 **EXAMPLE 2** In the following, `pA` is valid because the type of `A` after array-to-pointer conversion is a pointer type, and `qA` is valid because it does not declare an array but a pointer to an array.

```
double A[3] = { 0 };
auto const * pA = A;
auto const (*qA)[3] = &A;
```

9 **EXAMPLE 3** Type inference can be used to capture the type of a call to a type-generic function and can be used to ensure that the same type as the argument `x` is used.

```
#include <tgmath.h>
auto y = cos(x);
```

If instead the type of `y` is explicitly specified to a different type than `x`, a diagnosis of the mismatch is not enforced.

10 **EXAMPLE 4** A type-generic macro that generalizes the **`div`** functions (7.22.6.2) is defined and used as follows.

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 body that has an underspecified return type, 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 ~~terminates execution of the current function~~ is associated to the innermost function body in which appears. It evaluates the expression, if any, terminates the execution of that function body and returns control to its caller~~. A function~~; 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 a function that has an underspecified return type, 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.

6.9 External definitions

Syntax

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

external-declaration:
function-definition
declaration

Constraints

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

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

Semantics

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

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** or **_Alignof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[181)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
declaration-specifiers declarator declaration-list$_{\text{opt}}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[182)]

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

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

5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier. No declaration list shall follow.

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

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

7 An underspecified function definition shall contain an **`auto`** storage class specifier. The return type for such a function is determined as described for the **`return`** statement (6.8.6.4) and shall be visible prior to the function definition.

Semantics

8 If **`auto`** appears as a storage-class specifier it is ignored for the purpose of determining a storage class or linkage of the function. It then only indicates that the return type of the function may be inferred from **`return`** statements or the lack thereof, see 6.8.6.4.

9 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator (possibly adjusted by an inferred type specifier) also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,[183] the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.6.3 for a parameter type list; the resulting type shall be a complete object type.

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

11 Each parameter has automatic storage duration; its identifier is an lvalue.[184] The layout of the storage for parameters is unspecified.

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

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

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

15 Provided the constraints above are respected, the return type of an underspecified function definition is adjusted as if the corresponding type specifier had been inserted in the definition. The type of such a function is incomplete within the function body until the lexically first **`return`** statement that it contains, if any, or until the end of the function body, otherwise.[185]

16 **NOTE** In a function definition, the type of the function and its prototype cannot be inherited from a typedef:

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

[182] ~~The intent is that the type category in a function definition cannot be inherited from a typedef.~~

[183] See "future language directions" (6.11.7).

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

[185] This means that such a function cannot be used for direct recursion before or within the first return statement.

17 **EXAMPLE 1** In the following:

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

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

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

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

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

Here **int** `a, b;` is the declaration list for the parameters. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not.

18 **EXAMPLE 2** To pass one function to another, one might say

```
int f(void);
/* ... */
g(f);
```

Then the definition of `g` might read

```
void g(int (*funcp)(void))
{
      /* ... */
      (*funcp)(); /* or funcp(); ...*/
}
```

or, equivalently,

```
void g(int func(void))
{
      /* ... */
      func(); /* or (*func)(); ...*/
}
```

19 **EXAMPLE 3** Consider the following function that computes the maximum value of two parameters that have integer types T and S.

```
   inline auto max(T, S); // invalid: no definition visible
   ...
   inline auto max(T a, S b){
     return (a < 0)
            ? ((b <  0) ? ((a < b) ? b : a) : b)
            : ((b >= 0) ? ((a < b) ? b : a) : a);
   }
   ...
   // valid: definition visible
   extern auto max(T, S); // forces definition to be external
   auto max(T, S);        // same
   auto max();            // same
```

The **return** expression performs default arithmetic conversion to determine a type that can hold the maximum value and is at least as wide as **int**. The function definition is adjusted to that return type. This property holds regardless if types T and S have the same or different signedness.

The first forward declaration of the function is invalid, because an **auto** type function declaration that is not a definition is only valid if the definition of the function is visible. In contrast to that, the **extern** declaration and the two following equivalent ones are valid because they follow the definition and thus the inferred return type is known. Thereby in is ensured that the translation unit provides an external definition of the function.

20 **EXAMPLE 4** The following function computes the sum over an array of integers of type T and returns the value as the promoted type of T.

```
inline
auto sum(size_t n, T A[n]){
  switch(n) {
    case 0:
      return +((T)0);                          // return the promoted type
    case 1:
      return +A[0];                            // return the promoted type
    default:
      return sum(n/2, A) + sum(n - n/2, &A[n/2]);  // valid recursion
  }
}
```

If instead sum would have bee defined with a prototype as follows

```
T sum(size_t n, T A[n]);
```

for a narrow type T such as **unsigned char**, the return type and result would be different from the previous. In particular, the result of the addition would have been converted back from the promoted type to T before each **return**, possibly leading to a surprising overall results. Also, specifying the promoted type of a narrow type T explicitly can be tedious because that type depends on properties of the execution platform.

6.9.2 External object definitions

Semantics

1 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

2 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

3 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

4 **EXAMPLE 1**

```
int i1 = 1;         // definition, external linkage
static int i2 = 2;  // definition, internal linkage
extern int i3 = 3;  // definition, external linkage
int i4;             // tentative definition, external linkage
static int i5;      // tentative definition, internal linkage

int i1;             // valid tentative definition, refers to previous
int i2;             // 6.2.2 renders undefined, linkage disagreement
int i3;             // valid tentative definition, refers to previous
int i4;             // valid tentative definition, refers to previous
int i5;             // 6.2.2 renders undefined, linkage disagreement

extern int i1;      // refers to previous, whose linkage is external
```