```
Comment response for C23
WG14 3076

Title:                Auto as a placeholder type specifier
Author, affiliation: Alex Gilding, Perforce
Date:                 2023-01-05
Proposal category:    National Body Comment Response
Target audience:      Committee wording review
```

Abstract

This is a response to US National Body Comments 121, 122 and 123 against the CD ballot draft of the C23 Standard.

The comments request two main changes to the way the `auto` specifier works: it should become a type specifier rather than remaining a storage class specifier, because it does not describe a storage class any more; and it should permit partial deduction of the type of the initializing value by allowing derived types to derive from the placeholder it describes.

Both of these changes should improve compatibility with C++, the first mostly for implementers and readers of the text; the second to better conform to common user expectations about how the feature should be available.

Auto as a placeholder type specifier

Reply-to: Alex Gilding (agilding@perforce.com)
Document No: N3076
Revises Document No: N/A
Date: 2023-01-05

Proposal History

This is the first version of this document. However, the changes are applied to the feature that was introduced by:

N2953 Type inference for object definitions

- Adopted into C23 at July 2022 meeting

N3067 CD 9899 ballot comments

- Comments 121, 122 and 123 request changes to the specification and mechanism of the `auto` specifier from the version which was adopted.

Introduction

The adopted specification for `auto` aimed to meet two goals:

- not break any existing code that provided explicit type specifiers (code relying on implicit type specifiers is no longer allowed);
- inherit the *exact* semantics of the feature as already implemented by GCC and Clang via the `__auto_type` specifier found in the GNU C dialect.

`auto` can therefore be combined with other type specifiers, but as it is currently provided it is not completely compatible with the C++ feature. This is because the C++ feature specified in C++20/[dcl.type.auto.deduct] relies on wording from the definition of templates (C++20/[temp.deduct]), which are completely missing from C; an equivalent feature needs to be reconstructed instead of using a common specification.

However, the missing specifications also leave the C feature far more limited in scope, so reconstructing only the parts relevant to object definitions is not as difficult as initially feared.

Additionally, changing the definition of `auto` from a storage class to a type specifier is separately possible and comparatively easy. This is a prerequisite task for partial deduction which can be integrated separately.

Proposed Resolutions

Resolution 121: `auto` as a type specifier

US NB Comment 121 requests that `auto` be made into a type specifier rather than being left as a storage class. Treating it as a storage class seems to make unnecessary complexity for implementers and introduces incompatibility with C++. It leaves in the ambiguity of declarations with no type at all needing one to be implicitly added by the compiler (implicit `int` becoming implicitly-deduced, but still *syntactically* implicit), makes the specification more complicated, and serves no purpose because the specifier does not describe a storage class (it can be used with static and thread storage durations).

Changing the specification to refer to a type specifier simplifies all of this and is also simpler for implementers, who can choose to interpret the keyword as one of two internally-different specifiers depending on the language mode, rather than a storage class specifier with varying effects (i.e. one grammar can contain two *different* syntactic `auto` side-by-side). This is interesting to developers also shipping combined C++ compilers who would like to be able to reuse as much of their internals as possible.

Proposed Resolution

Introduce the term “placeholder type specifier” as found in C++. A *placeholder type specifier* is simply a syntactic type specifier that has some additional constraints on where it may appear (i.e. not in a *type-name*). This is an italicized term of art.

Change all references to `auto` as a storage class specifier to refer to it as a (placeholder) type specifier. Change all references to declarations that omit a type (of which there are now none), to declarations that contain a *placeholder type specifier*. Allow the *placeholder type specifier* to form part of any type specifier multiset, where it is ignored if it is not the only component of the type specifier (this subsumes usages like `auto int x = 10;` by making the type `auto int`, in which `auto` is ignored and therefore just `int`).

Resolution 122: support pointer and array declarators

US NB comment 122 requests that C explicitly require support for pointer and array declarators.

Using explicit pointer declarators is common practice in C++, and users would be surprised not to find it in C as well. C++ also supports deduction of function return types and array element types in pointer-to-function and pointer-to-array declarators (but *not* function parameter types).

Note that C++ deduces the type of a *braced-initializer* to be a `std::initializer_list`, not an array. The syntax `auto a[] = { … };` is therefore not valid C++. *However*, lacking this library feature, C users will expect arrays to substitute for it; and in any case there is no additional complexity in allowing this form so long as we do not bother with complicated element coercion rules, and simply demand the elements be explicitly compatible. (It would actually introduce more wording to disallow this.)

This form was explicitly requested by the US NB so it is included here. The comments from Intel are not concerned about a C++ compatibility issue here because definitions are mostly at block scope.

Proposed Resolution

Building on resolution 121, allow *derived types* to be constructed from the *placeholder type*, so long as the sequence of derived constructions matches the same outermost sequence of derived type constructions on the initializer expression.

This trivially and elegantly allows the following forms:

```
auto * p1 = …
auto p2[] = { … }
auto (*p3)[3] = …;
auto (*p4)(void) = …;
```

and disallows

```
atomic (auto) a = …;
int (*f)(auto) = …;
```

Note that while the `atomic` derived type is not allowed because it uses `auto` as a *type-name*, atomically-*qualifying* the declared object is perfectly OK. This is a rare difference in the behaviour of the `atomic` specifier and qualifier.

Resolution 123: no undefined behaviour on non-identifier declarators

Defining the behaviour as undefined when the declarator is not a (possibly-parenthesized) plain identifier allows implementations to do stupid things. This should be well-specified as either an error or well-supported.

Proposed Resolution

No additional action is needed because the changes to support resolution 122 remove this restriction and allow the full range of declarator syntaxes, with appropriate constraints when they do not make sense. This is therefore resolved by addressing the first two points.

Impact

As integrated into C23, the `auto` specifier was an **exact** standardization of the GNU C `__auto_type` specifier, renamed to look like the C++ keyword. This has the advantage of being completely backed by long-established practice.

This proposal significantly extends the feature to more closely match it as it appears in C++. This is *not* invention, but it is also not the exact GNU C feature we initially promised to standardize.

Feedback from US NB implementers seemed to indicate that being more generous with the syntax to allow additional C++-ish forms (i.e. `auto * px =`) is preferable to sticking to exact C dialect practice, because the C++ practice is more useful to them.

Proposed Wording

Changes are proposed against the wording in C23 draft n3054. Bolded text within an existing paragraph is new text.

Modify 6.7 “Declarations”:

The first sentence of paragraph 12 does not imply a declaration can have no type specifier:

> A declaration such that the declaration specifiers contain **a placeholder** type specifier **(6.7.9)** or that is declared with `constexpr` is said to be underspecified.

Add a forward reference to 6.7.2:

> Forward references: declarators (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.10), type names (6.7.7), **type specifiers (6.7.2),** type qualifiers (6.7.3).

Modify 6.7.1 “Storage-class specifiers” to remove all mention of `auto`:

- Remove `auto` from the list in paragraph 1.
- Remove the second bullet-point, from paragraph 2, and delete footnote 138. Remove `auto` from what is currently the third bullet-point.
- Remove the entire second sentence from paragraph 4.
- Remove `auto,` from the first bullet-point in paragraph 6.
- Remove the final part of the sentence in paragraph 7 that forward-references 6.7.9.
- Remove the entire paragraph 11.
- In footnote 142, change the typewriter-face word “`auto`” to the normal-face word “automatic” (this is not about the specifier anyway).

Modify 6.7.2 “Type specifiers”:

Add a new entry to the list in paragraph 1:

> ...
> *atomic-type-specifier*
> *struct-or-union-specifier*
> *enum-specifier*
> *typedef-name*
> *typeof-specifier*
> **`auto`**

Delete the first part of the first sentence of paragraph 2, up to the comma (“Except where the type is inferred (6.7.9),”).

Add a final bullet point to the multiset list in paragraph 2:

...
– enum specifier
– typedef name
– typeof specifier
– `auto`

Add a new paragraph before paragraph 3:

> The specifier `auto` may also appear, at most once, as an element of any other type specifier in the above list of multisets.

Add a reference to type inference in paragraph 5:

> Specifiers for structures, unions, enumerations, atomic types, and typeof specifiers are discussed in 6.7.2.1 through 6.7.2.5. Declarations of typedef names are discussed in 6.7.8. **Type inference from the placeholder type specifier is discussed in 6.7.9.** The characteristics of the other types are discussed in 6.2.5.

Delete paragraph 6.

Add a sentence to the end of paragraph 7:

> 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`. **If the specifier `auto` appears as an element of any other *specifier-qualifier-list* in the list above, it is ignored, designating the same type as would be selected if `auto` was not present.**

Add a new paragraph after paragraph 7:

> When the specifier `auto` appears as the only type specifier in a *specifier-qualifier-list*, it is a *placeholder type specifier* [footnote)]. The designated type is inferred from the declaration initializer as discussed in 6.7.9.
>
> footnote) This only appears in the declaration specifiers of a declaration and not in any other context where the name of a type is used (6.7.7).

(This removes the need to point out what optional elements of a declaration appertain to, because the designated type is now concretely associated with the `auto` specifier in the type position.)

Add forward references to 6.7.7 and 6.7.9 at the end of the section:

> 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 names (6.7.7),** type definitions (6.7.8)**, type inference (6.7.9)**.

Modify example 4 in 6.7.6.2 “Array declarators” to remove keyword-highlighting from the word “auto” in comments, which has nothing to do with the specifier in this context.

Modify 6.7.7 “Type names”:

Add a “Constraints” section before paragraph 2:

> **Constraints**
> An `auto` specifier shall not appear as part of a *type name*.

(This is enough to rule out the use of `auto` in casts, `sizeof`, etc. contexts where it makes no sense. It also forbids compound literals, which we may find a use case for allowing later.)

Modify 6.7.9 “Type inference”:

> **6.7.9 Type inference**
>
> **Constraints**
> A declaration for which the type is inferred shall contain the ***placeholder type specifier* (6.7.2)**.
>
> Each *init-declarator* in such a declaration shall have the form footnote):
>
> *declarator* = *initializer*
>
> footnote) In other words, the declaration is always a definition (of an object, not a function), with an explicit initializer.
>
> The placeholder type specifier shall not appear within the parameters of a function type derivation.
>
> If the initializer is a *braced-initializer* (6.7.10), there shall be an explicit *initializer-list*, and all elements of the *initializer-list* shall have the same type footnote).
>
> footnote) Compatible types need not be the same.
>
> If the declaration declares more than one identifier, each declared identifier shall infer the same *inferred type* for the placeholder type specifier.
>
> **Semantics**
> If the initializer is an *assignment-expression*, the type of the declared identifier is the type of the initializer after lvalue, array to pointer or function to pointer conversion, additionally qualified by qualifiers and amended by attributes as they appear in the declaration specifiers, if any 177).
>
> If the initializer is a *braced-initializer*, the type of the declared identifier is an array type, with an element type that is the type of the initializers of the initializer list footnote). If a size is not provided in an array declarator, it is deduced according to the rules for an array of unknown size (6.7.10).
>
> footnote) No coercion or deduction of a common type is performed, as the type of all initializers in the list is the same, per the constraints.
>
> If the type specified for the declarator is a derived type constructed from the placeholder type, the type of the initializer shall be a type that can be constructed by an identical sequence of type derivations from a complete object type.

If the type specified for the declarator is such a derived type, the *inferred type* designated by the placeholder type specifier is the corresponding complete object type from which the type of the initializer is constructed. Otherwise, the *inferred type* is the unqualified type of the declared identifier.

...then continue 6.7.9 from “NOTE 1” with the rest of the current body.

Modify the last sentence of (what is currently) paragraph 4:

Note that the final type here is a pointer type**, even though the declarator is not `*p`**.

Add a forward reference:

Forward references: initialization (6.7.10).

Add a two new examples after paragraph 9:

EXAMPLE 7 When a variable is declared with a derived placeholder type, the inferred type must match the derived-from type of the initializer:

```
     auto x = 10;  // no derivation from auto

     auto * px1 = &x;  // valid, initializer is a pointer
     auto const * px2 = &x;  // valid, const is not a
derivation
     auto * px3 = x;  // invalid, x does not have pointer
type

     auto ** ppx1 = &px1;  // valid, initializer is a
pointer to a pointer
     auto const ** ppx2 = &px2;  // valid, initializer is a
pointer to a pointer to const
     auto const ** ppx3 = &px1;  // invalid, cannot convert
to a pointer to pointer to const

     int y[10] = {};

     auto * py1 = &y; // valid, py1 has type int(*)[10]
     auto * py2 = y;  // valid, py2 has type int *
     auto (*py3)[10] = &y;  // valid, pointer to auto[10]
has same derivations as pointer to int[10]

     int f (int, float);
     auto * pf1 = f;  // valid
     auto (*pf2)(int, float) = f;  // valid, declared type
derives from placeholder return type
     int (*pf3)(auto, auto) = f;  // invalid, cannot derive
from placeholder parameter type
```

EXAMPLE 8 When a variable is initialized with a braced-initializer, it has an array type inferred from the initializer elements.

```
     auto a1 = { 1, 2, 3 };  // type is int[3]
     auto a2[] = { 1, 2 };   // type is int[2]
```

```
     auto a3 = { 1 };        // type is int[1]
     auto a4[] = {};  // invalid, must have an initializer

     auto a5[] = { [5] = 0 }; // type is int[6]
     auto a6[3] = { [5] = 0 }; // invalid, [5] is not
within the array
     auto a7 = { 1u, 2, 3.0 }; // invalid, initializer
types are not the same
```

EXAMPLE 9 When a declaration contains multiple identifiers to declare, the *inferred type* for each declarator needs to be the same, but the final object type can be different.

```
     auto x = 10, y = 20;  // valid, both the same
     auto w = 10, *z = &x;  // valid, both infer int
                            // even though object types are
different

     auto a = 5, b = 6.0;  // invalid, infer different
types
     auto *c = &x, d = z;  // invalid, inferring different
types (int and int*)
                           // even though object types are
the same
```

Modify 6.8.5 “Iteration statements”, paragraph 3:

> The declaration part of a for statement shall only declare identifiers for objects **with automatic storage duration**.

Modify 6.9 “External definitions”, second sentence of paragraph 2:

> The `auto` **specifier** shall only appear in the declaration specifiers in an external declaration if the **declaration is a definition and the** type is inferred **from the initializer**.

Remove undefined behaviour (77).

Remove J.5.11 “Type inference”.

(as currently specified in the text provided above, all behaviours are well-defined; we might choose to make some elements of this specification undefined again to allow for extensions like `int (*f) (auto, auto) = ...`)

References

N2953 2022/04/10 Gustedt, Type inference for object definitions

N3054 2022/09/03 Meneide, Programming languages — C

N3067 2022/12/21 Keaton, CD 9899 ballot comments

N4868 C++20 working draft

Acknowledgments

Thanks to Jens Gustedt and Aaron Ballman for some quick-turnaround on very insightful review comments and additional wording suggestions!