Proposal for C23
WG14 3020

Title: Qualifier-preserving standard library functions, v4
Author, affiliation: Alex Gilding, Perforce
Date: 2022-06-13
Proposal category: Feature enhancement
Target audience: Library developers, library users

Abstract

Proposal to improve type safety of existing C Standard library functions by preventing the ability to silently remove `const`-qualification without a cast. This introduces a suggested new notation for qualifier-generic functions, and aims to slightly reduce the difference between the C and C++ Standard libraries.

Qualifier-preserving standard library functions, v4

Reply-to: Alex Gilding (agilding@perforce.com)
Document No: N3020
Revises Document No: N3012
Date: 2022-06-28

Summary of Changes

N3020

- Fix explicitness of null pointer constant, considering C++, ignore `nullptr`
- Remove optional change 2, not how we want to proceed with polymorphism

N3012

- Fix example implementation, consider (type of) null pointer constant
- Remove mention of `_Atomic`, do mention `restrict`
- Consider ordering of `bsearch` argument conversions
- Explicitly allow suppression to access the old external declarations
- Add an option to make macro suppression for these functions obsolescent
- Add an option to retain the object type for `bsearch`

N2973

- Simplify notation to avoid possible ambiguity of nil-expansion
- Ensure external pointer type is specified and underlying functions are addressable
- Improve example implementation to preserve object type
- Clarify which qualifiers are supported/abstracted explicitly

N2603

- original proposal

Introduction

The C Standard Library contains twelve "qualifier-losing" search functions:

```
bsearch
bsearch_s
memchr
strchr
strpbrk
strrchr
strstr
```

```
wcschr
wcspbrk
wcsrchr
wcsstr
wmemchr
```

These functions accept a `const`-qualified pointer to a buffer to be searched, but return the pointer to the found element without `const`-qualification.

This proposal suggests that the library should specify that these functions return a found result element with the same qualification as the input buffer, as provided by the user. This avoids the risk of inadvertently "casting away" constness on an input buffer when this is not desired.

This proposal is extracted from n2522, "a common C/C++ core specification", by Jens Gustedt.

Rationale

The existing APIs were designed and added to the library before C11 introduced the ability to define generic and overloaded functions. Therefore the existing signatures are by-design, so that they can be used to search both `const` and mutable buffers - since `const`-qualification can be added implicitly, the input parameter is able to accept both kinds of buffer. If the functions are used correctly, the missing qualification on the return will be restored in the same way on assignment to an appropriately-qualified result pointer. This makes the search functions provide a primitive form of genericity.

Unfortunately this relies on the user to manually check that the variable for the returned pointer to the found element is of the correct type. If the user searches a `const`-qualified buffer but assigns the result to a pointer to a mutable object, the API has no way to directly communicate that this is a type error. An implementation may be able to warn anyway but this would require a hardcoded knowledge of the library functions and their contracts beyond what is communicated in-language.

For example:

```
wchar_t const buf[] = L"Hwæt! wē Gār-Dena in gēar-dagum þēod-cyningaþrym
gefrūnon";
wchar_t const * thr = wcschr (buf, L'þ'); // const implicitly restored
wchar_t * ans = wcschr (buf, L'æ'); // currently assignment not wrong;
const lost
// ...
*thr = L't'; // constraint violation
*ans = L'a'; // currently no error, runtime undefined behaviour
```

The `const`-loss is purely an API-accommodation for the fact that in C99 and earlier revisions, it was not obvious how to express qualifier-generic functions in-language. There is no valid use case for this lossiness (intentionally passing a `const`-qualified buffer and receiving the element without restoring the qualification), because casting away `const`-qualification explicitly is already a feature of the language. If the user really needs a non-`const`-qualified pointer into a `const`-qualified array, they can still communicate that this is an intentional decision, by casting the `const`-qualification from the pointer to the buffer before passing it as an input argument to the search API.

```
wchar_t * ans = wcschr ((wchart *)buf, L'æ'); // explicitly strip
constness
```

Unlike the `strerror` function and others discussed in n2526, there is no scope for a library implementation to provide additional mutability features, because the returned pointer is into a buffer controlled and provided completely by the user, not to some piece of library-internal state which may have other uses, or access points elsewhere in an extended API.

This vulnerability has already been fixed in C++, which is able to resolve an overloaded version of each function with the appropriate returned pointer type, based on the `const`-qualification of the buffer passed as the argument. This therefore reduces the footprint of differences between the two langauges' Standard libraries.

Proposal

We propose that each of these 12 functions be defined as a generic function, parameterized so that the qualification of the buffer pointed to by the first argument is the same as the qualification of the pointed to the found element.

This can be expressed with three choices of notation. Their relative merits, or lack thereof, may provide input to a related discussion about standardizing C's notation for generic library functions.

In all three cases the notation is proposed for the API description in the Standard only, and does **not** propose any new syntax for use in C code itself. (This was a point of confusion in the first round of discussion: the notation is not proposed C syntax.)

The concrete type of all functions is fully specified and therefore this proposal leaves the greater definition of *generic function* unaddressed, because it is not needed. In the case of every listed function a concretely-typed external declaration is fully specified, and the new signature descriptions only impose any constraints upon the functions' immediate uses. Unlike in the previous version of the proposal, there is no ambiguity about name redeclaration or the type of pointers to the underlying functions; it is not strictly necessary that a macro layer is implemented by the library at all, so expansion-suppression is not considered a problem or source of ambiguity.

Parameterizing only the qualifier:

```
char Q * strchr (char Q * s, int c);
void Q * memchr (void Q * s, int c, size_t n);
```

(following existing convention, *Q* would be italicized or otherwise marked)

The advantage of this notation is that it limits the scope of the constraint to the qualification and leaves the object type either explicitly typed, or `void`. This retains the explicit typing of those functions that examine characters, and does not place additional restrictions on how objects found in opaque buffers should be interpreted.

WG14 rejected the use of this notation.

Parameterizing the entire pointed-to type, including qualifier and object components:

```
C * strrchr (C * s, int c);
W * wcsrchr (W * s, wchar_t c);
C * strpbrk (C * s1, char const * s2);
```

The advantage of this convention is that the entire pointed-to type is parameterized, which may be visually clearer by communicating that the qualification is a preserved part of the complete derived type. It also does not imply that there is necessarily any qualification at all (i.e. Q can be empty / missing). However, by removing the explicit mention of the qualifier it also removes the focus on the fact that it is preserved.

This highlights an advantage: other pseudo-generic parameters (such as the `int` or `wint_t` second parameters) can potentially be marked as sharing the parameterized type. This indicates that the search key and the array of elements to examine are expected to be the same type of object. For `bsearch`, this might look like:

```
E * bsearch (E const * key, E * ptr, size_t count, size_t size
           , int (*comp) (void const *, void const *));
```

However, this highlights several problems: the notation now makes it visually ambiguous whether the parameter is supposed to include the qualification (the redundant qualification in `E const * ptr` is consistent with `typedef` rules, but not necessarily clear), and the fact that the `comp` parameter actually does need to be strongly-typed to accept void pointers means it cannot be written using the parameters. In addition, in the case of the `strchr` group, the key parameter does not have the same object representation, using `int` as a generic character key instead.

For the string search functions, this does show the possibility for further modification to the API, to unify the wide and narrow character searches into a single non-prefixed API. This would be a more intrusive change than proposed here. (see Future Directions, Generic functions)

For `bsearch`, this shows that simple parameterization in the signature is not sufficient to combine with `void` pointers used by higher-order function types. A more powerful mechanism for describing strongly-typed `void` pointers in generic higher-order functions is required. (see Future Directions, `void`-*which-binds*)

Overload lists:

```
char const * strstr (char const * s1, char const * s2);
char * strstr (char * s1, char const * s2);

wchar_t const * wmemchr (wchar_t const * s, wchar_t c, size_t n);
wchar_t * wmemchr (wchar_t * s, wchar_t c, size_t n);
```

This is consistent with C++, which actually does provide separate overloads.

However, this is misleading: this is not how overloading works or is implemented in C, and has no connection at all to the declarations that would exist in the library.

In C++, each overload is also a completely separate function - this declaration communicates that fact. In C, there is only ever one callable with a given name. If it exists as a non-macro entity, it has a singular type and it has a single exported external symbol. The C++ syntax is therefore not appropriate as it communicates a completely different set of assumptions about how the library makes names visible.

Committee feedback

Originally we proposed using the first option. In the first round of discussion the Committee found this confusing because the qualifier variable might expand to nothing, which was not visually clear from the notation. In addition there is no precedent anywhere in the document (or any existing practice at all, for C23) for abstracting qualifiers, but not whole types. This is therefore a concept with a distinct teach-ability disadvantage because it implicitly demands users to dive into the deep end of type-operators and kindedness, in order to properly understand what *should* really be a very simple signature.

Therefore the Committee settled on a preference for the second option, abstracting only whole types in generic function notation. This reduces the barriers to establishing a type for generic functions and reduces the complexity in the question of what a generic function "is" (luckily, outside of the scope of this proposal as generic functions already exist in C11).

The rest of this document will use the second syntax option, parameterizing the entire pointed-to type. We adopt a verbose naming scheme to make it clearer that the substituted types must match only a restricted class (e.g. *QChar* implying a pointer to a `char`, rather than admitting any type), and we do not extend the genericity to the `key` parameters in this proposal, instead leaving that to future work.

Alternatives

An alternative would be to deprecate all twelve functions and to add new APIs with the qualifier-preserving property. This has three major drawbacks:

- the new APIs would likely only be used by conscientious users who are less likely to make the underlying error in the first place.
- new APIs do not fix existing erroneous uses of the unsafe APIs; any errors will still be present and not elicit warnings.
- because C++ has already closed this loophole using its own in-language tools for these APIs, introducing new function names would widen rather than narrow the divergence between the two languages.

A potential heavier-weight alternative relying on future language directions would be to convert all APIs to use typed void pointers, i.e. the `void`-*which-binds*. This would also allow the compiler to enforce that the object type of the element is preserved and not just its qualification. This has the significant disadvantage of not yet existing in-language.

However, this may demonstrate an initial use case for such a feature.

Impact

ABI

None. The existing functions already work correctly when used with the appropriate operands. All this proposal does is alter the interpretation of the signature types to enable stricter checking of the invocations. The Standard guarantees that the object representation of a pointer to `T` and a pointer to

`T const` are compatible, so the ABI *must not* change to accommodate the proposed change (limited to qualifiers). A single underlying function body remains sufficient to implement the search feature itself, without changes.

C++ already provides two overloads which correctly propagate qualification. This means that the ABI problem would already be addressed in external linkage if two names are needed, and also that any existing code which compiles as both C and C++ has already been checked for qualifier-correctness by C++'s stricter library signatures. Therefore, user code in this category will be unaffected by the change as it must already be qualifier-correct.

Compilation

Some existing user code should be expected to raise an error when it compiled cleanly before. However, it seems that all such new warnings would indicate a legitimate design error - unlike the `strerror` case, there is no valid use case for implicitly removing `const`-qualification from a buffer that the user has ownership of. If they meant to do so, they can still strip the qualification explicitly themselves, and communicate that intent. Therefore, this should be considered an API safety improvement and the new errors would be desirable.

Implementation

A high-quality implementation should provide a macro wrapper around the calls to the implementation function that re-applies correct qualification to the result type. The only impact of this would be to introduce a compile-time implicit pointer type conversion between two compatible pointer types, meaning that there is guaranteed to be no run-time cost.

An example of this implementation wrapper might be:

```
// string.h
// ...

#define IS_POINTER_CONST(P) _Generic(1 ? (P) : (void *)(P)  \
    , void const *: 1                                       \
    , default     : 0)

#define STATIC_IF(P, T, E) _Generic (&(char [!!(P) + 1]) {0}  \
    , char (*) [2] : T                                        \
    , char (*) [1] : E)

#define _STRING_SEARCH_QP(T, F, S, ...)  \
  STATIC_IF (IS_POINTER_CONST ((S))      \
    , (T const *) (F) ((S), __VA_ARGS__) \
    , (T       *) (F) ((S), __VA_ARGS__))

#define memchr(S, C, N) _STRING_SEARCH_QP(void, memchr,  (S),  (C), (N))
#define strchr(S, C)    _STRING_SEARCH_QP(char, strchr,  (S),  (C))
#define strpbrk(S1, S2) _STRING_SEARCH_QP(char, strpbrk, (S1), (S2))
#define strrchr(S, C)   _STRING_SEARCH_QP(char, strrchr, (S),  (C))
#define strstr(S1, S2)  _STRING_SEARCH_QP(char, strstr,  (S1), (S2))
```

The null pointer constant is silently handled as a pointer to non-`const`, which is consistent with its definition. Since calling these functions with `NULL` is undefined and should be guarded, the usefulness of either behaviour is subjective; leaving the result type a pointer to mutable is slightly simpler to define and ever so slightly reduces the impact of the change (if anything was already depending on the result type in an unevaluated context, it will be unchanged).

In C++, attempting to use a null pointer constant (of any syntax) as the array operand results in an ambiguous overload call (compile error) and is therefore prohibited by that type system. There is no C++ behaviour to be compatible with, and there is also therefore no particular reason to treat `nullptr` differently from other null pointer constants in the event that feature is adopted.

(The previous version of this proposal suggested an implementation that would be back-portable to C90 and C99, relying on the qualifier-combining effect of the ternary operator; this is less suitable for C23 as it would risk exposing a `void *` return type for some arguments. In any case we expect implementations would mostly use `__builtin` magic here rather than a portable query.)

A check for specific inappropriate qualifiers is not required, as if the argument is more heavily qualified than the signature of the underlying search function allows, it will fail to type check after expansion.

A simpler implementation could continue to provide the APIs as they are and merely document that the constraint now applies. This might introduce some inconsistencies if the result of a search function is used with `_Generic`, but this is unlikely (the type is already one the user would not expect to match).

Future directions

This proposal introduces a number of ideas for further library development:

- in n2522, the wide and narrow versions of the string searches are all unified under the narrow-string names, and made generic in the character type. This would be a more intrusive change.
  - this would introduce the option for additional character types to be searchable by the same set of function names.
  - this significantly simplifies user-side type-generic string processing. However, this does not have precedent from C++, and would imply an ABI change. It is also unclear which headers would provide the APIs, and how completely (for instance, should the wide character searches be available from `string.h`, or should those "overloads" only become visible when both `string.h` and `wchar.h` are included? Should `wchar.h` still provide the explicit wide APIs?)
- the Embedded C TR 18037 introduces additional qualifiers to represent named address spaces. Pointers to objects in these address spaces can round-trip through `void`, but may have a different representation and can therefore not be examined by a generic pointer readthrough.
  - the `_Atomic` qualifier is already permitted to indicate different object layout requirements, and is similarly inappropriate to use with these APIs. This is therefore consistent with the idea that the overloads may restrict the set of qualifiers abstracted; currently only to `const`.
  - strongly-typed `void` pointers would be able to accommodate this when the read is only achieved through a provided access callback, so this restriction could be lifted from `bsearch` and `bsearch_s` in the presence of `void`-*which-binds*.

- Embedded C specifies that an implementation should provide some mechanism to declare the new qualifiers. This could exist separately as a way to tag opaque types with user-side meaningful data, without indicating an incompatible object representation (similar to `const`).
    - if such "tag qualifiers" were added to the language, it would be useful to allow them to qualify the operands to these search functions and to be preserved as well. They should be distinguished from qualifiers that may affect layout (address space, `_Atomic`) or the nature of access (`_Atomic` and `volatile`), which are not appropriate to abstract.
- ideally, the object type should also be preserved, rather than allowing the result pointer to a found element to be implicitly converted to a correctly-qualified pointer to an unrelated object layout.

  One proposal which would allow this to be communicated is [`void`-*which-binds*], introducing the concept of "strongly-typed `void`" to function declarations via attributes:

  ```
  #define Void(A) void [[bind_type (A)]]

  Void(T) * memchr (Void(T) * s, int c, size_t n);
  ```

  This proposal would require that all implicit conversions to or from a `void` pointer component of a signature that are annotated with the same parameter name, bind to or from the same object type. Therefore,

  ```
  char const buf[] = "o menel aglar elenath";
  char const * mp = memchr (buf, 'm', 21); // OK
  int const * ip = memchr (buf, 'm', 21);  // not OK if [[type(A)]] is
                                           // enforced
  ```

  This would allow typed parameterization of `bsearch's` key/buffer/result type without losing the connection to the `comp` function:

  ```
  Void(T) * bsearch (Void(T) const * key, Void(T) * ptr
                   , size_t count, size_t size
                   , int (*comp) (Void(T) const *, Void(T) const *));
  ```

  The ABI is not changed at all because the parameters are still `void` pointers, but their connection to a single element type is communicated.
- a unified notational convention for generic function signatures should be established for use throughout the Standard library. This will build consensus for whether it is better to abstract type components or whole pointed-to types, and will ensure that the signatures are easier to understand because they follow a coherent pattern. At the moment there is no formal convention because the needs of each library section are relatively distinct.

  Committee consensus has not yet settled on what kind of entity a generic function “is”. The eventual common notation should reflect this in a way that is clear and useful to the user and communicates the concrete type of any underlying addressable object or function clearly. The consensus was clear that abstracting only the qualifier component of a type did not achieve this.

Proposed wording

Changes are proposed against the wording in C23 draft n2912. Bolded text is new text.

bsearch

Modify 7.22.5.1 p1:

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

(make the return type and type of `base` qualifier-generic)

Modify the first sentence of 7.22.5.1 p2:

> The `bsearch` **generic** function searches an array of ...

Modify the first sentence of 7.22.5.1 p4:

> The `bsearch` **generic** function returns a pointer to ...

Add two new paragraphs after 7.22.5.1 p4:

> The `bsearch` function is generic in the qualification of the type pointed to by the argument to `base`. If this argument is a pointer to a `const`-qualified object type, the returned pointer will be a pointer to `const`-qualified `void`. Otherwise, the argument shall be a pointer to an unqualified object type or a null pointer constant footnote), and the returned pointer will be a pointer to unqualified `void`.
>
> footnote) If the argument is a null pointer and the call is executed, the behavior is undefined.
>
> The external declaration of `bsearch` has the concrete type: `void * (const void *, const void *, size_t, size_t, int (*) (const void *, const void *))`, which supports all correct uses. If a macro definition of this generic function is suppressed in order to access an actual function, the external declaration with this concrete type is visible.

7.24.5

Add three introductory paragraphs before the individual function descriptions, 7.24.5 p1:

> The stateless search functions in this section (`memchr`, `strchr`, `strpbrk`, `strrchr`, `strstr`) are *generic functions*. These functions are generic in the qualification of the array to be searched and will return a result pointer to an element with the same qualification as the passed array. If the array to be searched is `const`-qualified, the result pointer will be to a `const`-qualified element. If the array to be searched is not `const`-qualified footnote), the result pointer will be to an unqualified element.

footnote) the null pointer constant is not a pointer to a `const`-qualified type, and therefore the result expression has the type of a pointer to an unqualified element; however, evaluating such a call is undefined.

The external declarations of these generic functions have a concrete function type that returns a pointer to an unqualified element (of type `char` when specified as *`QChar`*, and `void` when specified as *`QVoid`*), and accepts a pointer to a `const`-qualified array of the same type to search. This signature supports all correct uses. If a macro definition of any of these generic functions is suppressed in order to access an actual function, the external declaration with the corresponding concrete type is visible.

The `volatile` and `restrict` qualifiers are not accepted on the elements of the array to search.

memchr

Modify 7.24.5.1 p1:

```
#include <string.h>
QVoid *memchr (QVoid *s, int c, size_t n);
```

Modify the first sentence of 7.24.5.1 p2:

The `memchr` **generic** function locates the first occurrence of ...

Modify 7.24.5.1 p3:

The `memchr` **generic** function returns a pointer to ...

strchr

Modify 7.24.5.2 p1:

```
#include <string.h>
QChar *strchr (QChar *s, int c);
```

Modify the first sentence of 7.24.5.2 p2:

The `strchr` **generic** function locates the first occurrence of ...

Modify 7.24.5.2 p3:

The `strchr` **generic** function returns a pointer to ...

strpbrk

Modify 7.24.5.4 p1:

```
#include <string.h>
QChar *strpbrk (QChar *s1, const char *s2);
```

Modify the first sentence of 7.24.5.4 p2:

The `strpbrk` **generic** function locates the first occurrence in ...

Modify 7.24.5.4 p3:

> The `strpbrk` **generic** function returns a pointer to ...

strrchr

Modify 7.24.5.5 p1:

```
#include <string.h>
QChar *strrchr (QChar *s, int c);
```

Modify the first sentence of 7.24.5.5 p2:

> The `strrchr` **generic** function locates the last occurrence of ...

Modify 7.24.5.5 p3:

> The `strrchr` **generic** function returns a pointer to ...

strstr

Modify 7.24.5.7 p1:

```
#include <string.h>
QChar *strstr (QChar *s1, const char *s2);
```

Modify the first sentence of 7.24.5.7 p2:

> The `strstr` **generic** function locates the first occurrence in ...

Modify the first sentence of 7.24.5.7 p3:

> The `strstr` **generic** function returns a pointer to ...

7.29.4.5

Add three introductory paragraphs before the individual function descriptions, 7.29.4.5 p1:

> The stateless search functions in this section (`wcschr`, `wcspbrk`, `wcsrchr`, `wmemchr`, `wcsstr`) are *generic functions*. These functions are generic in the qualification of the array to be searched and will return a result pointer to an element with the same qualification as the passed array. If the array to be searched is `const`-qualified, the result pointer will be to a `const`-qualified element. If the array to be searched is not `const`-qualified [footnote)], the result pointer will be to an unqualified element.
>
> footnote) the null pointer constant is not a pointer to a `const`-qualified type, and therefore the result expression has the type of a pointer to an unqualified element; however, evaluating such a call is undefined.
>
> The external declarations of these generic functions have a concrete function type that returns a pointer to an unqualified element of type `wchar_t`, and accepts a pointer to a `const`-qualified array of the same type to search. This signature supports all correct uses. If a macro definition of any of these generic functions is suppressed in order to access an actual function, the external declaration with this concrete type is visible.

The `volatile` and `restrict` qualifiers are not accepted on the elements of the array to search.

wcschr

Modify 7.29.4.5.1 p1:

```
#include <string.h>
QWchar_t *wcschr (QWchar_t *s, wchar_t c);
```

Modify the first sentence of 7.29.4.5.1 p2:

The `wcschr` **generic** function locates the first occurrence of ...

Modify 7.29.4.5.1 p3:

The `wcschr` **generic** function returns a pointer to ...

wcspbrk

Modify 7.29.4.5.3 p1:

```
#include <wchar.h>
QWchar_t *wcspbrk (QWchar_t *s1, const wchar_t *s2);
```

Modify the first sentence of 7.29.4.5.3 p2:

The `wcspbrk` **generic** function locates the first occurrence in ...

Modify 7.29.4.5.3 p3:

The `wcspbrk` **generic** function returns a pointer to ...

wcsrchr

Modify 7.29.4.5.4 p1:

```
#include <wchar.h>
QWchar_t *wcsrchr (QWchar_t *s, wchar_t c);
```

Modify the first sentence of 7.29.4.5.4 p2:

The `wcsrchr` **generic** function locates the last occurrence of ...

Modify 7.29.4.5.4 p3:

The `wcsrchr` **generic** function returns a pointer to ...

wcsstr

Modify 7.29.4.5.6 p1:

```
#include <wchar.h>
QWchar_t *wcsstr (QWchar_t *s1, const wchar_t *s2);
```

Modify the first sentence of 7.29.4.5.6 p2:

The `wcsstr` **generic** function locates the first occurrence in ...

Modify the first sentence of 7.29.4.5.6 p3:

The `wcsstr` **generic** function returns a pointer to ...

wmemchr

Modify 7.29.4.5.8 p1:

```
#include <wchar.h>
QWchar_t *wmemchr (QWchar_t *s, wchar_t c, size_t n);
```

Modify the first sentence of 7.29.4.5.8 p2:

The `wmemchr` **generic** function locates the first occurrence of ...

Modify 7.29.4.5.8 p3:

The `wmemchr` **generic** function returns a pointer to …

bsearch_s

Modify K.3.6.3.1 p1:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
QVoid *bsearch_s(const void *key, QVoid *base, rsize_t nmemb,
                 rsize_t size,
                 int (*compar)(const void *k, const void *y, void
*context),
                 void *context);
```

Modify the first sentence of K.3.6.3.1 p4:

The `bsearch_s` **generic** function searches an array of ...

Modify the first sentence of K.3.6.3.1 p6:

The `bsearch_s` **generic** function returns a pointer to ...

Add two new paragraphs after K.3.6.3.1 p6:

The `bsearch_s` function is generic in the qualification of the type pointed to by the argument to `base`. If this argument is a pointer to a `const`-qualified object type, the returned pointer will be a pointer to `const`-qualified `void`. Otherwise, the argument shall be a pointer to an unqualified object type or a null pointer constant footnote), and the returned pointer will be a pointer to unqualified `void`.

footnote) If the argument is a null pointer and the call is executed, the behavior is undefined.

The external declaration of `bsearch_s` has the concrete type: `void * (const void *, const void *, rsize_t, rsize_t, int (*) (const void *, const void *), void *)`, which supports all correct uses. If a macro

> definition of the generic function is suppressed in order to access an actual function, the external declaration with this concrete type is visible.

Optional changes

Obsolescence

Optionally, the Committee could make suppression of the generic functions' macro implementations an obsolescent feature to discourage future use.

Add a footnote at the end of the four "the external declaration(s)" paragraphs:

> footnote) This is an obsolescent feature.

Add a new paragraph to the end of 7.32.14:

> Suppressing the macro definition of `bsearch` in order to access the actual function is an obsolescent feature.

Add a new paragraph to the end of 7.32.15:

> Suppressing the macro definitions of `memchr`, `strchr`, `strpbrk`, `strrchr`, or `strstr` in order to access the corresponding actual function is an obsolescent feature.

Add a new paragraph to the end of 7.32.18:

> Suppressing the macro definitions of `wcschr, wcspbrk, wcsrchr, wmemchr`, or `wcsstr` in order to access the corresponding actual function is an obsolescent feature.

References

C17

n2912 C23 Working Draft

n2978 Introduce the nullptr constant v4

n2522 A Common C/C++ Core Specification rev 2

C++

Embedded C

n2526 use const for data from the library that shall not be modified

n2853 The `void`-*which-binds*: typesafe parametric polymorphism