Improve the allocator free functions to accept const qualified heap object pointers

Reply-to: davidmbanham at gmail dot com
Document number: N3894
Date: 2026-05-28

Summary of Changes

- Initial proposal

Problem Statement

The allocator's *`free()`* family of related functions do not accept a pointer that is referring to a *`const`* qualified heap object.

For example, the call to `free()` in the following code snippet results in diagnostic messages ("warnings") from gcc, clang, and msvc with their default diagnostic configuration:

```
const char* cstr = strdup("Hello");
…
free(cstr);
```

There are various motivations for working with immutable objects held on the heap, but typically they represent the programmer's desire to have the compiler (and other program analysers) report where the program code is potentially able to modify the object's value; that is the benefit afforded by `const` qualification after all. Such a situation can arise where data is loaded at runtime into a mutable heap object and thereafter this object is treated as an immutable one. In other words, the object's ownership is transferred from a mutable object pointer to an immutable object pointer.

The set of heap object *deallocating* functions includes:

```
void free(void *ptr);
void free_sized(void *ptr, size_t size);
void free_aligned_sized(void *ptr, size_t alignment, size_t size);
```

We could also consider that the object pointer that `realloc` uses should also be `const` qualified since it will copy the object's data to a newly allocated one and deallocate the original object.

```
void *realloc(void *ptr, size_t size);
```

Analysis

ISO 9899 (2024) states in §6.5.3.3 "Function calls" that

> An argument can 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.
> …

> 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.

In practice this means that a parameter's qualification is *ignored* when function call arguments are assigned to the function parameter's object. A `const` qualified parameter is mutable when the function is called but immutable to the function's body.

For pointer parameters, it is the top-level qualification that is impacted. That is, the qualification that applies to the pointer value and not to the object the pointer refers to.

For example, passing a `const` qualified pointer to `free()` is conforming as the top-level qualification is ignored:

```
char* const str = strdup("Mutable");
…
free(str);
```

Whereas the qualification of the pointer's referent object is not ignored, hence

```
const char* cstr = strdup("Immutable");
…
free(cstr);
```

results in a diagnostic for discarding the `const` qualifier of the object the pointer is referring to.

The vast majority of the standard library functions that take a pointer object as a read-only source pointer object, declare the parameter to be a pointer to `const T`. For example (with underlining for emphasis):

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
                                                    FILE * restrict stream);
int printf(const char * restrict format, ...);
int atoi(const char *nptr);
```

Bazley describes the more general problem in N3674, where they state:

> In a C program, any object can be accessed through a `const`-qualified lvalue, but only mutable objects can be modified through a modifiable lvalue without invoking undefined behaviour. An example of an immutable object is an object defined with a `const`-qualified type. A pointer to an unqualified referenced type which may hold the address of an immutable object is a maintenance hazard.

The common workaround is to remove the qualification with a cast, for example:

```
free((void*)cstr);
```

Cast-based suppression can hide real type/ownership mistakes and reduce diagnostics quality. E.g., if the object `cstr` was in fact a double indirection pointer, then the above cast would be incorrect and undefined behaviour would potentially result.

Another workaround is to configure the compiler to not report when qualifiers are discarded, for example, gcc has the `–Wno-discarded-qualifiers` option. However, the effect of this option is

to disable all reporting of discarded qualifiers and not just those when `free()` is called, which as Bazley notes is "a maintenance hazard".

Compatibility

There is minimal impact for ordinary call sites; some declarations and function-pointer use may require adjustment.

The change is fully ABI compatible as a `const` qualified object pointer parameter can refer to both mutable and immutable objects. Implementations may need to adjust their underlying code, although a simple fix is simply to remove the parameter's `const` qualifier upon entry to the affected functions.

However, the proposed change will change the function pointer type compatibility for this function. For example, the following line of code will result in a constraint violation diagnostic were the proposed change to be accepted:

```
void (*fp)(void *) = free;
```

A paper from 2020, N2526, proposed changes to four functions, `getenv`, `localeconv`, `setlocale`, and `strerror` to add the "`const`" qualifier, so they return an immutable "`const char *`" instead of a mutable "`char *`". The Austin Group subsequential issued a "rebuttal" of it (N2565) on the grounds that the change would break extant code, where they cite the first principle from the then C Charter (circa 2020):

> **Existing code is important, existing implementations are not.** A large body of C code exists of considerable commercial value. Every attempt has been made to ensure that the bulk of this code will be acceptable to any implementation conforming to the Standard. The Committee did not want to force most programmers to modify their C programs just to have them accepted by a conforming translator.

The current C Charter (N3280) does not state this as a principle, although, clearly, we should not be introducing breaking changes without some assessment of the how severe the impact is versus the perceived benefit. In the case of this proposal, the suggestion to the committee is that there is more to be gained from accepting the change than from the possible *pain* resulting from function pointer type conflicts for the `free` family of functions. This would be in the spirit of the current C Charter's principle for "Codify existing practice to address evident deficiencies".

Nevertheless, the N2565 paper proposes the use of codesearch.debian.net to check for extant usage in the large Debian codebase. The Debian codebase may be representative of desktop and server class software, but not of embedded software.

A regular expression search for "filetype:c =\s*?free\s*?;" (i.e., files with .h and .c extensions) returned 1,169 results from a grep scan of 970,108 files. An appraisal of the first ten results showed that whilst there were some clear uses of the C Standard library `free` function as a function pointer there were other uses of free as an identifier that were clearly not the standard library `free` function, and there is no way to separate these usages with a simple regular expression search. Moreover, there were C++ source files in the results that can be ignored too.

A small survey of twenty search results indicated that 13 had a high certainty of referring to the C Standard Library free (the other assignment expressions seemingly being a reuse of the identifier for other purposes, such as a free list count). If this is representative across all of the results then there are 760 assignments of the free function pointer in 970,108 C files, about 1 in every 1,276 files.
An example that is clearly (that is, with reasonable certainty) referring to the C Standard library `free` function in an assignment expression is:
gnupg2_2.4.9-4/tests/gpgscm/scheme.c

```
  sc->gensym_cnt=0;
  sc->malloc=malloc;
  sc->free=free;
  sc->sink = &sc->_sink;
  sc->NIL = &sc->_NIL;
```

An example where the use of a `free` identifier that is clearly not the same as the C Standard library `free` function is:
e2fsprogs_1.47.4-1/misc/fuse2fs.c

```
    buf->f_blocks = ext2fs_blocks_count(fs->super) - overhead;
      buf->f_bfree = free;
      if (free < reserved)
            buf->f_bavail = 0;
```

Whereas a search for `free()` call sites with the search expression “filetype:c \s+free\(” returns 726,993 search results in 253,569 files. (There is no obvious reason why the total number of files searched should be different from the search for it in assignment expressions, but it is; possibly due to a hidden “feature” of the tool.) That’s about 2.86 call sites for `free()` in every file. (Examining twenty of these results shows that they are all single argument `free(…)` call sites, which gives a reasonable confidence that the call is to the C Standard library function of that name. The use of a more sophisticated search expression potentially improves this confidence at the risk of it not being fully inclusive of all of the syntactic possibilities. For example, the search expression “\s+free\([a-zA-Z_\->]+?\) filetype:c”) returns 546,927 results in the same number of files, or about 2.1 calls per file.)

In summary, the Debian Code Search facility shows that the relative use of explicit (i.e. not counting any indirect invocations) `free()` function calls to the assignment of `free` function pointers places the latter in a very small minority. The cost impact for such projects switching to an implementation of a new version of the C standard with the proposed change is small, should they wish to do so.

Proposed wording

In the following, green underlined text is for insertion and ~~red strikethrough~~ text is for deletion. Section numbers refer to the ISO 9899:2024 document.

7.24.4.4 The free function
Synopsis

```
#include <stdlib.h>
void free(const void *ptr);
```

7.24.4.5 The free_sized function
Synopsis
```
#include <stdlib.h>
void free_sized(const void *ptr, size_t size);
```

7.24.4.6 The free_aligned_sized function
Synopsis
```
#include <stdlib.h>
void free_aligned_sized(const void *ptr, size_t alignment, size_t size);
```

7.24.4.8 The realloc function
Synopsis
```
#include <stdlib.h>
void *realloc(const void *ptr, size_t size);
```

B.23 General utilities <stdlib.h>
```
...
void free(const void *ptr);
void free_sized(const void *ptr, size_t size);
void free_aligned_sized(const void *ptr, size_t alignment, size_t size);
void *malloc(size_t size);
void *realloc(const void *ptr, size_t size);
...
```

Straw Poll questions

The following questions are suggested as the basis for a straw poll on this paper:

1. Does WG14 accept the change proposed by N3894 to the function type for `free()`, `free_sized()`, and `free_aligned_sized()`?
2. Does WG14 accept the change proposed by N3894 to the function type for `realloc()`?

Should there be insufficient consensus to proceed, then the following along-the-lines poll is requested:

1. Would WG14 like to see a paper along the lines of N3894 that proposes an alternative `free()` function that accepts `const` qualified object pointers?