

Slaying an Aqueous Demon: Writable String Literals, v2

Document: n3959

Author: Martin Uecker

Charter Principles (n3280):

- Enable secure programming
- Enable functional safety
- Avoid ambiguities
- Avoid quite changes
- Codify existing practice to address evident deficiencies_

Prior art: -Wwritable-strings, C++

ChangeLog:

N3959:

- Add impact analysis (Debian package rebuilding)
- Add implicit conversion to mitigate const-correctness issue (following C++'s path)
- Reworded for clarity.

N3860:

- Initial version

Background:

String literals in C are not const-qualified, which is a type-safety issue that can lead to undefined behavior. Modifying a string literal has undefined behavior. On many systems this results in a runtime trap, because string literals are placed in write-protected memory. Therefore, while it may not constitute a critical memory-safety issue on such systems, it remains problematic and can be more serious on systems without memory protection.

<https://godbolt.org/z/We7qvxxvvr>

```
int main()
{
    "aa"[0] = 1;
}
```

<source>: In function 'main':

<source>:6:5: warning: assignment of read-only location '"aa"[0]'

```
6 |     "aa"[0] = 1;
  |     ^~~~
```

Execution build compiler returned: 0

Program returned: 139

Program terminated with signal: SIGSEGV

In C++, string literals are const-qualified. Many C compilers provide an option to emulate this behavior (e.g., GCC and Clang's -Wwrite-strings). The name is unfortunate, because it does more than issue warnings: it also changes the type of string literals to be const-qualified.

Proposal:

The proposal is to make string literals const-qualified. This does not eliminate the undefined behavior (modifying an object that is defined as const-qualified remains undefined) but it ensures such modifications cannot occur so long as the program maintains type safety, that is, it avoids unsafe casts and conversions. Safety- and security-oriented C programming already avoids casts; when a cast is used in exceptional cases, it serves as a visual warning sign (much like Rust's `unsafe` keyword). In the future, compilers could also provide stronger diagnostics that flag all unsafe casts, potentially as a required component of a memory-safety mode

(note: updated) Unfortunately, because too much existing code would break, the proposal includes an implicit conversion from a string literal to an unqualified pointer type. As long as this implicit conversion is silently accepted, there is no immediate safety benefit. Still, the change opens a path forward: in a future version of the language, this unsafe conversion could be removed. This mirrors C++'s transition: C++98 and C++03 retained the implicit conversion, which was then removed in C++11. Even if the conversion can never be removed in C because too much code relies on it, the underlying type change would still improve safety. First, an optional compiler warning could simply diagnose the implicit conversions where they occur without changing program semantics, addressing the current limitations of `-Wwrite-strings` that hinder its widespread use. Second, such diagnostics could more readily be required in a future opt-in safe mode that remains fully compatible with the base standard. It is also worth noting that an increasing amount of code depends on constructs like `_Generic` or `typeof`, where the exact type matters, and that C2y is likely the last opportunity to make this underlying semantic change.

Package Rebuilding

To evaluate the impact on existing code, we rebuilt 37,635 source packages from the Debian archive. From these, we selected C projects for which we could modify the compiler flags and that built successfully with `-Werror=discarded-qualifiers`. In total, 10,937 packages built successfully and were used for further analysis. Those packages were then rebuilt with `-Wwrite-strings` as an additional flag, which changes the type of string literals to be const-qualified. We also rebuilt the failing packages with `-Wwrite-strings -Wno-error=discarded-qualifiers`. The results are shown in the following table.

10937	100,00%	Successfully build with <code>-Werror=discarded-qualifiers</code>
3999	36,56%	<code>-Wwrite-strings -Werror=discarded-qualifiers</code>
9	0,08%	<code>-Wwrite-string -Wno-error=discarded-qualifiers</code>
3	0,03%	<code>_Generic</code>
3	0,03%	cast to union member extension
3	0,03%	unknown

(Note that no attempt was made to determinate how many of these package are strictly conforming to ISO C23.)

Impact on Existing Code

While adopting this in ISO C2y would be a breaking change, many programs already maintain const-correctness. When the change does trigger diagnostics, they are almost always harmless, because they arise from a conversion that many C compilers have historically treated as a warning rather than an error. In the experiment described above, about one third of packages exhibited such diagnostics when string literals were made const-qualified, which suggests that an implicit conversion is needed to avoid producing an excessive number of warnings.

If we disregard the warnings that the implicit conversion would suppress, only nine packages fail to build due to the actual type change. In three related packages, the failure is caused by a direct test using `_Generic`; in three others, by a cast to a union member via a compiler extension (which compilers that support this extension could continue to accept); and in the remaining three, the cause could not be determined from the build logs. Overall, this appears acceptable (e.g. compared to similar changes made in C23 to the library or other changes to the language).

Existing Alternative

There may be rare scenarios in which string literals are writable on a given platform, and programs intentionally exploit this. Such platforms could continue to support the behavior as an extension via a compiler flag. However, it is also worth noting that, with compound literals, ISO C23 provides a standard-compliant way to create writable strings without relying on undefined behavior using a writable character array initialized from a string literal as in the following example.

```
(static char[]){ "aa" }[0] = 1;
```

<https://godbolt.org/z/zqhxP4Gas>

Acknowledgments: David Svoboda and Aaron Ballman for encouragement.

Wording Change (N3858)

6.4.6 String literals Semantics

6 In translation phase 7 (5.2.1.2), a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.⁷⁵⁾ The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. ~~For~~ Ordinary string literals, ~~the array elements have~~ **are unnamed objects defined with** type **array of const** char, and ~~their elements~~ are initialized with the individual bytes of the multibyte character sequence corresponding to the literal encoding (6.2.9). ~~For~~ UTF-8 string literals, ~~the array elements have~~ **are unnamed objects defined with** type **array of const** char8_t, and ~~their elements~~ are initialized with the characters of the multibyte character sequence, as encoded in UTF-8. ~~For~~ Wide string literals prefixed by the letter L, ~~the array elements have~~ **are unnamed objects defined with** type **array of const** wchar_t and ~~their elements~~ are initialized with the sequence of wide characters corresponding to the wide literal encoding. For wide string literals prefixed by the letter u or U, the array ~~elements have~~ **are unnamed objects defined with** type **array of const** char16_t or char32_t, respectively, and ~~their elements~~ are initialized with the sequence of wide characters corresponding to UTF-16 and UTF-32 encoded text, respectively. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined. Any hexadecimal escape sequence or octal escape sequence specified in a u8, u, or U string specifies a single char8_t, char16_t, or char32_t value and can result in the full character sequence not being valid UTF-8, UTF-16, or UTF-32.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values.^{xxx} ~~If the program attempts to modify such an array, the behavior is undefined.~~

xxx) If the program attempts to modify such an array, the behavior is undefined. See 6.7.4.1.

6.5.17.2 Simple assignment

Constraints

1 One of the following shall hold:⁹⁶⁾

- ...

- **The left operand has atomic, qualified, or unqualified pointer type to char, char8_t, char16_t, char32_t, or wchar_t, and the right operand is a string literal with the corresponding const-qualified array type (before conversion to a pointer as described in 6.3.3.1).**^{yyy)}

yyy) The implicit conversion allows the removal of the const qualifier. This implicit conversion is an obsolescent language feature.

6.11 Future language directions

6.11.X String literals

The implicit conversion of string literals to pointers with unqualified character type is an obsolescent feature.