ISO/IEC JTC 1/SC 22/WG14 N 2068

August 20, 2016

v 1

Type generic string interfaces
honor the const contract of application code

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

In several places, C library functions break the **const** contract of user code by returning an unqualified pointer to a **const** qualified object. This situation arises because these library functions are meant to deal with both, **const** qualified and unqualified pointer targets. With C11's **_Generic** such a break of const can easily avoided and it can be used to provide type generic interfaces that are **const** correct. As additional fall out, such type generic interfaces can combine the functionality of narrow and wide character support functions.

Introduction

The following string search functions in `<string.h>` are potentially dangerous, since they break the **const** contract of application code. They return a pointer that is not **const**-qualified which is pointing to a **const** qualified object that is received as a pointer parameter:

```
void *memchr(void const *s, int c, size_t n);
char *strchr(char const *s, int c);
char *strpbrk(char const *s1, const char *s2);
char *strrchr(char const *s, int c);
char *strstr(char const *s1, const char *s2);
```

If the application code stores the returned pointer in a variable and uses this later this can lead to two types of problems:

— Writing to read only memory. The result would be a runtime crash of the program.
— A subtle unnoticed modification of an object that is received through a **const**-qualified pointer, and thus a break of an interface contract.

In any case writing to such an object has undefined behavior.

Tracking provenance of pointers to objects to remember if an object is writable or not is a tedious enterprise. **const**-qualification had been introduced to C to avoid the need of such provenance tracking and C library functions that break this contract are jeopardizing the whole idea.

Goal 1. *Ensure that all C library interfaces honor the* **const** *contract.*

As a simple fall out of using type generic functions for `str*` and `mem*` functions is that we also can use these to unify functions for **char*** and **wchar_t**, much as `<tgmath.h>` provides type generic interfaces for the functions in `<math.h>`.

Goal 2. *Provide type generic interfaces for narrow and wide string functions.*

The string-to-number conversion functions from `<stdlib.h>` and `<inttypes.h>` such as

```
double strtod(const char * restrict nptr, char ** restrict endptr);
```

exhibit an even worse problem, because `endptr` is a pointer to pointer that is forcibly the wrong type if the origin of `nptr` had been **const** qualified: if `endptr` is non-null and the string is successfully scanned, the function stores `nptr+x` (of type **const char***) into the pointer `*endptr` (of type **char***). If the **const** properties of `endptr` are not correctly tracked by the application code, the application can be tempted to write into a **const** qualified object and thereby encounter undefined behavior.

Previous versions of the C standard followed that strategy for the interface specification because they wanted to ensure that these functions can be called for both types of strings, those that are **const** qualified and those that are or not. This was necessary because many of these functions existed before **const** was even integrated into C, backwards compatibility was a prime objective, and adding qualifiers to pointer targets was the only way to allow for qualified and unqualified arguments.

This situation has changed with C11. Because it added type generic selection with the keyword **_Generic**, type generic interfaces can now easily be added to the standard and help applications to keep their **const** contracts more easily.

In C++, from which C inherited **const** qualification, such functions would be overloaded with different interfaces that distinguish according to the **const**-qualification of their argument.

```
1  #ifdef __cplusplus
2  void *memchr(void *s, int c, size_t n);
3  void const*memchr(void const *s, int c, size_t n);
4
5  char *strchr(char *s, int c);
6  char const*strchr(char const *s, int c);
7
8  char *strpbrk(char *s1, const char *s2);
9  char const*strpbrk(char const *s1, const char *s2);
10
11 char *strrchr(char *s, int c);
12 char const*strrchr(char const *s, int c);
13
14 char *strstr(char *s1, const char *s2);
15 char const*strstr(char const *s1, const char *s2);
16
17 double strtod(char *nptr, char **endptr);
18 double strtod(const char *nptr, char const**endptr);
19 #endif
```

A solution for C using _Generic

With **_Generic** we now have the possibility to have a similar interface for C. We can provide macros of the same name as the standard library functions and provide the semantics of function overloading.

At the same time that we overcome the **const** qualification problem, we may also simplify the use of string functions even more. Most string functions and byte functions are copied by interfaces for wide character strings, that have analogous interfaces with **wchar_t** instead of **char**. For them, equivalent interfaces can be provided that also map the functionalities to the corresponding `str` names.

The following macro selects one of seven function pointers according to the first argument being

— a **void** pointer,
— a narrow character pointer, or
— a wide character pointer

and according to **const** qualification.

tgstring.h

```
44 /**
45  ** @brief An internal macro to select a const conserving string
```

```
 ** function.
 **
 ** This macro simply selects the function that corresponds to
 ** the type of argument @a X. It is not meant to be used
 ** directly but to be integrated in specific macros that
 ** overload C library functions.
 **
 ** The argument @a X must correspond to an array or pointer,
 ** otherwise an error is triggered in the controlling
 ** expression.
 **
 ** If any of the 7 cases makes no sense for the type generic
 ** interface in question, that case can be switched off by
 ** passing a 0 instead of a function. A use of that case in user
 ** code then results in a compile time error.
 **
 ** @see memchr on how to integrate this into a type generic user
 ** interface
 **/
# define _TG_TO_CONST(DF, VN, VC, SN, SC, WN, WC, X, ...)       \
_Generic(&(X[0]),                                               \
         default:         DF,                                   \
         void*:           VN,                                   \
         void const*:     VC,                                   \
         char*:           SN,                                   \
         char const*:     SC,                                   \
         wchar_t*:        WN,                                   \
         wchar_t const*: WC                                     \
         )((X), __VA_ARGS__)
```

Then we add auxiliar defintions such as the following two:

tgstring.h

```
void const*memchr_const(void const* _s,
                        int _c, size_t _n) {
  return memchr(_s, _c, _n);
}
```

tgstring.h

```
wchar_t const*wmemchr_const(wchar_t const _s[static 1],
                            wchar_t _c, size_t _n) {
  return wmemchr(_s, _c, _n);
}
```

Here, these functions can be of some specialized storage class, or attribute such as **static inline**, **register** or `_Attribute(_Always_inline)` if such features are included in C2x.

Then we define **memchr** as a macro:

tgstring.h

```
# undef memchr
# define memchr(...)                                    \
  _TG_TO_CONST(memchr,          /* default        */     \
               memchr,          /* void*          */     \
               memchr_const,    /* const void*    */     \
```

```
95                memchr,          /* char*          */      \
96                memchr_const,    /* const char*    */      \
97                wmemchr,         /* wchar_t*       */      \
98                wmemchr_const,   /* const wchar_t* */      \
99                __VA_ARGS__)
```

In this case of a "mem" function a **default** fallback makes sense. Any pointer type should still fallback to the **void*** variant.

In contrast to that "str" functions suppose that their arguments point to strings, that is to null terminated character arrays. With analogous definitions, for **strstr** we can force errors for the usage for the first three cases:

tgstring.h

```
149 # undef strrchr
150 # define strrchr(...)                                          \
151   _TG_TO_CONST(0, 0, 0,  /* the first three cases are errors */ \
152                strrchr, strrchr_const, wcsrchr, wcsrchr_const,  \
153                __VA_ARGS__)
```

The functions that should be covered by such type generic macros are: **memchr**, **memcmp**, **memcpy**, **memmove**, **memset**, **strlen**, **strncat**, **strncmp**, **strncpy**, **strpbrk**, **strrchr**, **strspn**, **strstr**, **strtod**, **strtof**, **strtok**, **strtol**, **strtold**, **strtoll**, **strtoul**, **strtoul**, **strtoull**, **strtoumax**, **wcsncmp**, **wcsncpy**, **wcspbrk**, **wcsrchr**, **wcsspn**, **wcsstr**, **wcstod**, **wcstof**, **wcstok**, **wcstol**, **wcstold**, **wcstoll**, **wcstoul**, **wcstoul**, **wcstoull**, **wmemchr**, **wmemcmp**, **wmemcpy**, **wmemmove**, and **wmemset**.

Appendix: a reference implementation

We provide a reference implementation of the proposed header that is intended to be usable with any compliant C compiler and that doesn't need any addition to an existing C library. The license for this is free, so it can be directly used by C library implementations.

```
1  #ifndef __STDC_TGSTRING__
2  # define __STDC_TGSTRING__ 201610L
3
4  /**
5   ** @file
6   ** @brief Type generic string interfaces
7   **
8   ** @copyright 2016 Jens Gustedt
9   **
10  ** This code is distributed under a Creative Commons Attribution
11  ** 4.0 International License.
12  **/
13
14 # if __STDC_VERSION__ < 201112L
15 #  error "tgstring.h needs a C11 conforming compiler"
16 # endif
17
18 # include <inttypes.h>
19 # include <stdlib.h>
20 # include <stdio.h>
21 # include <string.h>
22 # include <wchar.h>
23
24 /* We define a bunch of inline functions that have the only
```

```
25    purpose of providing conversions to the their arguments and
26    then to pass them on to the corresponding standard
27    function. We use the same names as the standard function and
28    append "_const" to them. This use is covered by the standard
29    since all of the "str" and "wcs" names are reserved,
30    anyhow. */
31
32 /* If the register extension is included in the standard, this
33    would be an ideal candidate. */
34 # if __STDC_REGISTER_IN_FUNCTION_SCOPE__
35 #  define _TG_INLINE register
36 /* Otherwise, have your platform specific definition, here*/
37 # elif __GNUC__
38 #  define _TG_INLINE __attribute__((always_inline)) inline
39 /* Fall back to static inline, if none of that worked */
40 # else
41 #  define _TG_INLINE static inline
42 # endif
43
44 /**
45  ** @brief An internal macro to select a const conserving string
46  ** function.
47  **
48  ** This macro simply selects the function that corresponds to
49  ** the type of argument @a X. It is not meant to be used
50  ** directly but to be integrated in specific macros that
51  ** overload C library functions.
52  **
53  ** The argument @a X must correspond to an array or pointer,
54  ** otherwise an error is triggered in the controlling
55  ** expression.
56  **
57  ** If any of the 7 cases makes no sense for the type generic
58  ** interface in question, that case can be switched off by
59  ** passing a 0 instead of a function. A use of that case in user
60  ** code then results in a compile time error.
61  **
62  ** @see memchr on how to integrate this into a type generic user
63  ** interface
64  **/
65 # define _TG_TO_CONST(DF, VN, VC, SN, SC, WN, WC, X, ...)       \
66 _Generic(&(X[0]),                                               \
67          default:        DF,                                    \
68          void*:          VN,                                    \
69          void const*:    VC,                                    \
70          char*:          SN,                                    \
71          char const*:    SC,                                    \
72          wchar_t*:       WN,                                    \
73          wchar_t const*: WC                                     \
74          )((X), __VA_ARGS__)
75
76 // void *memchr(void const *s, int c, size_t n);
77
78 _TG_INLINE
79 void const*memchr_const(void const* _s,
80                         int _c, size_t _n) {
81   return memchr(_s, _c, _n);
```

```
82 }
83
84 _TG_INLINE
85 wchar_t const*wmemchr_const(wchar_t const _s[static 1],
86                             wchar_t _c, size_t _n) {
87   return wmemchr(_s, _c, _n);
88 }
89
90 # undef memchr
91 # define memchr(...)                                    \
92   _TG_TO_CONST(memchr,          /* default        */     \
93                memchr,          /* void*          */     \
94                memchr_const,    /* const void*    */     \
95                memchr,          /* char*          */     \
96                memchr_const,    /* const char*    */     \
97                wmemchr,         /* wchar_t*       */     \
98                wmemchr_const,   /* const wchar_t* */     \
99                __VA_ARGS__)
100
101 // char *strchr(char const *s, int c);
102
103 _TG_INLINE
104 char const*strchr_const(char const *_s, int _c) {
105   return strchr(_s, _c);
106 }
107
108 _TG_INLINE
109 wchar_t const*wcschr_const(wchar_t const *_s, wchar_t _c) {
110   return wcschr(_s, _c);
111 }
112
113 # undef strchr
114 # define strchr(...)                                          \
115   _TG_TO_CONST(0, 0, 0,                                       \
116                strchr, strchr_const, wcschr, wcschr_const,    \
117                __VA_ARGS__)
118
119 // char *strpbrk(char const *_s, char const *_t);
120
121 _TG_INLINE
122 char const*strpbrk_const(char const *_s, char const *_t) {
123   return strpbrk(_s, _t);
124 }
125
126 _TG_INLINE
127 wchar_t const*wcspbrk_const(wchar_t const *_s, wchar_t const *_t) {
128   return wcspbrk(_s, _t);
129 }
130
131 # undef strpbrk
132 # define strpbrk(...)                                         \
133   _TG_TO_CONST(0, 0, 0,                                       \
134                strpbrk, strpbrk_const, wcspbrk, wcspbrk_const,\
135                __VA_ARGS__)
136
137 // char *strrchr(char const *s, int c);
138
```

```
139 _TG_INLINE
140 char const* strrchr_const(char const *_s, int _c) {
141   return strrchr(_s, _c);
142 }
143
144 _TG_INLINE
145 wchar_t const* wcsrchr_const(wchar_t const *_s, wchar_t _c) {
146   return wcsrchr(_s, _c);
147 }
148
149 # undef strrchr
150 # define strrchr(...)                                          \
151   _TG_TO_CONST(0, 0, 0,  /* the first three cases are errors */ \
152                strrchr, strrchr_const, wcsrchr, wcsrchr_const,  \
153                __VA_ARGS__)
154
155
156 // char *strstr(char const *_s, const char *_t);
157
158 _TG_INLINE
159 char const*strstr_const(char const *_s, char const *_t) {
160   return strstr(_s, _t);
161 }
162
163 _TG_INLINE
164 wchar_t const*wcsstr_const(wchar_t const *_s, wchar_t const *_t) {
165   return wcsstr(_s, _t);
166 }
167
168 # undef strstr
169 # define strstr(...)                                           \
170   _TG_TO_CONST(0,              /* default type: error */        \
171                0,              /* void*: error        */        \
172                0,              /* const*: error       */        \
173                strstr,         /* char*               */        \
174                strstr_const,   /* const char*         */        \
175                wcsstr,         /* wchar_t*            */        \
176                wcsstr_const,   /* const wchar_t*      */        \
177                __VA_ARGS__)
178
179 /**
180  ** @brief An internal macro to select a memory function
181  ** according to the base type @c void* or @c wchar_t*.
182  **
183  ** This macro simply adds the "w" prefix to the name of the
184  ** function, if a @c wchar_t function is to be called. All other
185  ** cases are mapped to the @c void*.
186  **
187  ** The argument @a X must correspond to an array or pointer,
188  ** otherwise an error is triggered in the controlling
189  ** expression.
190  **
191  ** @see memcpy on how to integrate this into a type generic user
192  ** interface
193  **/
194 # define _TG_MEM(NAME, X, ...)                                 \
195 _Generic(&(X[0]),                                              \
```

```
196           default:              NAME,              \
197           wchar_t*:         w ## NAME,             \
198           wchar_t const*: w ## NAME                \
199           )((X), __VA_ARGS__)
200
201  # undef memcpy
202  # define memcpy(...) _TG_MEM(memcpy, __VA_ARGS__)
203  # undef memmove
204  # define memmove(...) _TG_MEM(memmove, __VA_ARGS__)
205  # undef memcmp
206  # define memcmp(...) _TG_MEM(memcmp, __VA_ARGS__)
207  # undef memset
208  # define memset(...) _TG_MEM(memset, __VA_ARGS__)
209
210  /**
211   ** @brief An internal macro to select a string function
212   ** according to the base type @c char or @c wchar_t.
213   **
214   ** This macro simply adds the "wcs" prefix to the name of the
215   ** function, if a @c wchar_t function is to be
216   ** called. Traditional @c char strings are mapped to @c char.
217   **
218   ** There is no default case, because these functions need a
219   ** string and not some arbitrary bytes.
220   **
221   ** The argument @a X must correspond to an array or pointer,
222   ** otherwise an error is triggered in the controlling
223   ** expression.
224   **
225   ** No precaution is made concerning @c const. If a @c const
226   ** qualified pointer type is passed in and the function that is
227   ** selected does not accept such an argument, an error is
228   ** diagnosed as if the corresponding function would have been
229   ** used directly.
230   **
231   ** @see strcpy on how to integrate this into a type generic user
232   ** interface
233   **/
234  # define _TG_STR(NAME, X, ...)                    \
235  _Generic(&(X[0]),                                 \
236           char*: str ## NAME,                      \
237           char const*: str ## NAME,                \
238           wchar_t*: wcs ## NAME,                   \
239           wchar_t const*: wcs ## NAME              \
240           )((X), __VA_ARGS__)
241
242  # undef strcpy
243  # define strcpy(...) _TG_STR(cpy, __VA_ARGS__)
244  # undef strncpy
245  # define strncpy(...) _TG_STR(ncpy, __VA_ARGS__)
246  # undef strcat
247  # define strcat(...) _TG_STR(cat, __VA_ARGS__)
248  # undef strncat
249  # define strncat(...) _TG_STR(ncat, __VA_ARGS__)
250  # undef strcmp
251  # define strcmp(...) _TG_STR(cmp, __VA_ARGS__)
252  # undef strcoll
```

```
253 # define strcoll(...) _TG_STR(coll, __VA_ARGS__)
254 # undef strncmp
255 # define strncmp(...) _TG_STR(ncmp, __VA_ARGS__)
256 # undef strxfrm
257 # define strxfrm(...) _TG_STR(xfrm, __VA_ARGS__)
258 # undef strcspn
259 # define strcspn(...) _TG_STR(cspn, __VA_ARGS__)
260 # undef strspn
261 # define strspn(...) _TG_STR(spn, __VA_ARGS__)
262 # undef strntok
263 # define strntok(...) _TG_STR(ntok, __VA_ARGS__)
264 # undef strlen
265 # define strlen(...) _TG_STR(len, __VA_ARGS__)
266
267
268 /**
269  ** @brief An internal Xmacro to generate const conserving string
270  ** to floating point conversion function.
271  **
272  ** In the case of the string to number functions the original C
273  ** library functions cannot be used directly, because their
274  ** second argument is an unqualified pointer to pointer.
275  **
276  ** @param T is the target type of the conversion
277  ** @param C is the base type of the string, so @c char or @c
278  ** wchar_t
279  ** @param NAME is the base name for the function
280  **/
281 #define _TG_STR_F_FUNC_(T, C, NAME)                                  \
282 _TG_INLINE                                                           \
283 T _TG_ ## NAME ## _c(C const* _p, C const** _e) {                    \
284   return NAME(_p, (C**)_e);                                          \
285 }                                                                    \
286 _TG_INLINE                                                           \
287 T _TG_ ## NAME ## _nc(C* _p, C** _e) {                               \
288   return NAME(_p, _e);                                               \
289 }                                                                    \
290 typedef T _TG_ ## NAME ## _c_ftype(C const*, C const**);             \
291 typedef T _TG_ ## NAME ## _nc_ftype(C*, C**)
292
293 /**
294  ** @brief An internal Xmacro to generate const conserving string
295  ** to floating point conversion function.
296  **
297  ** This macro just assembles _TG_STR_F_FUNC_() for the two case
298  ** of @a C being @c char or @c wchar_t.
299  **
300  ** @param T is the target type of the conversion
301  ** @param NAME is the base name for the function
302  **/
303 #define _TG_STR_F_FUNC(T, NAME)                                      \
304   _TG_STR_F_FUNC_(T, char, str ## NAME);                             \
305   _TG_STR_F_FUNC_(T, wchar_t, wcs ## NAME)
306
307
308 /* generate the three groups of floating point conversion functions
     */
```

```
309 _TG_STR_F_FUNC(float, tof);
310 _TG_STR_F_FUNC(double, tod);
311 _TG_STR_F_FUNC(long double, told);
312
313
314
315 /**
316  ** @brief An internal Xmacro to generate const conserving string
317  ** to integer conversion function.
318  **/
319 #define _TG_STR_I_FUNC_(T, C, NAME)                                  \
320 _TG_INLINE                                                           \
321 T _TG_ ## NAME ## _c(C const* _p, C const** _e, int _b) {            \
322   return NAME(_p, (C**)_e, _b);                                      \
323 }                                                                    \
324 _TG_INLINE                                                           \
325 T _TG_ ## NAME ## _nc(C* _p, C** _e, int _b) {                       \
326   return NAME(_p, _e, _b);                                           \
327 }                                                                    \
328 typedef T _TG_ ## NAME ## _c_ftype(C const*, C const**, int);        \
329 typedef T _TG_ ## NAME ## _nc_ftype(C*, C**, int)
330
331
332 /**
333  ** @brief An internal Xmacro to generate const conserving string
334  ** to integer conversion function.
335  **
336  ** This macro just assembles _TG_STR_I_FUNC_() for the two case
337  ** of @C being @c char or @c wchar_t.
338  **
339  ** @param T is the target type of the conversion
340  ** @param NAME is the base name for the function
341  **/
342 #define _TG_STR_I_FUNC(T, NAME)                  \
343   _TG_STR_I_FUNC_(T, char, str ## NAME);         \
344   _TG_STR_I_FUNC_(T, wchar_t, wcs ## NAME)
345
346
347 /* generate the six groups of integer conversion functions */
348 _TG_STR_I_FUNC(long, tol);
349 _TG_STR_I_FUNC(long long, toll);
350 _TG_STR_I_FUNC(unsigned long, toul);
351 _TG_STR_I_FUNC(unsigned long long, toull);
352 _TG_STR_I_FUNC(intmax_t, toimax);
353 _TG_STR_I_FUNC(uintmax_t, toumax);
354
355 /**
356  ** @brief An internal macro to select a const conserving string
357  ** to integer conversion function.
358  **
359  ** This macro simply selects the function that corresponds to
360  ** the type of argument @a X. It is not meant to be used
361  ** directly but to be integrated in specific macros that
362  ** overload C library functions.
363  **
364  ** The argument @a X must correspond to an array or pointer,
365  ** otherwise an error is triggered in the controlling
```

```
 ** expression.
 **
 ** There is no default case, because these functions need a
 ** string and not some arbitrary bytes.
 **
 ** @see strtol on how to integrate this into a type generic user
 ** interface
 **/
# define _TG_STR_FUNC3(NAME, X, P, B, ...)              \
_Generic(&((*P)[0]),                                    \
         char*:          _TG_str ## NAME ## _nc,        \
         char const*:    _TG_str ## NAME ## _c,         \
         wchar_t*:       _TG_wcs ## NAME ## _nc,        \
         wchar_t const*: _TG_wcs ## NAME ## _c          \
         )((X), (P), (B))

/**
 ** @brief An internal macro to select a const conserving string
 ** to floating point conversion function.
 **
 ** @see _TG_STR_FUNC3 for a complete description of the strategy
 **/
# define _TG_STR_FUNC2(NAME, X, P, ...)                 \
_Generic(&((*P)[0]),                                    \
         char*:          _TG_str ## NAME ## _nc,        \
         char const*:    _TG_str ## NAME ## _c,         \
         wchar_t*:       _TG_wcs ## NAME ## _nc,        \
         wchar_t const*: _TG_wcs ## NAME ## _c          \
         )((X), (P))

# undef strtod
# define strtod(...) _TG_STR_FUNC2(tod, __VA_ARGS__, 0, 0, 0)
# undef strtof
# define strtof(...) _TG_STR_FUNC2(tof, __VA_ARGS__, 0, 0, 0)
# undef strtold
# define strtold(...) _TG_STR_FUNC2(told, __VA_ARGS__, 0, 0, 0)
# undef strtol
# define strtol(...) _TG_STR_FUNC3(tol, __VA_ARGS__, 0, 0, 0)
# undef strtoll
# define strtoll(...) _TG_STR_FUNC3(toll, __VA_ARGS__, 0, 0, 0)
# undef strtoul
# define strtoul(...) _TG_STR_FUNC3(toul, __VA_ARGS__, 0, 0, 0)
# undef strtoull
# define strtoull(...) _TG_STR_FUNC3(toull, __VA_ARGS__, 0, 0, 0)
# undef strtoimax
# define strtoimax(...) _TG_STR_FUNC3(toimax, __VA_ARGS__, 0, 0, 0)
# undef strtoumax
# define strtoumax(...) _TG_STR_FUNC3(toumax, __VA_ARGS__, 0, 0, 0)


#endif
```