Abstract

Hacker's guide to *quickselect*.

Design of quickselect

Bruce Lilly
bruce.lilly@gmail.com

November 3, 2017

1 Introduction

Quickselect, in the *libmedian* library efficiently implements multiple quickselect, *qsort*, C11 [1] *qsort_s*, and a similar variant of multiple quickselect *quickselect_s*. This paper will explain the design of the library code, its configuration and build process, and the location of the various source files which comprise the source code. The objective is to explain the rationale for the source code layout and to enable the reader to configure, build, use, understand, and possibly modify the library functions.

The C11 variants suffixed with *_s* exhibit two variations from standard *qsort* and *quickselect*:

1. Run-time bounds checks on several arguments, with constraint violation handling and error indication via return value (standard *qsort* has no return value) and via *errno* (this implementation of *qsort* and *quickselect* also sets *errno* for invalid arguments). The additional tests include the ability to detect huge values for the number of array elements *nmemb* or the element *size*, such as might result from underflow of unsigned integer arithmetic. However, there are some shortcomings in the defined error handling: the sole interface for obtaining the address of the error handling function (e.g. in order to be able to call the handler) is via *set_constraint_handler_s*, which changes the handler function (and returns a pointer to the previous handler). C11 does not indicate whether the handler is per-thread, per-process, or global, and there is no mechanism provided to ensure exclusive access. While a second call to *set_constraint_handler_s* could in theory return the handler to its previous value, it is also possible that another thread or process may have made a change to the handler in the meantime, causing a race condition and resulting in unpredictable behavior.

2. An additional *void ** argument to the sort function which is passed unchanged as an additional argument to the comparison function. This has extensive implications: all internal functions which use comparisons (pivot selection, partitioning, dedicated sorting and selection, etc.) are affected. This appears to be a supposed alternative to using application-specific comparison functions; it does not seem to justify essentially doubling the library code size.

These variants will not be further discussed in detail; note that there are two additional public entry points and that much of the internal code has variations in order to accommodate these variants.

2 Implementation

Referring to Figure 1:

- A caller provides a comparison function (green node labeled "compar") and calls either public interface: *qsort* (cyan node in library object file qsort.o) or *quickselect* (cyan node in library object quickselect.o). When calling *quickselect*, the caller may optionally supply a pointer to a swapping function ("swapf"); if NULL, a suitable function is chosen as described below for *qsort*. For some data types on some platforms with some compilers, a caller-supplied swap function may result in a run-time reduction of several percent. The caller of *quickselect* can specify processing options, for example to speficfy partial order stability.

- In the case of *quickselect*, the caller has also provided an array of *size_t* order statistic ranks and the number of ranks, also as a *size_t* argument. Internal operation of *quickselect* requires that the ranks be unique and sorted. A pass is made over the ranks, and if an adjacent pair is found not to be in non-decreasing order, the array of ranks is sorted (using the internal sorting interface, which is a simplification of selection, as will be explained later). Comparison of the *size_t* ranks while checking for

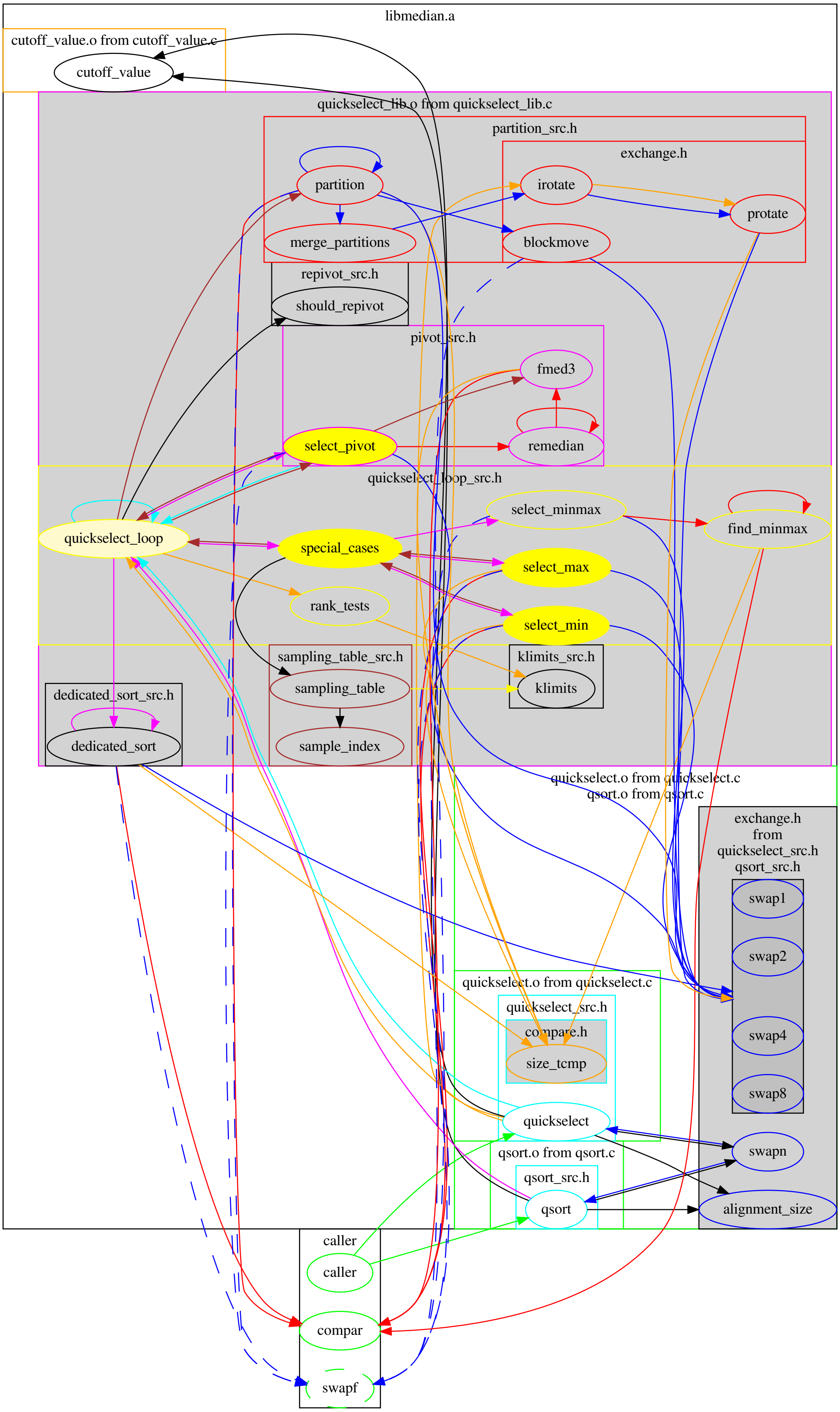

Figure 1: Library code implementation diagram

non-decreasing order is done directly; sorting uses a comparison function, *size_tcmp*, which is defined in header file compare.h. The comparison function *size_tcmp* is compiled as part of quickselect.o. The array of ranks is then checked for uniqueness; and duplicates are rotated to the end of the array. Only unique ranks are actually used for selection; the full set of ranks is returned to sorted order after selection. Rotation is implemented by element (*size_t*) swaps using an appropriate swapping function. The size and alignment of the *size_t* ranks array is determined using function *alignment_size*, which is defined in header file exchange.h and is compiled into quickselect.o (and also into qsort.o). The same function is used in both object files for determining the size and alignment of the array elements to be sorted or selected. The aligned size is passed to function *swapn*, which returns a pointer to a swap function which swaps in suitable increments of *char*; one of *swap1*, *swap2*, *swap4*, or *swap8*, all of which are also defined in exchange.h and are compiled into the two (four overall including *_s* variants) object files. A pointer to the swap function is passed to support functions which perform swaps, just as the pointer to the caller's comparison function is passed to support functions which perform comparisons.

- As is performed for the *quickselect size_t* ranks array, *quickselect* and *qsort* use *alignment_size* and *swapn* to determine a suitable swapping function for the *base* array elements (for *quickselect*, only if the caller did not supply a swapping function).

- The dedicated sorting function used for very small sub-arrays uses fewer comparisons and usually fewer swaps than divide-and-conquer quicksort. The optimum cutoff for switching between the dedicated sort and divide-and-conquer depends on the ratio of element size to the basic type size used for swapping (unless a caller-supplied swap function is used); it is also to some extent dependent on the relative CPU performance of comparisons and data transfers. The ratio is important because it affects the total run-time for swapping (but not for comparisons). That, in turn, affects the relative performance of the dedicated sort vs. divide-and-conquer sorting. Cutoff value is also dependent on processing options; because stable partitioning is expensive, dedicated sorting can be used for larger sub-array sizes. Both *qsort* and *quickselect* call function *cutoff_value* to determine a suitable cutoff value using tables based on the ratio of element size to alignment size. The tables (in file cutoff_table.c) can be optimized for a particular machine at compile-time, though the defaults should provide generally good performance (performance has a broad optimum with respect to cutoff size because the difference in the ratio of average comparisons to swaps for dedicated sort vs. divide-and-conquer is not too great over the range of sub-array sizes for which the dedicated sort is useful).

- The public functions *qsort* and *quickselect* call *quickselect_loop* in library object file quickselect_lib.o. Once processing has entered library code in quickselect_lib.o, it continues from there except for calls to comparison and swapping functions until the sorting or selection is complete.

- For sorting very small arrays, *quickselect_loop* calls *dedicated_sort*. The *dedicated_sort* function may call itself recursively; it completes sorting of the sub-array passed to it using the caller's comparison function.

- Selection of minimum, maximum, second-smallest, second-largest, or both smallest and largest array elements can be performed more efficiently than by divide-and-conquer. Function *special_cases* checks for those specific selection cases and calls the applicable functions to handle them. As a side-effect, *special_cases* also calls *sampling_table* to determine if selection can be more efficiently performed by sorting. In some cases, depending on the number of array elements and the number and distribution of order statistic ranks, sorting might be more efficient than divide-and-conquer selection. The distribution of desired order statistics is determined by analyzing the number of order statistics in three bands; function *klimits* indicates how many ranks are in each band.

- When the special-case selection does not apply, divide-and-conquer multiple selection (similar to quicksort) is used. The first step is to select a sample of the array; the number of samples is determined from a table based on the number of elements and on the distribution of desired order statistic ranks, and is returned from a call to *sampling_table*. That function also returns a pointer to one of three sampling tables; the index into the table is obtained by a call to *sample_index*, which is called by *sampling_table*. The relevant tables are in file tables.c.

- Having obtained an index into the appropriate sampling table, a pivot element is selected by calling function *select_pivot*. One of three pivot selection methods is used:

1. Remedian with base 3 of the sample of elements using internal function *remedian* which is recursive and which also calls internal function *fmed3*. The ternary median of three elements is found by *fmed3* using a minimal number of calls to the comparison function.
2. Median-of-medians is used after a particularly lopsided partition; *select_pivot* repeatedly calls *fmed3* to obtain medians of sets of three elements. The medians are swapped (using the appropriate swapping function) to the first position of the set (minimizing the number of swaps using bias in the ternary median-of-3). The median of the medians (now located in the initial third of the array) is found by selection of the median using *quickselect_loop*.
3. Remedian with base 3 of a large subset of elements using internal function *remedian* which is recursive and which also calls internal function *fmed3*. The ternary median of three elements is found by *fmed3* using a minimal number of calls to the comparison function. This variant of remedian is used when repivoting is required and partial order stability is also specified. Median-of-medians cannot be used in that case because it rearranges elements, disrupting partial order stability.

- The pivot element and the range of elements comparing equal to the pivot and bracketing a partition around the pivot (only the pivot element itself unless median-of-medians was used) is passed to *partition* to partition the array elements around the pivot element. The *partition* function uses the bracketing information (if available) to avoid recomparisons. One of two partitioning methods is used:

  1. Non-stable split-end partitioning using Kiwiel's Algorithm L. Comparison and swapping functions are used, as is *blockmove* for moving a minimal number of elements to effectively reorder regions with similar ordering with respect to the pivot element.
  2. A stable in-place divide-and-conquer implementation which recursively partitions unpartitioned regions of the sub-array and then merges partitions using rotations.

- After partitioning, the sizes of the regions resulting from the partition are examined. For selection only, a region can be removed from further processing if there are no desired order statistic ranks in that region. Function *klimits* makes this determination using an efficient binary search through the (sorted and unique) order statistic ranks. The selection-specific code including the call to *klimits* is encapsulated in inline function *rank_tests*. For sorting or selection, a region with only one element is necessarily in its correct position, and need not be processed. Elements comparing equal to the pivot are also in position and not in need of further processing. For selection, a region containing no desired order statistic ranks requires no processing. The smaller of the two regions of elements not comparing equal to the pivot element, if it requires processing at all, is processed by a recursive call to *quickselect_loop* unless the large region requires no processing, in which case the small region is processed iteratively.

- If not selecting order statistics and the larger region's size is small enough to use the dedicated sort, function *dedicated_sort* is called to complete sorting. Otherwise, the size of the larger region is compared to the original number of elements to determine whether that large region should be repivoted by using median-of-medians for the next pivot selection instead of remedian of samples. Function *should_repivot* determines this from the region sizes, a table of repivoting factors, and a count maintained of the number of times the ratio of region sizes has exceeded a threshold. The large region is iteratively processed in *quickselect_loop*'s main loop. The repivot factor tables are in file tables.c, compiled into library object file tables.o.

3 Source code layout

Meeting a number of goals resulted in a somewhat unusual source code layout. These goals include:

- Support for the C11 variants without repetition of common code. This is partially achieved by macros which provide for the variation of function return type and arguments.

- Modularity of source code; separate functions in separate files, avoiding huge files.

- The ability to build for various tradeoffs between object code size and run-time performance. Use of related inline functions linked together provides fast execution but large object file size. Conversely, individual functions can be called as needed to avoid multiple copies of code, but function call overhead limits run-time performance. The layout described in the previous section tends to favor run-time

function	declaration	source	inclusion	object
alignment_size	exchange.h	exchange.h	qsort_src.h	qsort.o qsort_s.o
swapn	exchange.h	exchange.h	qsort_src.h	qsort.o qsort_s.o
swap1	exchange.h	exchange.h	qsort_src.h	qsort.o qsort_s.o
swap2	exchange.h	exchange.h	qsort_src.h	qsort.o qsort_s.o
swap4	exchange.h	exchange.h	qsort_src.h	qsort.o qsort_s.o
swap8	exchange.h	exchange.h	qsort_src.h	qsort.o qsort_s.o
qsort	quickselect.h (stdlib.h)	qsort_src.h	qsort.c	qsort.o
qsort_s	quickselect.h (stdlib.h)	qsort_src.h	qsort_s.c	qsort_s.o
alignment_size	exchange.h	exchange.h	quickselect_src.h	quickselect.o quickselect_s.o
swapn	exchange.h	exchange.h	quickselect_src.h	quickselect.o quickselect_s.o
swap1	exchange.h	exchange.h	quickselect_src.h	quickselect.o quickselect_s.o
swap2	exchange.h	exchange.h	quickselect_src.h	quickselect.o quickselect_s.o
swap4	exchange.h	exchange.h	quickselect_src.h	quickselect.o quickselect_s.o
swap8	exchange.h	exchange.h	quickselect_src.h	quickselect.o quickselect_s.o
quickselect	quickselect.h	quickselect_src.h	quickselect.c	quickselect.o
quickselect_s	quickselect.h	quickselect_src.h	quickselect_s.c	quickselect_s.o
size_tcmp	compare.h	compare.h	quickselect_src.h	quickselect.o quickselect_s.o
irotate	exchange.h	exchange.h	quickselect_src.h	- (inline)
irotate	exchange.h	exchange.h	partition_src.h	- (inline)
protate	exchange.h	exchange.h	partition_src.h	quickselect_lib.o quickselect_lib_s.o
dedicated_sort	dedicated_sort_decl.h	dedicated_sort_src.h	quickselect_lib.c	quickselect_lib.o
quickselect_loop	quickselect_loop_decl.h	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o
special_cases	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o quickselect_lib_s.o
select_max	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o
select_min	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o
select_minmax	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o
find_minmax	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o
sampling_table	sampling_table_decl.h	sampling_table_src.h	quickselect_lib.c	quickselect_lib.o quickselect_lib_s.o
sample_index	- (static)	sampling_table_src.h	quickselect_lib.c	quickselect_lib.o quickselect_lib_s.o
select_pivot	select_pivot_decl.h	pivot_src.h	quickselect_lib.c	quickselect_lib.o
remedian	- (static)	pivot_src.h	quickselect_lib.c	quickselect_lib.o
fmed3	- (static)	pivot_src.h	quickselect_lib.c	quickselect_lib.o
partition	partition_decl.h	partition_src.h	quickselect_lib.c	quickselect_lib.o
merge_partitions	- (static)	partition_src.h	quickselect_lib.c	quickselect_lib.o
should_repivot	should_repivot_decl.h	repivot_src.h	quickselect_lib.c	quickselect_lib.o quickselect_lib_s.o
rank_tests	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o quickselect_lib_s.o
klimits	klimits_decl.h	klimits_src.h	quickselect_lib.c	quickselect_lib.o quickselect_lib_s.o
blockmove	exchange.h	exchange.h	partition_src.h	quickselect_lib.o quickselect_lib_s.o
dedicated_sort_s	dedicated_sort_decl.h	dedicated_sort_src.h	quickselect_lib_s.c	quickselect_lib_s.o
quickselect_loop_s	quickselect_loop_decl.h	quickselect_loop_src.h	quickselect_lib_s.c	quickselect_lib_s.o
quicksort_loop_s	quicksort_loop_decl.h	quicksort_loop_src.h	quickselect_lib_s.c	quickselect_lib_s.o
select_max_s	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib_s.o
select_min_s	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib_s.o
select_minmax_s	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib_s.o
find_minmax_s	- (static)	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib_s.o
sampling_table	sampling_table_decl.h	sampling_table_src.h	quickselect_lib_s.c	quickselect_lib_s.o
sample_index	- (static)	sampling_table_src.h	quickselect_lib_s.c	quickselect_lib_s.o
select_pivot_s	select_pivot_decl.h	pivot_src.h	quickselect_lib_s.c	quickselect_lib_s.o
remedian_s	- (static)	pivot_src.h	quickselect_lib_s.c	quickselect_lib_s.o
fmed3_s	- (static)	pivot_src.h	quickselect_lib_s.c	quickselect_lib_s.o
partition_s	partition_decl.h	partition_src.h	quickselect_lib_s.c	quickselect_lib_s.o
cutoff_value	cutoff_value_decl.h	cutoff_value.c	-	cutoff_value.o

Table 1: Source code layout

table	access functions	source	object
cutoff_table	cutoff_value	cutoff_table.c	cutoff_table.o
cutoff_table_c	cutoff_value	cutoff_table.c	cutoff_table.o
cutoff_table_s	cutoff_value	cutoff_table.c	cutoff_table.o
cutoff_table_sc	cutoff_value	cutoff_table.c	cutoff_table.o
sorting_sampling_table	sampling_table	tables.c	tables.o
ends_sampling_table	sampling_table	tables.c	tables.o
middle_sampling_table	sampling_table	tables.c	tables.o
selection_breakpoint	sampling_table	tables.c	tables.o
sorting_repivot_table_transparent	sampling_table	tables.c	tables.o
sorting_repivot_table_aggressive	sampling_table	tables.c	tables.o
sorting_repivot_table_relaxed	sampling_table	tables.c	tables.o
sorting_repivot_table_loose	sampling_table	tables.c	tables.o
sorting_repivot_table_disabled	sampling_table	tables.c	tables.o
selection_repivot_table	sampling_table	tables.c	tables.o

Table 2: Tables

performance, but it was desired to arrange the source code so that a library could be built for small object code size without entailing enormous effort.

- Avoiding errors due to inconsistencies, e.g. between function declarations and definitions.

The following general file layout was used to achieve those goals:

- Configuration parameters placed in a single header file, quickselect_config.h.
- Function definitions and declarations taken from the same source file, located as a header file containing the function name and parameter list. That header file can be #included and followed by a semicolon to act as a declaration (possibly prefixed with qualifiers such as *extern*). The same header file can be #included and followed by the function definition in curly braces for the function implementation. The same source ensures consistency at some small inconvenience (a separate file may need to be examined to see the parameter list).
- Being able to choose between separate object file creation or combined inline linkage is made possible by placing the source code in header files, which can the be #included separately (for separate object files) or in combination (for inline linkage) in a source file which consists predominantly of such #include directives.
- Use of feature test macros in combination with the inclusion of source code as header files permits building the C11 variants from a common code base, ensuring that bug fixes, performance improvements, feature enhancements, etc. are applied to all variants.
- Individual files can be kept to a reasonable size while still permitting combination into a single object file by inclusion of multiple source files (as header files) when building.

The source code file layout is detailed in Table 1 and can also be seen in Figure 1.

4 Data tables

Several aspects of program operation are controlled by tabular data:

- Cutoff values (number of array elements) for *dedicated_sort* are derived from a *cutoff_values* table based on the ratio of array element size to the size used for swapping (which is itself based on the array element size and on the alignment of the array elements).
- Breakpoints for increasing the number of samples used for pivot selection are maintained in 3 tables: one for sorting and two for selection, depending on the distribution of the desired order statistic ranks.
- Tables of thresholds compared to the ratio of the number of elements in the large region resulting from a partition to the number of remaining elements from the original sub-array which was partitioned are used to determine when to use median-of-medians for pivot selection. One table is used for selection, and of of several possible tables (chosen at compile-time) is used for sorting depending on the desired tradeoff between performance for adverse and non-adverse input sequences.

- A table is used to detrmine when sorting is likely to be more efficient than selection for small sub-arrays, based on the number and distribution of desired order statistics and the number of sub-array elements.

5 Building details

The order of object files in a library archive is important. Sorting of disordered order statistic ranks uses standard sorting via the internal *quickselect_loop* function (no extra “context” is necessary or desirable, and arguments once checked need not be checked again when passing unchanged to another function). Consequently, the object files for standard sorting should appear after the C11 variants, as those variants may need to call the standard functions. Object files containing the data tables appear last; the tables are used by all variants.

References

[1] ISO/IEC JTC1/SC22/WG14 - C. ISO/IEC9899:201x. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, April, 2011. [23 September 2017].