NAME

DEFINE_STACK_OF, DEFINE_STACK_OF_CONST, DEFINE_SPECIAL_STACK_OF, DEFINE_SPECIAL_STACK_OF_CONST, sk_TYPE_num, sk_TYPE_value, sk_TYPE_new, sk_TYPE_new_null, sk_TYPE_reserve, sk_TYPE_free, sk_TYPE_zero, sk_TYPE_delete, sk_TYPE_delete_ptr, sk_TYPE_push, sk_TYPE_unshift, sk_TYPE_pop, sk_TYPE_shift, sk_TYPE_pop_free, sk_TYPE_insert, sk_TYPE_set, sk_TYPE_find, sk_TYPE_find_ex, sk_TYPE_find_all, sk_TYPE_sort, sk_TYPE_is_sorted, sk_TYPE_dup, sk_TYPE_deep_copy, sk_TYPE_set_cmp_func, sk_TYPE_new_reserve, OPENSSL_sk_deep_copy, OPENSSL_sk_delete, OPENSSL_sk_delete_ptr, OPENSSL_sk_dup, OPENSSL_sk_find, OPENSSL_sk_find_ex, OPENSSL_sk_find_all, OPENSSL_sk_free, OPENSSL_sk_insert, OPENSSL_sk_is_sorted, OPENSSL_sk_new, OPENSSL_sk_new_null, OPENSSL_sk_new_reserve, OPENSSL_sk_num, OPENSSL_sk_pop, OPENSSL_sk_pop_free, OPENSSL_sk_push, OPENSSL_sk_reserve, OPENSSL_sk_set, OPENSSL_sk_set_cmp_func, OPENSSL_sk_shift, OPENSSL_sk_sort, OPENSSL_sk_unshift, OPENSSL_sk_value, OPENSSL_sk_zero - stack container

SYNOPSIS

```
#include <openssl/safestack.h>

STACK_OF(TYPE)
DEFINE_STACK_OF(TYPE)
DEFINE_STACK_OF_CONST(TYPE)
DEFINE_SPECIAL_STACK_OF(FUNCTYPE, TYPE)
DEFINE_SPECIAL_STACK_OF_CONST(FUNCTYPE, TYPE)

typedef int (*sk_TYPE_compfunc)(const TYPE *const *a, const TYPE *const *b);
typedef TYPE * (*sk_TYPE_copyfunc)(const TYPE *a);
typedef void (*sk_TYPE_freefunc)(TYPE *a);

int sk_TYPE_num(const STACK_OF(TYPE) *sk);
TYPE *sk_TYPE_value(const STACK_OF(TYPE) *sk, int idx);
STACK_OF(TYPE) *sk_TYPE_new(sk_TYPE_compfunc compare);
STACK_OF(TYPE) *sk_TYPE_new_null(void);
int sk_TYPE_reserve(STACK_OF(TYPE) *sk, int n);
void sk_TYPE_free(const STACK_OF(TYPE) *sk);
void sk_TYPE_zero(const STACK_OF(TYPE) *sk);
TYPE *sk_TYPE_delete(STACK_OF(TYPE) *sk, int i);
TYPE *sk_TYPE_delete_ptr(STACK_OF(TYPE) *sk, TYPE *ptr);
int sk_TYPE_push(STACK_OF(TYPE) *sk, const TYPE *ptr);
int sk_TYPE_unshift(STACK_OF(TYPE) *sk, const TYPE *ptr);
TYPE *sk_TYPE_pop(STACK_OF(TYPE) *sk);
```

```
TYPE *sk_TYPE_shift(STACK_OF(TYPE) *sk);
void sk_TYPE_pop_free(STACK_OF(TYPE) *sk, sk_TYPE_freefunc freefunc);
int sk_TYPE_insert(STACK_OF(TYPE) *sk, TYPE *ptr, int idx);
TYPE *sk_TYPE_set(STACK_OF(TYPE) *sk, int idx, const TYPE *ptr);
int sk_TYPE_find(STACK_OF(TYPE) *sk, TYPE *ptr);
int sk_TYPE_find_ex(STACK_OF(TYPE) *sk, TYPE *ptr);
int sk_TYPE_find_all(STACK_OF(TYPE) *sk, TYPE *ptr, int *pnum);
void sk_TYPE_sort(const STACK_OF(TYPE) *sk);
int sk_TYPE_is_sorted(const STACK_OF(TYPE) *sk);
STACK_OF(TYPE) *sk_TYPE_dup(const STACK_OF(TYPE) *sk);
STACK_OF(TYPE) *sk_TYPE_deep_copy(const STACK_OF(TYPE) *sk,
                                  sk_TYPE_copyfunc copyfunc,
                                  sk_TYPE_freefunc freefunc);
sk_TYPE_compfunc (*sk_TYPE_set_cmp_func(STACK_OF(TYPE) *sk,
                                        sk_TYPE_compfunc compare));
STACK_OF(TYPE) *sk_TYPE_new_reserve(sk_TYPE_compfunc compare, int n);
```

DESCRIPTION

Applications can create and use their own stacks by placing any of the macros described below in a header file. These macros define typesafe inline functions that wrap around the utility **OPENSSL_sk_** API. In the description here, ***TYPE*** is used as a placeholder for any of the OpenSSL datatypes, such as **X509**.

The **STACK_OF()** macro returns the name for a stack of the specified ***TYPE***. This is an opaque pointer to a structure declaration. This can be used in every header file that references the stack. There are several **DEFINE...** macros that create static inline functions for all of the functions described on this page. This should normally be used in one source file, and the stack manipulation is wrapped with application-specific functions.

DEFINE_STACK_OF() creates set of functions for a stack of ***TYPE*** elements. The type is referenced by **STACK_OF**(***TYPE***) and each function name begins with **sk_*TYPE*_**.
DEFINE_STACK_OF_CONST() is identical to **DEFINE_STACK_OF()** except each element is constant.

```
/* DEFINE_STACK_OF(TYPE) */
TYPE *sk_TYPE_value(STACK_OF(TYPE) *sk, int idx);
/* DEFINE_STACK_OF_CONST(TYPE) */
const TYPE *sk_TYPE_value(STACK_OF(TYPE) *sk, int idx);
```

DEFINE_SPECIAL_STACK_OF() and **DEFINE_SPECIAL_STACK_OF_CONST()** are similar

except **FUNCNAME** is used in the function names:

```
/* DEFINE_SPECIAL_STACK_OF(TYPE, FUNCNAME) */
TYPE *sk_FUNCNAME_value(STACK_OF(TYPE) *sk, int idx);
/* DEFINE_SPECIAL_STACK_OF(TYPE, FUNCNAME) */
const TYPE *sk_FUNCNAME_value(STACK_OF(TYPE) *sk, int idx);
```

sk_*TYPE*_num() returns the number of elements in *sk* or -1 if *sk* is NULL.

sk_*TYPE*_value() returns element *idx* in *sk*, where *idx* starts at zero. If *idx* is out of range then NULL is returned.

sk_*TYPE*_new() allocates a new empty stack using comparison function *compare*. If *compare* is NULL then no comparison function is used. This function is equivalent to **sk_*TYPE*_new_reserve**(*compare*, 0).

sk_*TYPE*_new_null() allocates a new empty stack with no comparison function. This function is equivalent to **sk_*TYPE*_new_reserve**(NULL, 0).

sk_*TYPE*_reserve() allocates additional memory in the *sk* structure such that the next *n* calls to **sk_*TYPE*_insert**(), **sk_*TYPE*_push**() or **sk_*TYPE*_unshift**() will not fail or cause memory to be allocated or reallocated. If *n* is zero, any excess space allocated in the *sk* structure is freed. On error *sk* is unchanged.

sk_*TYPE*_new_reserve() allocates a new stack. The new stack will have additional memory allocated to hold *n* elements if *n* is positive. The next *n* calls to **sk_*TYPE*_insert**(), **sk_*TYPE*_push**() or **sk_*TYPE*_unshift**() will not fail or cause memory to be allocated or reallocated. If *n* is zero or less than zero, no memory is allocated. **sk_*TYPE*_new_reserve**() also sets the comparison function *compare* to the newly created stack. If *compare* is NULL then no comparison function is used.

sk_*TYPE*_set_cmp_func() sets the comparison function of *sk* to *compare*. The previous comparison function is returned or NULL if there was no previous comparison function.

sk_*TYPE*_free() frees up the *sk* structure. It does *not* free up any elements of *sk*. After this call *sk* is no longer valid.

sk_*TYPE*_zero() sets the number of elements in *sk* to zero. It does not free *sk* so after this call *sk* is still valid.

sk_*TYPE*_pop_free() frees up all elements of *sk* and *sk* itself. The free function **freefunc()** is called on

each element to free it.

sk_*TYPE*_delete() deletes element *i* from *sk*. It returns the deleted element or NULL if *i* is out of range.

sk_*TYPE*_delete_ptr() deletes element matching *ptr* from *sk*. It returns the deleted element or NULL if no element matching *ptr* was found.

sk_*TYPE*_insert() inserts *ptr* into *sk* at position *idx*. Any existing elements at or after *idx* are moved downwards. If *idx* is out of range the new element is appended to *sk*. **sk_*TYPE*_insert**() either returns the number of elements in *sk* after the new element is inserted or zero if an error (such as memory allocation failure) occurred.

sk_*TYPE*_push() appends *ptr* to *sk* it is equivalent to:

```
sk_TYPE_insert(sk, ptr, -1);
```

sk_*TYPE*_unshift() inserts *ptr* at the start of *sk* it is equivalent to:

```
sk_TYPE_insert(sk, ptr, 0);
```

sk_*TYPE*_pop() returns and removes the last element from *sk*.

sk_*TYPE*_shift() returns and removes the first element from *sk*.

sk_*TYPE*_set() sets element *idx* of *sk* to *ptr* replacing the current element. The new element value is returned or NULL if an error occurred: this will only happen if *sk* is NULL or *idx* is out of range.

sk_*TYPE*_find() searches *sk* for the element *ptr*. In the case where no comparison function has been specified, the function performs a linear search for a pointer equal to *ptr*. The index of the first matching element is returned or **-1** if there is no match. In the case where a comparison function has been specified, *sk* is sorted and **sk_*TYPE*_find**() returns the index of a matching element or **-1** if there is no match. Note that, in this case the comparison function will usually compare the values pointed to rather than the pointers themselves and the order of elements in *sk* can change. Note that because the stack may be sorted as the result of a **sk_*TYPE*_find**() call, if a lock is being used to synchronise access to the stack across multiple threads, then that lock must be a "write" lock.

sk_*TYPE*_find_ex() operates like **sk_*TYPE*_find**() except when a comparison function has been specified and no matching element is found. Instead of returning **-1**, **sk_*TYPE*_find_ex**() returns the index of the element either before or after the location where *ptr* would be if it were present in *sk*. The

function also does not guarantee that the first matching element in the sorted stack is returned.

sk_*TYPE*_find_all() operates like **sk_*TYPE*_find**() but it also sets the **pnum* to number of matching elements in the stack. In case no comparison function has been specified the **pnum* will be always set to 1 if matching element was found, 0 otherwise.

sk_*TYPE*_sort() sorts *sk* using the supplied comparison function.

sk_*TYPE*_is_sorted() returns **1** if *sk* is sorted and **0** otherwise.

sk_*TYPE*_dup() returns a shallow copy of *sk* or an empty stack if the passed stack is NULL. Note the pointers in the copy are identical to the original.

sk_*TYPE*_deep_copy() returns a new stack where each element has been copied or an empty stack if the passed stack is NULL. Copying is performed by the supplied **copyfunc()** and freeing by **freefunc()**. The function **freefunc()** is only called if an error occurs.

NOTES

Care should be taken when accessing stacks in multi-threaded environments. Any operation which increases the size of a stack such as **sk_*TYPE*_insert**() or **sk_*TYPE*_push**() can "grow" the size of an internal array and cause race conditions if the same stack is accessed in a different thread. Operations such as **sk_*TYPE*_find**() and **sk_*TYPE*_sort**() can also reorder the stack.

Any comparison function supplied should use a metric suitable for use in a binary search operation. That is it should return zero, a positive or negative value if *a* is equal to, greater than or less than *b* respectively.

Care should be taken when checking the return values of the functions **sk_*TYPE*_find**() and **sk_*TYPE*_find_ex**(). They return an index to the matching element. In particular **0** indicates a matching first element. A failed search is indicated by a **-1** return value.

STACK_OF(), **DEFINE_STACK_OF()**, **DEFINE_STACK_OF_CONST()**, and **DEFINE_SPECIAL_STACK_OF()** are implemented as macros.

It is not an error to call **sk_*TYPE*_num**(), **sk_*TYPE*_value**(), **sk_*TYPE*_free**(), **sk_*TYPE*_zero**(), **sk_*TYPE*_pop_free**(), **sk_*TYPE*_delete**(), **sk_*TYPE*_delete_ptr**(), **sk_*TYPE*_pop**(), **sk_*TYPE*_shift**(), **sk_*TYPE*_find**(), **sk_*TYPE*_find_ex**(), and **sk_*TYPE*_find_all**() on a NULL stack, empty stack, or with an invalid index. An error is not raised in these conditions.

The underlying utility **OPENSSL_sk_** API should not be used directly. It defines these functions:

OPENSSL_sk_deep_copy(), **OPENSSL_sk_delete()**, **OPENSSL_sk_delete_ptr()**, **OPENSSL_sk_dup()**, **OPENSSL_sk_find()**, **OPENSSL_sk_find_ex()**, **OPENSSL_sk_find_all()**, **OPENSSL_sk_free()**, **OPENSSL_sk_insert()**, **OPENSSL_sk_is_sorted()**, **OPENSSL_sk_new()**, **OPENSSL_sk_new_null()**, **OPENSSL_sk_new_reserve()**, **OPENSSL_sk_num()**, **OPENSSL_sk_pop()**, **OPENSSL_sk_pop_free()**, **OPENSSL_sk_push()**, **OPENSSL_sk_reserve()**, **OPENSSL_sk_set()**, **OPENSSL_sk_set_cmp_func()**, **OPENSSL_sk_shift()**, **OPENSSL_sk_sort()**, **OPENSSL_sk_unshift()**, **OPENSSL_sk_value()**, **OPENSSL_sk_zero()**.

RETURN VALUES

sk_*TYPE*_num() returns the number of elements in the stack or **-1** if the passed stack is NULL.

sk_*TYPE*_value() returns a pointer to a stack element or NULL if the index is out of range.

sk_*TYPE*_new(), **sk_*TYPE*_new_null**() and **sk_*TYPE*_new_reserve**() return an empty stack or NULL if an error occurs.

sk_*TYPE*_reserve() returns **1** on successful allocation of the required memory or **0** on error.

sk_*TYPE*_set_cmp_func() returns the old comparison function or NULL if there was no old comparison function.

sk_*TYPE*_free(), **sk_*TYPE*_zero**(), **sk_*TYPE*_pop_free**() and **sk_*TYPE*_sort**() do not return values.

sk_*TYPE*_pop(), **sk_*TYPE*_shift**(), **sk_*TYPE*_delete**() and **sk_*TYPE*_delete_ptr**() return a pointer to the deleted element or NULL on error.

sk_*TYPE*_insert(), **sk_*TYPE*_push**() and **sk_*TYPE*_unshift**() return the total number of elements in the stack and 0 if an error occurred. **sk_*TYPE*_push**() further returns -1 if *sk* is NULL.

sk_*TYPE*_set() returns a pointer to the replacement element or NULL on error.

sk_*TYPE*_find() and **sk_*TYPE*_find_ex**() return an index to the found element or **-1** on error.

sk_*TYPE*_is_sorted() returns **1** if the stack is sorted and **0** if it is not.

sk_*TYPE*_dup() and **sk_*TYPE*_deep_copy**() return a pointer to the copy of the stack or NULL on error.

HISTORY

Before OpenSSL 1.1.0, this was implemented via macros and not inline functions and was not a public

API.

sk_*TYPE*_reserve() and **sk_*TYPE*_new_reserve**() were added in OpenSSL 1.1.1.

COPYRIGHT