Simple Dynamic Strings

Notes about verison 2: this is an updated version of SDS in an attempt to finally unify Redis, Disque, Hiredis, and
the stand alone SDS versions. This version is *NOT binary compatible** with SDS verison 1, but the API is 99%
compatible so switching to the new lib should be trivial.

Note that this version of SDS may be a slower with certain workloads, but uses less memory compared to V1 since header
size is dynamic and depends to the string to alloc.

Moreover it includes a few more API functions, notably `sdscatfmt` which is a faster version of `sdscatprintf` that can
be used for the simpler cases in order to avoid the libc `printf` family functions performance penalty.

How SDS stirngs work

SDS is a string library for C designed to augment the limited libc string handling functionalities by adding heap
allocated strings that are:

- Simpler to use.
- Binary safe.
- Computationally more efficient.
- But yet... Compatible with normal C string functions.

This is achieved using an alternative design in which instead of using a C structure to represent a string, we use a
binary prefix that is stored before the actual pointer to the string that is returned by SDS to the user.

```
+--------+-------------------------------+-----------+
| Header | Binary safe C alike string... | Null term |
+--------+-------------------------------+-----------+
         |
         `-> Pointer returned to the user.
```

Because of meta data stored before the actual returned pointer as a prefix, and because of every SDS string implicitly
adding a null term at the end of the string regardless of the actual content of the string, SDS strings work well
together with C strings and the user is free to use them interchangeably with real-only functions that access the string
in read-only.

SDS was a C string I developed in the past for my everyday C programming needs, later it was moved into Redis where it
is used extensively and where it was modified in order to be suitable for high performance

operations. Now it was
extracted from Redis and forked as a stand alone project.

Because of its many years life inside Redis, SDS provides both higher level functions for easy strings manipulation in
C, but also a set of low level functions that make it possible to write high performance code without paying a penalty
for using an higher level string library.

Advantages and disadvantages of SDS

Normally dynamic string libraries for C are implemented using a structure that defines the string. The structure has a
pointer field that is managed by the string function, so it looks like this:

```
struct yourAverageStringLibrary {
    char *buf;
    size_t len;
    ... possibly more fields here ...
};
```

SDS strings are already mentioned don't follow this schema, and are instead a single allocation with a prefix that
lives *before* the address actually returned for the string.

There are advantages and disadvantages with this approach over the traditional approach:

Disadvantage #1: many functions return the new string as value, since sometimes SDS requires to create a new string
with more space, so the most SDS API calls look like this:

```
s = sdscat(s,"Some more data");
```

As you can see s is used as input for `sdscat` but is also set to the value returned by the SDS API call, since we are
not sure if the call modified the SDS string we passed or allocated a new one. Not remembering to assign back the return
value of `sdscat` or similar functions to the variable holding the SDS string will result in a bug.

Disadvantage #2: if an SDS string is shared in different places in your program you have to modify all the
references when you modify the string. However most of the times when you need to share SDS strings it is much better to
encapsulate them into structures with a `reference count` otherwise it is too easy to incur into memory leaks.

Advantage #1: you can pass SDS strings to functions designed for C functions without accessing a struct member or
calling a function, like this:

```
printf("%s\n", sds_string);
```

In most other libraries this will be something like:

```
printf("%s\n", string->buf);
```

Or:

```
printf("%s\n", getStringPointer(string));
```

Advantage #2: accessing individual chars is straightforward. C is a low level language so this is an important
operation in many programs. With SDS strings accessing individual chars is very natural:

```
printf("%c %c\n", s[0], s[1]);
```

With other libraries your best chance is to assign `string->buf` (or call the function to get the string pointer) to
a `char` pointer and work with this. However since the other libraries may reallocate the buffer implicitly every time
you call a function that may modify the string you have to get a reference to the buffer again.

Advantage #3: single allocation has better cache locality. Usually when you access a string created by a string
library using a structure, you have two different allocations for the structure representing the string, and the actual
buffer holding the string. Over the time the buffer is reallocated, and it is likely that it ends in a totally different
part of memory compared to the structure itself. Since modern programs performances are often dominated by cache misses,
SDS may perform better in many workloads.

SDS basics

The type of SDS strings is just the char pointer `char *`. However SDS defines an `sds` type as alias of `char *` in its
header file: you should use the
`sds` type in order to make sure you remember that a given variable in your program holds an SDS string and not a C
string, however this is not mandatory.

This is the simplest SDS program you can write that does something:

```
sds mystring = sdsnew("Hello World!");
printf("%s\n", mystring);
sdsfree(mystring);

output> Hello World!
```

The above small program already shows a few important things about SDS:

- SDS strings are created, and heap allocated, via the `sdsnew()` function, or other similar functions that we'll see in
  a moment.
- SDS strings can be passed to `printf()` like any other C string.
- SDS strings require to be freed with `sdsfree()`, since they are heap allocated.

Creating SDS strings

```
sds sdsnewlen(const void *init, size_t initlen);
sds sdsnew(const char *init);
sds sdsempty(void);
sds sdsdup(const sds s);
```

There are many ways to create SDS strings:

- The `sdsnew` function creates an SDS string starting from a C null terminated string. We already saw how it works in
  the above example.

- The `sdsnewlen` function is similar to `sdsnew` but instead of creating the string assuming that the input string is
  null terminated, it gets an additional length parameter. This way you can create a string using binary data:

  ```
  char buf[3];
  sds mystring;

  buf[0] = 'A';
  buf[1] = 'B';
  buf[2] = 'C';
  mystring = sdsnewlen(buf,3);
  printf("%s of len %d\n", mystring, (int) sdslen(mystring));

  output> ABC of len 3
  ```

  Note: `sdslen` return value is casted to `int` because it returns a `size_t`
  type. You can use the right `printf` specifier instead of casting.

- The `sdsempty()` function creates an empty zero-length string:

  ```
  sds mystring = sdsempty();
  printf("%d\n", (int) sdslen(mystring));

  output> 0
  ```

- The `sdsdup()` function duplicates an already existing SDS string:

  ```
  sds s1, s2;

  s1 = sdsnew("Hello");
  s2 = sdsdup(s1);
  printf("%s %s\n", s1, s2);

  output> Hello Hello
  ```

Obtaining the string length

```
size_t sdslen(const sds s);
```

In the examples above we already used the `sdslen` function in order to get the length of the string. This function
works like `strlen` of the libc except that:

- It runs in constant time since the length is stored in the prefix of SDS strings, so calling `sdslen` is not expensive
  even when called with very large strings.
- The function is binary safe like any other SDS string function, so the length is the true length of the string
  regardless of the content, there is no problem if the string includes null term characters in the middle.

As an example of the binary safeness of SDS strings, we can run the following code:

```
sds s = sdsnewlen("A\0\0B",4);
printf("%d\n", (int) sdslen(s));

output> 4
```

Note that SDS strings are always null terminated at the end, so even in that case `s[4]` will be a null term, however
printing the string with `printf`
would result in just `"A"` to be printed since libc will treat the SDS string like a normal C string.

Destroying strings

```
void sdsfree(sds s);
```

The destroy an SDS string there is just to call `sdsfree` with the string pointer. However note that empty strings
created with `sdsempty` need to be destroyed as well otherwise they'll result into a memory leak.

The function `sdsfree` does not perform any operation if instead of an SDS string pointer, `NULL` is passed, so you
don't need to check for `NULL` explicitly before calling it:

```
if (string) sdsfree(string); /* Not needed. */
sdsfree(string); /* Same effect but simpler. */
```

Concatenating strings

Concatenating strings to other strings is likely the operation you will end using the most with a dynamic C string
library. SDS provides different functions to concatenate strings to existing strings.

```
sds sdscatlen(sds s, const void *t, size_t len);
sds sdscat(sds s, const char *t);
```

The main string concatenation functions are `sdscatlen` and `sdscat` that are identical, the only difference being
that `sdscat` does not have an explicit length argument since it expects a null terminated string.

```
sds s = sdsempty();
s = sdscat(s, "Hello ");
s = sdscat(s, "World!");
printf("%s\n", s);

output> Hello World!
```

Sometimes you want to cat an SDS string to another SDS string, so you don't need to specify the length, but at the same
time the string does not need to be null terminated but can contain any binary data. For this there is a special
function:

```
sds sdscatsds(sds s, const sds t);
```

Usage is straightforward:

```
sds s1 = sdsnew("aaa");
sds s2 = sdsnew("bbb");
s1 = sdscatsds(s1,s2);
sdsfree(s2);
printf("%s\n", s1);

output> aaabbb
```

Sometimes you don't want to append any special data to the string, but you want to make sure that there are at least a
given number of bytes composing the whole string.

```
sds sdsgrowzero(sds s, size_t len);
```

The `sdsgrowzero` function will do nothing if the current string length is already `len` bytes, otherwise it will
enlarge the string to `len` just padding it with zero bytes.

```
sds s = sdsnew("Hello");
s = sdsgrowzero(s,6);
s[5] = '!'; /* We are sure this is safe because of sdsgrowzero() */
printf("%s\n', s);

output> Hello!
```

Formatting strings

There is a special string concatenation function that accepts a `printf` alike format specifier and cats the formatted
string to the specified string.

```
sds sdscatprintf(sds s, const char *fmt, ...) {
```

Example:

```
sds s;
int a = 10, b = 20;
s = sdsnew("The sum is: ");
s = sdscatprintf(s,"%d+%d = %d",a,b,a+b);
```

Often you need to create SDS string directly from `printf` format specifiers. Because `sdscatprintf` is actually a
function that concatenates strings all you need is to concatenate your string to an empty string:

```
char *name = "Anna";
int loc = 2500;
sds s;
s = sdscatprintf(sdsempty(), "%s wrote %d lines of LISP\n", name, loc);
```

You can use `sdscatprintf` in order to convert numbers into SDS strings:

```
int some_integer = 100;
sds num = sdscatprintf(sdsempty(),"%d\n", some_integer);
```

However this is slow and we have a special function to make it efficient.

Fast number to string operations

Creating an SDS string from an integer may be a common operation in certain kind of programs, and while you may do this
with `sdscatprintf` the performance hit is big, so SDS provides a specialized function.

```
sds sdsfromlonglong(long long value);
```

Use it like this:

```
sds s = sdsfromlonglong(10000);
printf("%d\n", (int) sdslen(s));

output> 5
```

Trimming strings and getting ranges

String trimming is a common operation where a set of characters are removed from the left and the right of the string.
Another useful operation regarding strings is the ability to just take a range out of a larger string.

```
void sdstrim(sds s, const char *cset);
void sdsrange(sds s, int start, int end);
```

SDS provides both the operations with the `sdstrim` and `sdsrange` functions. However note that both functions work
differently than most functions modifying SDS strings since the return value is null: basically those functions always

destructively modify the passed SDS string, never allocating a new one, because both trimming and ranges will never need
more room: the operations can only remove characters from the original strings.

Because of this behavior, both functions are fast and don't involve reallocation.

This is an example of string trimming where newlines and spaces are removed from an SDS strings:

```
sds s = sdsnew("         my string\n\n  ");
sdstrim(s," \n");
printf("-%s-\n",s);

output> -my string-
```

Basically `sdstrim` takes the SDS string to trim as first argument, and a null terminated set of characters to remove
from left and right of the string. The characters are removed as long as they are not interrupted by a character that is
not in the list of characters to trim: this is why the space between
`"my"` and `"string"` was preserved in the above example.

Taking ranges is similar, but instead to take a set of characters, it takes to indexes, representing the start and the
end as specified by zero-based indexes inside the string, to obtain the range that will be retained.

```
sds s = sdsnew("Hello World!");
sdsrange(s,1,4);
printf("-%s-\n");

output> -ello-
```

Indexes can be negative to specify a position starting from the end of the string, so that `-1` means the last
character, `-2` the penultimate, and so forth:

```
sds s = sdsnew("Hello World!");
sdsrange(s,6,-1);
printf("-%s-\n");
sdsrange(s,0,-2);
printf("-%s-\n");

output> -World!-
output> -World-
```

`sdsrange` is very useful when implementing networking servers processing a protocol or sending messages. For example
the following code is used implementing the write handler of the Redis Cluster message bus between nodes:

```
void clusterWriteHandler(..., int fd, void *privdata, ...) {
    clusterLink *link = (clusterLink*) privdata;
    ssize_t nwritten = write(fd, link->sndbuf, sdslen(link->sndbuf));
    if (nwritten <= 0) {
        /* Error handling... */
    }
    sdsrange(link->sndbuf,nwritten,-1);
    ... more code here ...
}
```

Every time the socket of the node we want to send the message to is writable we attempt to write as much bytes as
possible, and we use `sdsrange` in order to remove from the buffer what was already sent.

The function to queue new messages to send to some node in the cluster will simply use `sdscatlen` in order to put more
data in the send buffer.

Note that the Redis Cluster bus implements a binary protocol, but since SDS is binary safe this is not a problem, so the
goal of SDS is not just to provide an high level string API for the C programmer but also dynamically allocated buffers
that are easy to manage.

String copying

The most dangerous and infamus function of the standard C library is probably
`strcpy`, so perhaps it is funny how in the context of better designed dynamic string libraries the concept of copying
strings is almost irrelevant. Usually what you do is to create strings with the content you want, or concatenating more
content as needed.

However SDS features a string copy function that is useful in performance critical code sections, however I guess its
practical usefulness is limited as the function never managed to get called in the context of the 50k lines of code
composing the Redis code base.

```
sds sdscpylen(sds s, const char *t, size_t len);
sds sdscpy(sds s, const char *t);
```

The string copy function of SDS is called `sdscpylen` and works like that:

```
s = sdsnew("Hello World!");
s = sdscpylen(s,"Hello Superman!",15);
```

As you can see the function receives as input the SDS string `s`, but also returns an SDS string. This is common to many
SDS functions that modify the string: this way the returned SDS string may be the original one modified or a newly
allocated one (for example if there was not enough room in the old SDS string).

The `sdscpylen` will simply replace what was in the old SDS string with the new data you pass using the pointer and
length argument. There is a similar function called `sdscpy` that does not need a length but expects a null terminated
string instead.

You may wonder why it makes sense to have a string copy function in the SDS library, since you can simply create a new
SDS string from scratch with the new value instead of copying the value in an existing SDS string. The reason is
efficiency: `sdsnewlen` will always allocate a new string while `sdscpylen` will try to reuse the existing string if
there is enough room to old the new content specified by the user, and will allocate a new one only if needed.

Quoting strings

In order to provide consistent output to the program user, or for debugging purposes, it is often important to turn a
string that may contain binary data or special characters into a quoted string. Here for quoted string we mean the
common format for String literals in programming source code. However today this format is also part of the well known
serialization formats like JSON and CSV, so it definitely escaped the simple gaol of representing literals strings in
the source code of programs.

An example of quoted string literal is the following:

```
"\x00Hello World\n"
```

The first byte is a zero byte while the last byte is a newline, so there are two non alphanumerical characters inside
the string.

SDS uses a concatenation function for this goal, that concatenates to an existing string the quoted string
representation of the input string.

```
sds sdscatrepr(sds s, const char *p, size_t len);
```

The `scscatrepr` (where `repr` means *representation*) follows the usualy SDS string function rules accepting a char
pointer and a length, so you can use it with SDS strings, normal C strings by using strlen() as `len` argument, or
binary data. The following is an example usage:

```
sds s1 = sdsnew("abcd");
sds s2 = sdsempty();
s[1] = 1;
s[2] = 2;
s[3] = '\n';
s2 = sdscatrepr(s2,s1,sdslen(s1));
printf("%s\n", s2);

output> "a\x01\x02\n"
```

This is the rules `sdscatrepr` uses for conversion:

- \ and " are quoted with a backslash.
- It quotes special characters `'\n'`, `'\r'`, `'\t'`, `'\a'` and `'\b'`.
- All the other non printable characters not passing the `isprint` test are quoted in `\x..` form, that is: backslash
  followed by x followed by two digit hex number representing the character byte value.
- The function always adds initial and final double quotes characters.

There is an SDS function that is able to perform the reverse conversion and is documented in the *Tokenization*
paragraph below.

Tokenization

Tokenization is the process of splitting a larger string into smaller strings. In this specific case, the split is
performed specifying another string that acts as separator. For example in the following string there are two substrings
that are separated by the `|-|` separator:

```
foo|-|bar|-|zap
```

A more common separator that consists of a single character is the comma:

```
foo,bar,zap
```

In many progrems it is useful to process a line in order to obtain the sub strings it is composed of, so SDS provides a
function that returns an array of SDS strings given a string and a separator.

```
sds *sdssplitlen(const char *s, int len, const char *sep, int seplen, int
*count);
void sdsfreesplitres(sds *tokens, int count);
```

As usually the function can work with both SDS strings or normal C strings. The first two arguments `s` and `len`
specify the string to tokenize, and the other two arguments `sep` and `seplen` the separator to use during the
tokenization. The final argument `count` is a pointer to an integer that will be set to the number of tokens (sub
strings) returned.

The return value is a heap allocated array of SDS strings.

```
sds *tokens;
int count, j;

sds line = sdsnew("Hello World!");
tokens = sdssplitlen(line,sdslen(line)," ",1,&count);

for (j = 0; j < count; j++)
    printf("%s\n", tokens[j]);
sdsfreesplitres(tokens,count);

output> Hello
output> World!
```

The returned array is heap allocated, and the single elements of the array are normal SDS strings. You can free
everything calling `sdsfreesplitres`
as in the example. Alternativey you are free to release the array yourself using the `free` function and use and/or free
the individual SDS strings as usually.

A valid approach is to set the array elements you reused in some way to
`NULL`, and use `sdsfreesplitres` to free all the rest.

Command line oriented tokenization

Splitting by a separator is a useful operation, but usually it is not enough to perform one of the most common tasks
involving some non trivial string manipulation, that is, implementing a **Command Line Interface** for a program.

This is why SDS also provides an additional function that allows you to split arguments provided by the user via the
keyboard in an interactive manner, or via a file, network, or any other mean, into tokens.

```
sds *sdssplitargs(const char *line, int *argc);
```

The `sdssplitargs` function returns an array of SDS strings exactly like
`sdssplitlen`. The function to free the result is also identical, and is
`sdsfreesplitres`. The difference is in the way the tokenization is performed.

For example if the input is the following line:

```
call "Sabrina"    and "Mark Smith\n"
```

The function will return the following tokens:

- "call"
- "Sabrina"
- "and"
- "Mark Smith\n"

Basically different tokens need to be separated by one or more spaces, and every single token can also be a quoted
string in the same format that

`sdscatrepr` is able to emit.

String joining

There are two functions doing the reverse of tokenization by joining strings into a single one.

```
sds sdsjoin(char **argv, int argc, char *sep, size_t seplen);
sds sdsjoinsds(sds *argv, int argc, const char *sep, size_t seplen);
```

The two functions take as input an array of strings of length `argc` and a separator and its length, and produce as
output an SDS string consisting of all the specified strings separated by the specified separator.

The difference between `sdsjoin` and `sdsjoinsds` is that the former accept C null terminated strings as input while the
latter requires all the strings in the array to be SDS strings. However because of this only `sdsjoinsds` is able to
deal with binary data.

```
char *tokens[3] = {"foo","bar","zap"};
sds s = sdsjoin(tokens,3,"|",1);
printf("%s\n", s);

output> foo|bar|zap
```

Error handling

All the SDS functions that return an SDS pointer may also return `NULL` on out of memory, this is basically the only
check you need to perform.

However many modern C programs handle out of memory simply aborting the program so you may want to do this as well by
wrapping `malloc` and other related memory allocation calls directly.

SDS internals and advanced usage

At the very beginning of this documentation it was explained how SDS strings are allocated, however the prefix stored
before the pointer returned to the user was classified as an *header* without further details. For an advanced usage it
is better to dig more into the internals of SDS and show the structure implementing it:

```
struct sdshdr {
    int len;
    int free;
    char buf[];
};
```

As you can see, the structure may resemble the one of a conventional string library, however the `buf` field of the
structure is different since it is not a pointer but an array without any length declared, so `buf`

actually points at
the first byte just after the `free` integer. So in order to create an SDS string we just allocate a piece of memory
that is as large as the
`sdshdr` structure plus the length of our string, plus an additional byte for the mandatory null term that every SDS
string has.

The `len` field of the structure is quite obvious, and is the current length of the SDS string, always computed every
time the string is modified via SDS function calls. The `free` field instead represents the amount of free memory in the
current allocation that can be used to store more characters.

So the actual SDS layout is this one:

```
+------------+-----------------------+-----------+---------------\
| Len | Free | H E L L O W O R L D \n | Null term |  Free space   \
+------------+-----------------------+-----------+---------------\
             |
             `-> Pointer returned to the user.
```

You may wonder why there is some free space at the end of the string, it looks like a waste. Actually after a new SDS
string is created, there is no free space at the end at all: the allocation will be as small as possible to just hold
the header, string, and null term. However other access patterns will create extra free space at the end, like in the
following program:

```
s = sdsempty();
s = sdscat(s,"foo");
s = sdscat(s,"bar");
s = sdscat(s,"123");
```

Since SDS tries to be efficient it can't afford to reallocate the string every time new data is appended, since this
would be very inefficient, so it uses the **preallocation of some free space** every time you enlarge the string.

The preallocation algorithm used is the following: every time the string is reallocated in order to hold more bytes, the
actual allocation size performed is two times the minimum required. So for instance if the string currently is holding
30 bytes, and we concatenate 2 more bytes, instead of allocating 32 bytes in total SDS will allocate 64 bytes.

However there is an hard limit to the allocation it can perform ahead, and is defined by `SDS_MAX_PREALLOC`. SDS will
never allocate more than 1MB of additional space (by default, you can change this default).

Shrinking strings

```
sds sdsRemoveFreeSpace(sds s);
size_t sdsAllocSize(sds s);
```

Sometimes there are class of programs that require to use very little memory. After strings concatenations, trimming,
ranges, the string may end having a non trivial amount of additional space at the end.

It is possible to resize a string back to its minimal size in order to hold the current content by using the
function `sdsRemoveFreeSpace`.

```
s = sdsRemoveFreeSpace(s);
```

There is also a function that can be used in order to get the size of the total allocation for a given string, and is
called `sdsAllocSize`.

```
sds s = sdsnew("Ladies and gentlemen");
s = sdscat(s,"... welcome to the C language.");
printf("%d\n", (int) sdsAllocSize(s));
s = sdsRemoveFreeSpace(s);
printf("%d\n", (int) sdsAllocSize(s));

output> 109
output> 59
```

NOTE: SDS Low level API use cammelCase in order to warn you that you are playing with the fire.

Manual modifications of SDS strings

```
void sdsupdatelen(sds s);
```

Sometimes you may want to hack with an SDS string manually, without using SDS functions. In the following example we
implicitly change the length of the string, however we want the logical length to reflect the null terminated C string.

The function `sdsupdatelen` does just that, updating the internal length information for the specified string to the
length obtained via `strlen`.

```
sds s = sdsnew("foobar");
s[2] = '\0';
printf("%d\n", sdslen(s));
sdsupdatelen(s);
printf("%d\n", sdslen(s));

output> 6
output> 2
```

Sharing SDS strings

If you are writing a program in which it is advantageous to share the same SDS string across different data structures,
it is absolutely advised to encapsulate SDS strings into structures that remember the number of references of the
string, with functions to increment and decrement the number of references.

This approach is a memory management technique called *reference counting* and in the context of SDS has two advantages:

- It is less likely that you'll create memory leaks or bugs due to non freeing SDS strings or freeing already freed
  strings.
- You'll not need to update every reference to an SDS string when you modify it (since the new SDS string may point to a
  different memory location).

While this is definitely a very common programming technique I'll outline the basic ideas here. You create a structure
like that:

```
struct mySharedStrings {
    int refcount;
    sds string;
}
```

When new strings are created, the structure is allocated and returned with
`refcount` set to 1. The you have two functions to change the reference count of the shared string:

- `incrementStringRefCount` will simply increment `refcount` of 1 in the structure. It will be called every time you add
  a reference to the string on some new data structure, variable, or whatever.
- `decrementStringRefCount` is used when you remove a reference. This function is however special since when
  the `refcount` drops to zero, it automatically frees the SDS string, and the
  `mySharedString` structure as well.

Interactions with heap checkers

Because SDS returns pointers into the middle of memory chunks allocated with
`malloc`, heap checkers may have issues, however:

- The popular Valgrind program will detect SDS strings are *possibly lost* memory and never as *definitely lost*, so it
  is easy to tell if there is a leak or not. I used Valgrind with Redis for years and every real leak was consistently
  detected as "definitely lost".
- OSX instrumentation tools don't detect SDS strings as leaks but are able to correctly handle pointers pointing to the
  middle of memory chunks.

Zero copy append from syscalls

At this point you should have all the tools to dig more inside the SDS library by reading the source code, however there
is an interesting pattern you can mount using the low level API exported, that is used inside Redis in order to improve
performances of the networking code.

Using `sdsIncrLen()` and `sdsMakeRoomFor()` it is possible to mount the following schema, to cat bytes coming from the
kernel to the end of an sds string without copying into an intermediate buffer:

```
oldlen = sdslen(s);
s = sdsMakeRoomFor(s, BUFFER_SIZE);
nread = read(fd, s+oldlen, BUFFER_SIZE);
... check for nread <= 0 and handle it ...
sdsIncrLen(s, nread);
```

`sdsIncrLen` is documented inside the source code of `sds.c`.

Embedding SDS into your project

This is as simple as copying the `sds.c` and `sds.h` files inside your project. The source code is small and every C99
compiler should deal with it without issues.

Credits and license

SDS was created by Salvatore Sanfilippo and is released under the BDS two clause license. See the LICENSE file in this
source distribution for more information.