This resource consists of libraries which originally have been divided into 3 separate resources. However, from a conceptional perspective they fit together, and they may depend on each other. That's the reason why I decided to bundle them in one package. A few updates have been made, too.

*** cstr – string library ***

fully compatible with nullterminated C-strings

History

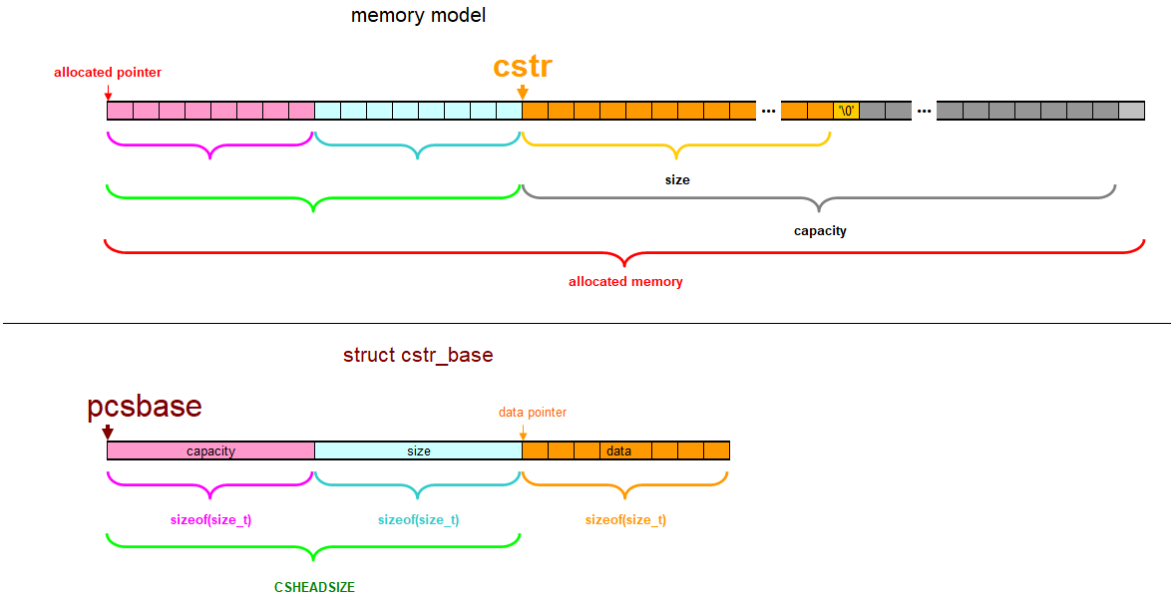
There are several possibilities of how to define a string. For historical reasons C uses an implementation which eventually turned out to be inconvenient. How can you imagine this?
Say, there are two German villages, called Hundsdorf and Katzleben. Peter is about to go from Hundsdorf to Katzleben by bike because he wants to know the distance between the two. Leonie is waiting for him at the entrance of Katzleben. So, he resets the odometer and starts. Exhausted he arrives in Katzleben after 15 minutes and asks Leonie if she knew the distance to Hundsdorf was 10 km? Leonie turns her head to the left and points to the backside of the place-name sign which states "Hundsdorf 10 km".
Well, German town signs are comfortable, and it wasn't worth the ride only to get the length of the street. But let's come back to C where a string doesn't have this luxury. To measure the length of a string you are forced to search for the terminating null at its end. Thus, you have to walk down the entire string to find it, just like Peter on his bike.
Pioneers of software development already considered this a poor concept. An alternative string implementation is known as Pascal string. The memory space a Pascal string takes is the same as for a C string. But rather than appending a null byte, the first byte is used to store the length of the subsequent string. The advantage is that the complexity of the operation to determine the string end drops from O(n) to O(1). Furthermore, the string may contain null bytes which enables you to store binary data as well. However, the type size of a byte limits the maximum length to 255. Obviously, the original Pascal string still had a lack of maturity. But it has been a nice concept for the beginning. Besides of the pointer to char, recent string implementations hold the length of the string and the capacity of the allocated memory in a structure with members of reasonable size. That gives them a high level of flexibility and performance as necessary reallocations can be reduced. You can't use those string classes for a direct interaction with the functions of the C standard library though. To achieve this, we have to get back to the implementation of a Pascal string where all information is stored in a consecutive block of memory. The offset to the actual string is one byte there. But why should we be limited to one byte as long as only the offset remains the same … ? ;-)

`cstr` - impementation

The aim is to combine the advantages of a string class with the compatibility to nullterminated C strings.
Example:
If a `cstr` was created …
```
cstr cs = cs_init("ABC");
```
… then …
```
char ch = cs[1];
```
… shall assign 'B' to variable ch, or …
```
char *ptr = strchr(cs, 'B');
```
… shall determine the pointer to 'B' in the `cstr`, or …
```
puts(cs);
```
… shall print the string `ABC`.
In order to ensure this works we have to store the capacity and the string length along with the actual string in a consecutive memory space. The user only gets the pointer to the first character of the actual string. The prepended area which contains the capacity and the length remains being hidden in the implementation. These data have the width of a `size_t` each. Thus, the offset between the beginning of the allocated memory range and the first character of the string is always the same. Hence this offset can be used to step backwards from the pointer to the first character to get the pointer to the whole allocated memory. For the sake of ease a helper structure `cstr_base` is implemented. It'll never be instantiated. Only its pointer type is used to access the capacity, length, and char pointer by their member names.
The disadvantage of this implementation is that there is no possibility to include a mutex to get the string thread-safe by design. All data is accessed via the `cstr` pointer. If the pointer value is altered due to possible reallocations, the pointer

to a mutex would become invalid before a concurrent accessor had the chance to lock it. Furthermore, you should rather use only one variable for the same `cstr`. Functions may update the memory address a `cstr` variable holds. Copies of this pointer potentially become invalid.

The figure below shows how the data is stored in memory. The `cstr` pointer is the interface used in the library functions. The scheme of the `cstr_base` structure explains how and why it can be used to get access to the values even if the `data` member is only a placeholder for the actual string.



The two defined types to represent a cstr in this library are pointers to char.
They exist to semantically separate usual C strings from the cstr strings with their special kind of allocation.

```
typedef char *cstr;
typedef const char *ccstr;
```

A structure type is implemented to store exclusive information for the partials handling in UTF-8-encoded strings.

```
typedef struct tag_u8state u8state_t;
```

The `BIN` macro has no functionality. It's just a marker for those functions of the cstr library which can be used along with binary data in a string.

The functions below are used to create and destroy cstr strings:

```
cstr cs_init(const char *str);
cstr cs_init_format(const char *format, ...);
BIN cstr cs_init_n(const char *seq, size_t cnt);
BIN cstr cs_init_rdentirefile(FILE *stream);
BIN cstr cs_init_rdfile(FILE *stream, size_t cnt);
cstr cs_init_rdline(FILE *stream);
BIN cstr cs_init_zero(size_t size);
BIN void cs_free(ccstr cs);
```

Get the maximum number of chars that technically can be saved into a cstr:

```
size_t cs_max_size(void);
```

For the processing of cstr strings functions are used where many of them have names and a functionality derived from the C++ string library:

```
cstr cs_append(cstr *pcs, const char *str);
BIN cstr cs_append_n(cstr *pcs, const char *seq, size_t cnt);
cstr cs_assign(cstr *pcs, const char *str);
BIN cstr cs_assign_n(cstr *pcs, const char *seq, size_t cnt);
BIN char cs_at(ccstr cs, size_t pos);
BIN size_t cs_capacity(ccstr cs);
BIN cstr cs_clear(cstr cs);
BIN bool cs_empty(ccstr cs);
BIN cstr cs_erase(cstr cs, size_t pos, size_t length);
cstr cs_fix(cstr *pcs, size_t length, const char *pad, unsigned mode);
cstr cs_insert(cstr *pcs, size_t pos, const char *str);
BIN cstr cs_insert_n(cstr *pcs, size_t pos, const char *seq, size_t cnt);
BIN size_t cs_length(ccstr cs);
BIN char cs_pop_back(cstr cs);
BIN cstr cs_push_back(cstr *pcs, const char ch);
cstr cs_replace(cstr *pcs, size_t pos, size_t length, const char *str);
BIN cstr cs_replace_n(cstr *pcs, size_t pos, size_t length, const char *seq, size_t cnt);
BIN cstr cs_reserve(cstr *pcs, size_t capa);
BIN cstr cs_resize(cstr *pcs, size_t size, char fill);
BIN cstr cs_reverse(cstr cs);
BIN cstr cs_shrink_to_fit(cstr *pcs);
BIN size_t cs_size(ccstr cs);
BIN cstr cs_substr(ccstr cs, size_t pos, size_t length);
cstr cs_trim(cstr cs, const char *char_set, unsigned mode);
BIN cstr cs_update_length(cstr cs, size_t length);
```

Furthermore a `cstr` string can be used directly with the standard string functions of the C library.

For strings based on `wchar_t` the two types `cstrw` and `ccstrw` are implemented. The related functions begin with prefix `csw_...` rather than `cs_...`.

Functions especially for UTF-8-encoded text:

```
size_t cs_utf8_count(ccstr cs);
cstr cs_utf8_handle_partials(cstr *pcs, u8state_t *pstate);
bool cs_utf8_hasbom(ccstr cs);
size_t cs_utf8_idx_of_nth_codepoint(ccstr cs, size_t pos);
cstr cs_utf8_replace_invalids(cstr *pcs, const char *rep);
```

For descriptions of the functions, their parameters, and their return values refer to the `cstr.h` header.

*** memmem – searching of byte sequences in binary data ***

The `memmem` function has a similar behaviour as strstr. It addresses the possibility of the cstr implementation to work with binary data.
While `memmem` is searching for the first occurrence, the `memrmem` is searching for the last occurrence of a byte sequence in a data range.
They don't fit into the cstr library because they are rather extensions for the string library in the <string.h> header. Because this header belongs to the C standard library, the additional functions got their own source-header pair. They can be used independently of the cstr library.

```
void *memmem(const void *buffer, size_t buffer_size, const void *target, size_t target_size);
void *memrmem(const void *buffer, size_t buffer_size, const void *target, size_t target_size);
```

The functions ...

```
char *strrstr(const char *str, const char *substr);
wchar_t *wcsrstr(const wchar_t *strw, const wchar_t *substrw);
```

... are implemented especially for searching backwards in nullterminated strings.

*** cvec – vector library ***

arrays with dynamic memory management

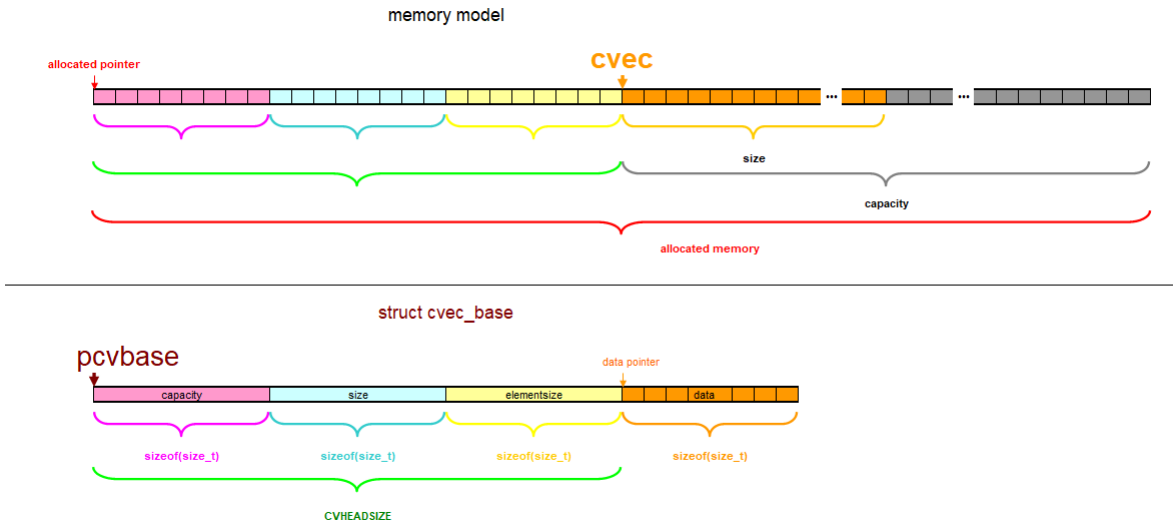
Preface

In the description of the cstr library I already wrote a little about the concept. The idea behind the implementation of a `cvec` is also based on the Pascal string which is a length-prefixed string. In a cstr this is insofar generalized that not only the length is preceded but also the capacity. Eventually a string is an array of char values. It's only a tiny step to get to the idea of generalizing the concept even more in order to get rid of the constraints of using character types only. In theory this is no problem at all. But if we adopt the cstr concept we would have to write a library for every particular element type of all possible arrays. C lacks the opportunity of having templates like in C++. Thus, as always in such cases, we have to use untyped pointers `void*` and have to bear all of their additional risks …

`cvec` – implementation

Again the conceptional aim is that the users directly get a pointer to the first element of the vector, while the properties are hidden in the implementation. Also, for a cvec an offset is used to access these data.
Because we have to work with untyped pointers we additionally need the type size of an element. So, a cvec holds the capacity, the length, and the type size besides of the actual vector data. They are accessed using a helper structure `cvec_base`.
In general also multidimensional `cvec` vectors are feasible (refer to the example in `main.c`).

The figure below shows how the data is stored in memory. The `cvec` pointer is the interface used in the library functions. The scheme of the `cvec_base` structure explains how and why it can be used to get access to the values.



The two defined types in this library are pointers to void.
They exist to semantically separate cvec pointers with their special kind of allocation from other pointers.

```
typedef void *cvec;
typedef const void *ccvec;
```

Also the pointer type of a callback function to be used along with `cv_for_each` is defined.

```
typedef bool(*cv_for_each_proc_t)(void *element_data, void *user_param);
```

The `PTRTO` macro casts a cvec to a pointer to the actual element type.
The `MAKEPTR` macro generates a pointer to a single value or to a list of values.

The functions below are used to create and destroy cvec vectors:

```
cvec cv_init(const void *seq, size_t cnt, size_t elementsize);
cvec cv_init_zero(size_t size, size_t elementsize);
void cv_free(ccvec cv);
```

Get the maximum number of elements that technically can be saved into a cvec:

```
size_t cv_max_size(size_t elementsize);
```

For the processing of cvec vectors functions are used where many of them have names and a functionality derived from the C++ vector library:

```
cvec cv_append(cvec *pcv, const void *seq, size_t cnt);
cvec cv_assign(cvec *pcv, const void *seq, size_t cnt);
const void *cv_at(ccvec cv, size_t pos);
size_t cv_capacity(ccvec cv);
cvec cv_clear(cvec cv);
bool cv_empty(ccvec cv);
cvec cv_erase(cvec cv, size_t pos, size_t cnt);
bool cv_for_each(cvec cv, size_t pos, size_t cnt, cv_for_each_proc_t callback_func, void *user_param);
cvec cv_insert(cvec *pcv, size_t pos, const void *seq, size_t cnt);
const void *cv_pop_back(cvec cv);
cvec cv_push_back(cvec *pcv, const void *elem);
cvec cv_reserve(cvec *pcv, size_t capa);
cvec cv_resize(cvec *pcv, size_t size, const void *elem);
cvec cv_shrink_to_fit(cvec *pcv);
size_t cv_size(ccvec cv);
cvec cv_update_size(cvec cv, size_t size);
```

For descriptions of the functions, their parameters, and their return values refer to the `cvec.h` header.

*** cstrvec – interaction between cstr and cvec ***

dependent on cstr and cvec

When I began to work on @BAGZZlash 's suggestion to implement a Split and a Join function into the `cstr` lib it was instantly clear to me why C++ doesn't have those functions in its string class. They just don't fit. The simple reason is that we need both a string and a vector implementation. To avoid an *unconditional* dependency of the `cstr` and `cvec` libs this little additional library exists only for functions where this dependency is necessary.

The `cstrvec` lib contains functions to concatenate strings of a vector, or to divide a string into a vector using delimiters ...
`cstr cs_join(ccvec strvec, const char *delims);` und
`cvec cs_split(ccstr cs, const char *delims, size_t maxtok);`

... and to read a text file line-wise into a vector.
`cvec cstrvec_rdlines(FILE *stream, size_t maxlines);`

There is also an additional function for the special case "vector of strings" ...
`void cstrvec_free(ccvec strvec);`
... which unites the memory release of the cstr elements and the cvec container.

For the same reason do 3 additional macros exist.
`PTRTOCSTR(_cvec)`
`MAKECSTRPTR(...)`
`MAKECCSTRPTR(...)`
Those are specializations for cstr of the `PTRTO` and `MAKEPTR` macros in the cvec library.

Also for strings based on `wchar_t` the related functions and macros are implemented:
`cstrw csw_join(ccvec strwvec, const wchar_t *delimsw);`
`cvec csw_split(ccstrw csw, const wchar_t *delimsw, size_t maxtok);`
`cvec cstrwvec_rdlines(FILE *stream, size_t maxlines);`
`void cstrwvec_free(ccvec strwvec);`
`PTRTOCSTRW(_cvec)`
`PTRTOCCSTRW(_cvec)`
`MAKECSTRWPTR(...)`

To create a string from a vector of bytes or vice versa use the following functions:
`cstr cs_from_cvbyte(ccvec cv);`
`cvec cvbyte_from_cs(ccstr cs);`

For detailed descriptions refer to the `cstrvec.h` header.

The `main.c` file contains test code and examples of how to work with the functions and macros.

I'm looking forward to your questions, bug reports, feature requests, and feedback of any kind.