utringbuffer: dynamic ring-buffer macros for C
==
Arthur O'Dwyer <arthur.j.odwyer@gmail.com>
v2.3.0, February 2021

Here's a link back to the https://github.com/troydhanson/uthash[GitHub project
page].

Introduction

The functions in `utringbuffer.h` are based on the general-purpose array macros
provided in `utarray.h`, so before reading this page you should read
link:utarray.html[that page] first.

To use these macros in your own C program, copy both `utarray.h` and
`utringbuffer.h`
into your source directory and use `utringbuffer.h` in your program.

  #include "utringbuffer.h"

The provided <<operations,operations>> are based loosely on the C++ STL vector
methods.
The ring-buffer data type supports construction (with a specified capacity),
destruction, iteration, and push, but not pop; once the ring-buffer reaches full
capacity, pushing a new element automatically pops and destroys the oldest
element.
The elements contained in the ring-buffer can be any simple datatype or
structure.

Internally the ring-buffer contains a pre-allocated memory region into which the
elements are copied, starting at position 0. When the ring-buffer reaches full
capacity, the next element to be pushed is pushed at position 0, overwriting the
oldest element, and the internal index representing the "start" of the ring-
buffer
is incremented. A ring-buffer, once full, can never become un-full.

Download
~~~~~~~~
To download the `utringbuffer.h` header file,
follow the links on https://github.com/troydhanson/uthash to clone uthash or get
a zip file,
then look in the src/ sub-directory.

BSD licensed
~~~~~~~~~~~~
This software is made available under the
link:license.html[revised BSD license].
It is free and open source.

Platforms
~~~~~~~~~
The 'utringbuffer' macros have been tested on:

 * Linux,
 * Mac OS X,
 * Windows, using Visual Studio 2008 and Visual Studio 2010

Usage

Declaration
~~~~~~~~~~~

The ring-buffer itself has the data type `UT_ringbuffer`, regardless of the type of elements to be stored in it. It is declared like,

```
  UT_ringbuffer *history;
```

New and free

The next step is to create the ring-buffer using `utringbuffer_new`. Later when you're done with the ring-buffer, `utringbuffer_free` will free it and all its elements.

Push, etc

The central features of the ring-buffer involve putting elements into it and iterating over them. There are several <<operations,operations>> that deal with either single elements or ranges of elements at a time. In the examples below we will use only the push operation to insert elements.

Elements

Support for dynamic arrays of integers or strings is especially easy. These are best shown by example:

Integers

This example makes a ring-buffer of integers, pushes 0-9 into it, then prints it two different ways. Lastly it frees it.

.Integer elements

```
#include <stdio.h>
#include "utringbuffer.h"

int main() {
  UT_ringbuffer *history;
  int i, *p;

  utringbuffer_new(history, 7, &ut_int_icd);
  for(i=0; i < 10; i++) utringbuffer_push_back(history, &i);

  for (p = (int*)utringbuffer_front(history);
       p != NULL;
       p = (int*)utringbuffer_next(history, p)) {
    printf("%d\n", *p);  /* prints "3 4 5 6 7 8 9" */
  }

  for (i=0; i < utringbuffer_len(history); i++) {
    p = utringbuffer_eltptr(history, i);
    printf("%d\n", *p);  /* prints "3 4 5 6 7 8 9" */
  }

  utringbuffer_free(history);

  return 0;
}
```

The second argument to `utringbuffer_push_back` is always a 'pointer' to the type (so a literal cannot be used). So for integers, it is an `int*`.

Strings
~~~~~~~
In this example we make a ring-buffer of strings, push two strings into it, print
it and free it.

.String elements
```
--------------------------------------------------------------------------------
#include <stdio.h>
#include "utringbuffer.h"

int main() {
  UT_ringbuffer *strs;
  char *s, **p;

  utringbuffer_new(strs, 7, &ut_str_icd);

  s = "hello"; utringbuffer_push_back(strs, &s);
  s = "world"; utringbuffer_push_back(strs, &s);
  p = NULL;
  while ( (p=(char**)utringbuffer_next(strs,p))) {
    printf("%s\n",*p);
  }

  utringbuffer_free(strs);

  return 0;
}
--------------------------------------------------------------------------------
```

In this example, since the element is a `char*`, we pass a pointer to it
(`char**`) as the second argument to `utringbuffer_push_back`. Note that "push" makes
a copy of the source string and pushes that copy into the array.

About UT_icd
~~~~~~~~~~~~

Arrays can be made of any type of element, not just integers and strings. The
elements can be basic types or structures. Unless you're dealing with integers
and strings (which use pre-defined `ut_int_icd` and `ut_str_icd`), you'll need
to define a `UT_icd` helper structure. This structure contains everything that
utringbuffer (or utarray) needs to initialize, copy or destruct elements.

```
  typedef struct {
      size_t sz;
      init_f *init;
      ctor_f *copy;
      dtor_f *dtor;
  } UT_icd;
```

The three function pointers `init`, `copy`, and `dtor` have these prototypes:

```
  typedef void (ctor_f)(void *dst, const void *src);
  typedef void (dtor_f)(void *elt);
  typedef void (init_f)(void *elt);
```

The `sz` is just the size of the element being stored in the array.

The `init` function is used by utarray but is never used by utringbuffer;
you may safely set it to any value you want.

The `copy` function is used whenever an element is copied into the buffer.

It is invoked during `utringbuffer_push_back`.
If `copy` is `NULL`, it defaults to a bitwise copy using memcpy.

The `dtor` function is used to clean up an element that is being removed from the buffer. It may be invoked due to `utringbuffer_push_back` (on the oldest element in the buffer), `utringbuffer_clear`, `utringbuffer_done`, or `utringbuffer_free`.
If the elements need no cleanup upon destruction, `dtor` may be `NULL`.

Scalar types
~~~~~~~~~~~~

The next example uses `UT_icd` with all its defaults to make a ring-buffer of `long` elements. This example pushes two longs into a buffer of capacity 1, prints the contents of the buffer (which is to say, the most recent value pushed), and then frees the buffer.

.long elements
```
--------------------------------------------------------------------------------
#include <stdio.h>
#include "utringbuffer.h"

UT_icd long_icd = {sizeof(long), NULL, NULL, NULL };

int main() {
  UT_ringbuffer *nums;
  long l, *p;
  utringbuffer_new(nums, 1, &long_icd);

  l=1; utringbuffer_push_back(nums, &l);
  l=2; utringbuffer_push_back(nums, &l);

  p=NULL;
  while((p = (long*)utringbuffer_next(nums,p))) printf("%ld\n", *p);

  utringbuffer_free(nums);
  return 0;
}
--------------------------------------------------------------------------------
```

Structures
~~~~~~~~~~

Structures can be used as utringbuffer elements. If the structure requires no special effort to initialize, copy or destruct, we can use `UT_icd` with all its defaults. This example shows a structure that consists of two integers. Here we push two values, print them and free the buffer.

.Structure (simple)
```
--------------------------------------------------------------------------------
#include <stdio.h>
#include "utringbuffer.h"

typedef struct {
    int a;
    int b;
} intpair_t;

UT_icd intpair_icd = {sizeof(intpair_t), NULL, NULL, NULL};

int main() {

  UT_ringbuffer *pairs;
  intpair_t ip, *p;
```

```
  utringbuffer_new(pairs, 7, &intpair_icd);

  ip.a=1;  ip.b=2;  utringbuffer_push_back(pairs, &ip);
  ip.a=10; ip.b=20; utringbuffer_push_back(pairs, &ip);

  for(p=(intpair_t*)utringbuffer_front(pairs);
      p!=NULL;
      p=(intpair_t*)utringbuffer_next(pairs,p)) {
    printf("%d %d\n", p->a, p->b);
  }

  utringbuffer_free(pairs);
  return 0;
}
```

The real utility of `UT_icd` is apparent when the elements stored in the ring-buffer are structures that require special work to initialize, copy or destruct.

For example, when a structure contains pointers to related memory areas that need to be copied when the structure is copied (and freed when the structure is freed), we can use custom `init`, `copy`, and `dtor` members in the `UT_icd`.

Here we take an example of a structure that contains an integer and a string. When this element is copied (such as when an element is pushed), we want to "deep copy" the `s` pointer (so the original element and the new element point to their own copies of `s`). When an element is destructed, we want to "deep free" its copy of `s`. Lastly, this example is written to work even if `s` has the value `NULL`.

.Structure (complex)

```
#include <stdio.h>
#include <stdlib.h>
#include "utringbuffer.h"

typedef struct {
    int a;
    char *s;
} intchar_t;

void intchar_copy(void *_dst, const void *_src) {
  intchar_t *dst = (intchar_t*)_dst, *src = (intchar_t*)_src;
  dst->a = src->a;
  dst->s = src->s ? strdup(src->s) : NULL;
}

void intchar_dtor(void *_elt) {
  intchar_t *elt = (intchar_t*)_elt;
  free(elt->s);
}

UT_icd intchar_icd = {sizeof(intchar_t), NULL, intchar_copy, intchar_dtor};

int main() {
  UT_ringbuffer *intchars;
  intchar_t ic, *p;
  utringbuffer_new(intchars, 2, &intchar_icd);

  ic.a=1; ic.s="hello"; utringbuffer_push_back(intchars, &ic);
  ic.a=2; ic.s="world"; utringbuffer_push_back(intchars, &ic);
  ic.a=3; ic.s="peace"; utringbuffer_push_back(intchars, &ic);
```

```
  p=NULL;
  while( (p=(intchar_t*)utringbuffer_next(intchars,p))) {
    printf("%d %s\n", p->a, (p->s ? p->s : "null"));
    /* prints "2 world 3 peace" */
  }

  utringbuffer_free(intchars);
  return 0;
}
```

```
-------------------------------------------------------------------------------

[[operations]]
Reference
---------
This table lists all the utringbuffer operations. These are loosely based on the
C++
vector class.

Operations
~~~~~~~~~~

[width="100%",cols="50<m,40<",grid="none",options="none"]
|===============================================================================
| utringbuffer_new(UT_ringbuffer *a, int n, UT_icd *icd)  | allocate a new
ringbuffer
| utringbuffer_free(UT_ringbuffer *a)                     | free an allocated
ringbuffer
| utringbuffer_init(UT_ringbuffer *a, int n, UT_icd *icd) | init a ringbuffer
(non-alloc)
| utringbuffer_done(UT_ringbuffer *a)                     | dispose of a
ringbuffer (non-alloc)
| utringbuffer_clear(UT_ringbuffer *a)                    | clear all elements
from a, making it empty
| utringbuffer_push_back(UT_ringbuffer *a, element *p)    | push element p onto
a
| utringbuffer_len(UT_ringbuffer *a)                      | get length of a
| utringbuffer_empty(UT_ringbuffer *a)                    | get whether a is
empty
| utringbuffer_full(UT_ringbuffer *a)                     | get whether a is
full
| utringbuffer_eltptr(UT_ringbuffer *a, int j)            | get pointer of
element from index
| utringbuffer_eltidx(UT_ringbuffer *a, element *e)       | get index of element
from pointer
| utringbuffer_front(UT_ringbuffer *a)                    | get oldest element
of a
| utringbuffer_next(UT_ringbuffer *a, element *e)         | get element of a
following e (front if e is NULL)
| utringbuffer_prev(UT_ringbuffer *a, element *e)         | get element of a
before e (back if e is NULL)
| utringbuffer_back(UT_ringbuffer *a)                     | get newest element
of a
|===============================================================================

Notes
~~~~~

1. `utringbuffer_new` and `utringbuffer_free` are used to allocate a new ring-
buffer
   and to free it,
   while `utringbuffer_init` and `utringbuffer_done` can be used if the
UT_ringbuffer
   is already allocated and just needs to be initialized or have its internal
```

resources
   freed.
2. Both `utringbuffer_new` and `utringbuffer_init` take a second parameter `n` indicating
   the capacity of the ring-buffer, that is, the size at which the ring-buffer is considered
   "full" and begins to overwrite old elements with newly pushed ones.
3. Once a ring-buffer has become full, it will never again become un-full except by
   means of `utringbuffer_clear`. There is no way to "pop" a single old item from the
   front of the ring-buffer. You can simulate this ability by maintaining a separate
   integer count of the number of "logically popped elements", and starting your iteration
   with `utringbuffer_eltptr(a, popped_count)` instead of with `utringbuffer_front(a)`.
4. Pointers to elements (obtained using `utringbuffer_eltptr`, `utringbuffer_front`,
   `utringbuffer_next`, etc.) are not generally invalidated by `utringbuffer_push_back`,
   because utringbuffer does not perform reallocation; however, a pointer to the oldest
   element may suddenly turn into a pointer to the 'newest' element if
   `utringbuffer_push_back` is called while the buffer is full.
5. The elements of a ring-buffer are stored in contiguous memory, but once the ring-buffer
   has become full, it is no longer true that the elements are contiguously in order from
   oldest to newest; i.e., `(element *)utringbuffer_front(a) + utringbuffer_len(a)-1` is
   not generally equal to `(element *)utringbuffer_back(a)`.

// vim: set nowrap syntax=asciidoc: