# 20171127:Practical:C:Malloc

## Introduction

The purpose of this session is to implement a simple version of `malloc(3)` and the affiliated functions (`free`, `calloc` and `realloc`).

This session will also be used as a mini-project that you start this week and with an extra week to finish it.

> **Deadline:** Tuesday 2017/12/12 - 23h42 (edit)

You must commit the code in the directory `20171127_malloc` and you must provide the following files:

- `20171127_malloc`
  - `AUTHORS`
  - `Makefile`
  - `malloc.c`

Your `Makefile` must be able to build the library file `libmalloc.so` (see the end of the subject.)

The file `malloc.c` **must not** contain any `main` function.

## Prequel: debugging

Before starting your malloc, you should train yourself on gdb. This is important and nothing replace practice on you own code. So, you should take a piece of your own code, with some known bugs and try the following actions:

- build with symbols (at least `-g` and no optimization.)
- run your program in gdb
- in the presence of a segfault, use gdb to detect the position in code and the context
- use breakpoints and step by step execution
- evaluate some expression and get some context information

**Beware:** you will **not** be able to use `printf` for malloc debugging !

## Goals

You need to provide a full replacement for `malloc(3)` but also the affiliated functions, here is the list of the **required functions**:

```
void* malloc(size_t size);
void* calloc(size_t nb, size_t size);
void* realloc(void *old, size_t newsize);
void  free(void *p);
```

**You must first read the manual page for these functions (on most Unix, you've got one page describing the four functions) in order to get all the details.** Important part are *DESCRIPTION* and *RETURN VALUE*.

## Tools

These functions will be useful for the rest but don't need to be *exported* and thus you'll marked them as `static`.

### Alignment

We need to work with size that are multiple of the size of a pointer. For that we'll use a simple `static inline` function that take a `size_t` integer and return the least multiple of `sizeof (size_t)` greater or equal to the input value.

```
static inline
size_t word_align(size_t n);
```

You can perform this alignment with only the following operations: `+`, `-`, `^`, `~` and `&`. The size obtained by `sizeof (size_t)` will be either 4 or 8 in most cases (at least you can expect it to be a power of 2), as you probably know, a multiple of a power of 2 has its lower bits set to `0` (two `0` for a multiple of 4 and 3 for a multiple of 8 and so on … ) From this, getting a previous multiple of 4 or 8 is obtained by zeroing the last bits. The rest is obvious …

If you want to convince yourself, here is a simple piece of python that demonstrate this idea:

```
from random import randint

def to_bin(n):
    s = ''
    while n > 0:
        s = str(n % 2) + s
        n = n // 2
    return s

if __name__ == '__main__':
    n = 8 * randint(1, 100)
    print(n, ':', to_bin(n))
    print(7, ':', to_bin(7))
    print(31, ':', to_bin(31))
    a, b = 31, 31 ^ 7
    # b is a XOR 0b111
    print('31 XOR 0b111', ':', to_bin(b), '=', b)
```

## Basic memory operations

We need to be able to copy data between two separated memory regions and to be able to fill a memory region with zeros. Functions doing this exists in the C library, but our operations can have some specific optimizations and thus we'll write our own version.

- **Implement the following functions:**

```
/* zerofill(ptr, len): write len 0 bytes at the address ptr */
void zerofill(void *ptr, size_t len);
/* wordcpy(dst, src, len) copy len bytes from src to dst */
void wordcpy(void *dst, void *src, size_t len);
```

**Note that we work with aligned sizes: all our sizes are divisible by the size of a pointer. This mean that rather than working data byte by byte, we can operate with a wider type, `size_t` is a good candidate.**

## Memory chunk structure

To represent our memory chunk, we'll use the following structure:

```
struct chunk {
  struct chunk *next, *prev;
  size_t        size;
  long          free;
  void         *data;
};
```

*Note that we don't really care about the wasted space, this is a pedagogical `malloc`.*

## Heap Base

Now we need a simple way to retrieve the base address of the heap, this function will also help us initialize the heap by setting a sentinel for our list of chunks. For that we'll use a local static variable (a variable that survive the function call), here is the function `get_base()` principle, `base` is your static variable:

- If `base` is `NULL` (initial value):
  - extend the heap by the size of the chunk structure, using `sbrk(2)`, and set `base` to the old break
  - Initialize the sentinel: `next` and `prev` must be `NULL`, the data size is zero and the chunk is not free.
- return `base`

- **Complete the following code:**

```
static
struct chunk* get_base(void) {
  static struct chunk *base = NULL;
  if (base == NULL) {
    /* FIX ME */
  }
}
```

This function is used by `malloc` in order to get the base pointer on the heap. The first to you call it, it'll initialize the heap (create an empty chunk at the beginning.) Other call will just return the base pointer.

## Heap Extension

This function will add a new chunk at the end of the heap (using `sbrk`). The new chunk must be attached to the (previous) last chunk, and must contain the required memory size in the data area. We'll directly passed a pointer to the last chunk in order to avoid a traversal of the whole heap.

```
/*
 * extend_heap(last, size) extends the heap with a new chunk containing a data
 * Return 1 in case of success, and return 0 if sbrk(2) fails.
 */
static
int extend_heap(struct chunk *last, size_t size);
```

*Note that `size` is already aligned and thus you must extends the heap by `size` bytes plus the size of the chunk structure.*

## Find a free chunk

This function is responsible to find a free chunk of at least the asked size. It start from the beginning of the heap (retrieved using the function `get_base()`) and look for the first free chunk with at least the required size. It will return the pointer to the chunk **before** the founded chunk (so you will use the `next` field to access the chunk.) If no matching chunk have been found, the function will return a pointer to the last chunk of the heap (which next pointer will be NULL.)

- **Implement the following function:**

```
static
struct chunk* find_chunk(size_t size);
```

## Retrieve Chunk Pointer from Data

For `free` and `realloc`, you'll need to retrieve the pointer to the chunk from the data. The main part of this function is to check that the given pointer is correct.

The test are:

- the pointer is not `NULL` and correctly aligned
- the pointer is between the base pointer and the current break (obtained with `sbrk(0)`)
- After retrieving the chunk pointer, you must check that your pointer is equal to the `data` field of the chunk.

If one of the test failed, you must return `NULL`.

```
static
struct chunk* get_chunk(void *p);
```

### malloc

Implement the `malloc(3)` function using the following strategy:

- Align the size
- Try to find a free chunk
- If no free chunk have been found, extend the heap
- Mark the block has been used (not free)
- Return the pointer to the data

**Beware, if the heap extension fails, you must return `NULL`**

### free

The `free(3)` operation is pretty straightforward:

- get the chunk pointer
- mark data as free

### calloc

The `calloc(3)` function is also pretty simple to implement:

- Allocate (with `malloc`) a block of `nb * size` bytes
- Fill the block with 0
- Return the pointer

Don't forget to handle error case (when `malloc` returns `NULL`.) It can also be interesting to handle integer overflow correctly, the following piece of code is an example of detecting overflow using gcc (since version 5) or clang (at least 3.8) for multiplication:

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

int main(void)
{
  size_t a, b, c;
  // we assume sizeof (size_t) == 8
  // if you're using a 32bit arch, adapt this code ...
  a = 1ul << 32; // a = 2^32
  b = 1ul << 32; // b = 2^32
  // a * b is 2^64 which doesn't fit in size_t
  if (__builtin_mul_overflow(a, b, &c)) {
    printf("Overflow detected\n");
  } else {
    printf("Success: %zu\n", c);
  }
  return 0;
}
```

## `realloc`

In order to implement `realloc(3)`, follow this strategy:

- Get the pointer to the chunk
- Check if you can satisfy the new size without another allocation
- Allocate a new block
- Copy the data from the old block to the new one

- Free the old block

When doing the copy, you must be sure that you use the smallest size between the old and the new block. And of course, you must manage the error case.

## Extensions

Once everything works, you can extend your `malloc(3)` with the following features:

- **Merge**: allow `free` to merge the freed block with the next and/or previous chunk if they are free
- **Split**: allow `malloc` to split the founded chunk if it is too big
- **Give back memory**: allow `free` to move the break when freeing the last chunk

You can probably also use the merge operation in `realloc`.

## Building a library

In order to correctly use your malloc, you should build a dynamic library. I suppose that you have put all your functions in a file called `malloc.c`.

First, you must check that the only visible functions are `malloc`, `free`, `calloc` and `realloc` (mark all other functions with the keyword `static`).

The following `Makefile` will help you build the file `libmalloc.so` (the dynamic library):

```
# Makefile for malloc

CC=gcc
CFLAGS= -Wall -Wextra -std=c99 -fno-builtin -O0 -g -fPIC
LDFLAGS=
LDLIBS=

libmalloc.so: malloc.o
        ${CC} -shared -o libmalloc.so malloc.o

clean:
        rm -f *~ *.o
        rm -f libmalloc.so

# end
```

## Testing

You should test your `malloc` in 2 steps:

1. Test all your functions one by one with your own code (try to allocate several block, free them, realloc them ... )
2. Replace the libC's malloc with your using `LD_PRELOAD`

In order to replace the libC's malloc, you should first build a dynamic library. If you want to test your code against a real program like `ls`, you must have all the functions. Then, here is an example with `ls`:

```
> ls
Makefile  malloc.c  test.c
> make libmalloc.so
gcc -Wall -Wextra -std=c99 -fno-builtin -O0 -g -fPIC   -c -o malloc.o malloc.c
gcc -shared -o libmalloc.so malloc.o
> ls
Makefile  libmalloc.so  malloc.c  malloc.o  test.c
> LD_PRELOAD=./libmalloc.so /bin/ls
Makefile  libmalloc.so  malloc.c  malloc.o  test.c
```

If things fails, you can use `gdb`:

```
> gdb -q /bin/ls
```

```
(gdb) set exec-wrapper env 'LD_PRELOAD=./libmalloc.so'
(gdb) run
Starting program: /usr/bin/ls
warning: Could not load shared library symbols for linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?
Makefile  libmalloc.so  malloc.c  malloc.o  test.c
[Inferior 1 (process 29583) exited normally]
(gdb)
```

Some remarks:

- You really need all functions
- Be sure that `malloc(0)` returns a valid pointer that can be passed to `free` (while you can return `NULL`, it's not a good idea.)
- Be sure that `realloc(NULL, s)` is equivalent to `malloc(s)`
- Be sure that `realloc(p, 0)` returns `NULL` and free `p`