# My Malloc Library

## Learning Objectives

Upon completion of this assignment, you should be able:

1. Manipulate C pointers to traverse a process' address space
2. Use pointer arithmetic to adjust pointer references
3. Use casting to dereference memory storage as different types
4. Manually adjust the process heap

New mechanisms you will see and use include:

- C: enum, type casting, pointer arithmetic, fprintf(), stdout, stderr
- system calls: sbrk()

## Function Specifications

NAME

```
my_malloc(), my_free(),coalesce_freelist(), free_list_begin()
```

SYNOPSIS

```
#include "my_malloc.h"

void * my_malloc(size_t size);
void my_free(void *ptr);

void coalesce_free_list(void);
FreeListNode * free_list_begin( void );

typedef struct freelistnode {
    struct freelistnode *flink;
    size_t size;
} * FreeListNode;
```

DESCRIPTION

my_malloc()

allocates `size` bytes of memory

my_free()

deallocates memory allocation pointed to by `ptr`, previously allocated by my_malloc().

coalesce_free_list()

merges adjacent chunks on the free list into single larger chunks.

free_list_begin()

retrieves the first node of the free list

RETURN VALUES AND ERRORS

On success, `my_malloc()` returns an 8-byte aligned pointer to the allocated memory.
On failure, `my_malloc()` returns `NULL` and sets `myerrno` to `MYENOMEM`.

`my_free()` returns nothing.
On failure, when called with a non-malloc'd pointer, `my_free()` sets `myerrno` to `MYEBADFREEPTR`,

`free_list_begin()` returns a pointer to the first free list node or NULL if the free list is empty.

## Implementation Details

### Memory Allocation

We refer to the entire memory block of memory used to satisfy an allocation request as the "chunk" – the chunk will be bigger than the extent of memory we expect the user to access. The minimum chunk size should be the size of the struct freelistnode.

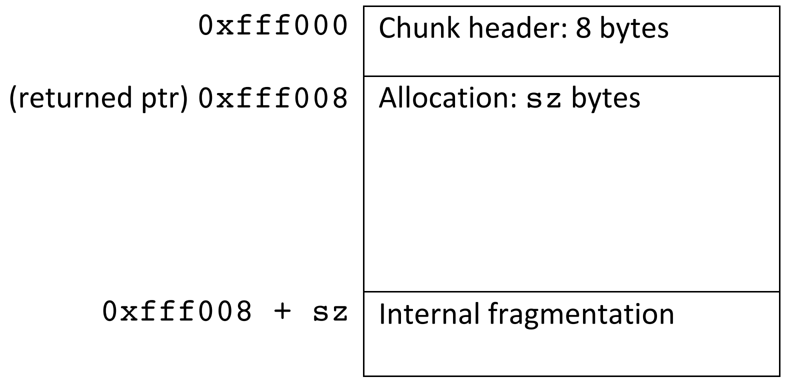
**Chunks**

Allocated chunks returned `my_malloc()` will be inflated by:

1. chunk header: 8 bytes
2. internal fragmentation:
    a. any padding necessary to make the chunk size a multiple of 8
    b. small wastage (when splitting a chunk would result in a chunk smaller than the minimum chunk size)

**Chunk header**

Use the 8 bytes just before the address returned by `my_malloc()` for your bookkeeping chunk header. Use the first four bytes for the total size of the chunk (including bookkeeping and padding bytes) and the second 4-bytes to store a hash or checksum used to determine that the chunk was allocated by `my_malloc()`.



**An Example Allocated Chunk**

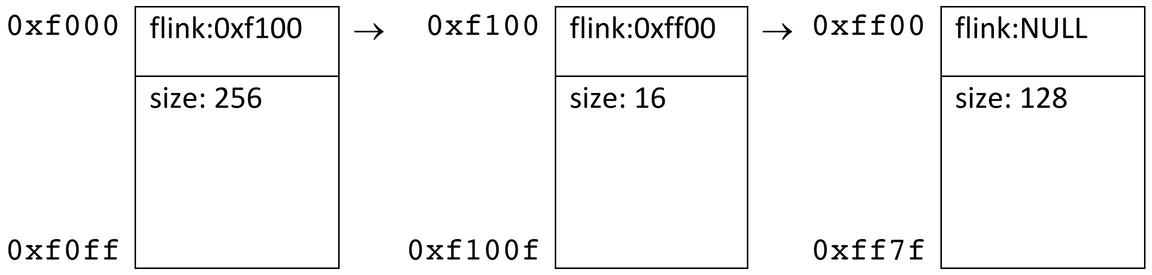
**Allocating a chunk**
my_malloc() first searches the free list for a usable chunk. If no usable chunk is found, call `sbrk()`[1] to extend the heap segment. `my_malloc()` returns a pointer referencing 8 bytes into the chunk, i.e. after the chunk header.

## Memory Deallocation
`my_free()` places freed or deallocated chunks of memory onto a free list. Calls to `my_free()` do not coalesce adjacent memory chunks.

## Free List Management
Use `struct freelistnode` in `my_malloc.h` to implement a singly-linked free list to manage free chunks. To add a chunk to the free list, inline a `struct freelistnode` using the very same memory chunk[2]. If `ptr` is the address of the chunk, this can be done by:

```
    FreeListNode node;
    node = (FreeListNode)ptr;
```

Then properly set `node->size` and `node->flink` and insert `node` into the free list. `flink` should be NULL for the last node in the list.



**Example Free List**

## Requirements and Constraints
1. Your program may have a single global variable for a pointer to the first free list node.
2. Generally, you should extend the heap by 8KiB (8,192 bytes) at a time, however if `my_malloc()` is called with a size greater than 8,192, call `sbrk()` with the minimum size to accommodate the needed chunk. You may not make calls to `sbrk()` with less than 8,192, except `sbrk(0)` to identify the heap's current end.
3. You should assume that other library routines may also make calls to sbrk().
4. Besides sbrk(), you may not use **any** other library or system calls.
5. You may not allocate more than 8 bookkeeping bytes.
6. Your free list should always be sorted in ascending order by chunk address.
7. Use a *first fit* strategy to search the free list, i.e. return the first usable chunk found.

[1] For this exercise, we use the simpler yet deprecated sbrk() not the more complex, POSIX-compliant mmap.
[2] This is why minimum chunk size must be the size of struct freelistnode.

8. Unless the result would be a chunk smaller than the *minimum chunk size*, oversized chunks (from the free list or the heap) must be split in two. The remainder chunk should be added to the free list.
9. Check the return value of `sbrk()` to ensure it properly extended the heap.

## Submission

FOLLOW THESE INSTRUCTIONS PRECISELY

Requisite files:

- Sources: `my_malloc.c` and any auxiliary files needed to implement the functions in `my_malloc.h`
- README: you may submit an optional README file with comments, feedback, known issues, etc.

Your submission must use the following naming convention: firstinitiallastname_lab? where firstinitial is the initial of your first name, lastname is your last name, and '?' is the number of this lab [0-5]. For example, the Lab 3 directory for Candace Parker would be 'cparker_lab3'.

Place the requisite files in your submission directory and execute the command:

```
    tar -czf labdir.tgz labdir
```

where labdir is your submission directory. This will create a new file labdir.tgz containing the contents of labdir. You can verify the contents of this *compressed tar file* using:

```
    tar -tzf labdir.tgz
```

Submit your assignment via Canvas.

## my_malloc.h

```c
//the size of the header for heap allocated memory chunks
#define CHUNKHEADERSIZE 8

//error signaling
typedef enum {MYNOERROR, MYENOMEM, MYBADFREEPTR } MyErrorNo;
MyErrorNo myerrno=MYNOERROR;

//my_malloc: returns a pointer to a chunk of heap allocated memory
void *my_malloc(size_t size);

//my_free: reclaims the previously allocated chunk referenced by ptr
void my_free(void *ptr);

//struct freelistnode: node for linked list of 'free' chunks
typedef struct freelistnode {
    struct freelistnode *flink; //pointer to next free chunk node
    size_t size; //size of current chunk
} * FreeListNode;

//free_list_begin(): returns pointer to first chunk in free list
FreeListNode free_list_begin(void);

//coalesce_free_list(): merge adjacent chunks on the free list
void coalesce_free_list(void);
```