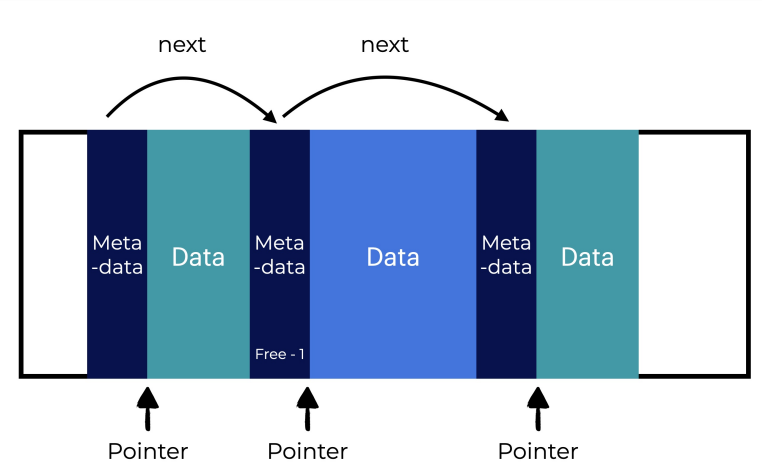
Introduction

In this section, we'll explore a simple implementation of the `malloc()` and `free()` functions used for dynamic memory allocation and deallocation.

Representing Blocks

We will need a small block at the beginning of each chunk that contains extra information called **meta-data**. This block holds, at the very least:
* A pointer to the next chunk
* A flag to mark free chunks, and
* The size of the data of the chunk.

This block of information comes before the pointer returned by `malloc`



.guides/img/impMalloc1

The graphic above shows an example of heap organization with the meta-data in front of the allocated block. Each chunk of data is made of a block of metadata followed by the block of data. The pointer returned by `malloc` is shown in the lower part of the graphic. Notice that it points on the data block, not on the complete chunk.

The metadata (data about data) is stored in a block of memory next to the specific block of allocated memory to which it belongs. This metadata block, or more specifically the linked list structure is essential because we need to know whether a given block is free or already allocated, as well as the size of the block to which it refers.

The following is the metadata block structure:

```
struct block{
    size_t size;
    int free;
    struct block *next;
};
```

The Header File

A header file is required for this implementation. The file has to be saved with the `.h` extension. Notice the extension in the open file, `mymalloc.h`.

Include the following headers in your header file.

```
#incude <stdio.h>
#include <stddef.h>
```

`<stddef.h>` is required because the definition for the `size_t` datatype is found here.

Say we have a block of memory with a size of $20,000$ bytes `char memory[20000];` and that all of the data structures and the allocated memory blocks are kept in this same chunk of memory.

Let's declare the array that we will consider as our memory. We will get a **contiguous allocation of memory** by using an array.

```
char memory[20000];
```

Now, let's initialize a pointer of type `block`, named `freeList`. This points to the starting address of the chunk of memory we created before. This `freeList` pointer will point to the start of the linked list of metadata blocks. The starting address of the array (memory) should be cast to type `void` so that we can allocate blocks of memory that are of different datatypes (int, char, float, etc.).

```
/* The structure definition to contain metadata of each block
allocated or deallocated*/
struct block{
    size_t size;  // carries the size of the block described by
it
    int free;  /* This flag is used to know whether the block
described by the metadata is free or not --> if it is free, this
is set to 1, otherwise, it is 0*/
    struct block *next  // points to the next metadata block
};

struct block *freeList = (void*)memory;
```

In the following sections, we will continue by defining the following functions in a source file:

```
/*The function definitions which are defined in the next source
file malloc.c*/
void initialize();
void split(struct block *fitting_slot,size_t size);
void *MyMalloc(size_t noOfBytes);
void merge();

void MyFree(void* ptr);
```

Initialize

We'll use a source file for our implementation of the `malloc` and `free` functions. This will have a regular `.c` extension. Notice the extension on the open file, `mymalloc.c`

Let's start by including the necessary headers. We'll also include the header file we created earlier.

```
#include<stdio.h>
#include<stddef.h>
#include "mymalloc.h" /*Here we include the header file given
above*/
```

Now, let's initialize the first metadata block and update it to refer to the next block of memory.

```
/*Initializing the block of memory*/
void initialize(){
    freeList->size=20000-sizeof(struct block);
    freeList->free=1;
    freeList->next=NULL;
}
```

- The block it refers to is 20000 bytes in size (the size of one metadata block).
- We set the free flag to 1 to indicate that the block has not yet been allocated.
- And the first metadata block does not yet have a next metadata block, so we set next to `NULL`.

Split

To allocate memory, we use the **First-fit-algorithm** to find a free block. Assume we get a request to allocate a 500-byte memory block. Starting with the first metadata block, we can search for the first block that has enough space to allocate.

We will allocate the block of size 750 if the free block sizes are 250, 130, and 750 in that order.

```
/*Making way for a new block allocation by splitting a free
block -- (Assume first fit algorithm)*/
void split(struct block *fitting_slot,size_t size){
    struct block *new=(void*)
((void*)fitting_slot+size+sizeof(struct block));
    new->size=(fitting_slot->size)-size-sizeof(struct block);
    new->free=1;
    new->next=fitting_slot->next;
    fitting_slot->size=size;
    fitting_slot->free=0;
    fitting_slot->next=new;
}
```

- If we find a free block that exactly fits the required size, we don't need to split it. This function is only required when we have more than we need.
- It accepts the following arguments: — A pointer to the metadata block which refers to the memory block that is larger than necessary (fitting_slot) and the desired size of the memory chunk.
- We create a new metadata block pointer called "new" here. It should point to the location provided by passing(setting aside) a block of memory which is equal to **the_size_of_the_metadata_block_we_considered** + **the_size_requested_to_be_allocated**
- The new pointer points to the next free chunk in the metadata block. In the code, you can see that both the new and fitting_slot metadata blocks have their attributes set accordingly.

Malloc

Let's build our `malloc` function called `myMalloc`. The first thing we need is metadata block pointers to traverse through the freeList.

```
void *MyMalloc(size_t noOfBytes){
    struct block *curr,*prev;
```

Then we need a result pointer to return the starting address of the allocated chunk of memory.

```
void *result;
```

We'll create a condition that will initialize the memory if it has not already been initialized. This series of statements will be executes only if the size of the first metadata block hasn't been set yet, meaning if the memory hasn't been initialized.

```
if(!(freeList->size)){
    initialize();
    printf("Memory initialized\n");
 }
```

Now, let's make the temporary pointer `curr` to point to the start of the metadata block list.

```
curr = freeList;
```

If the following condition is met,`while((((curr->size)<noOfBytes)||((curr->free)==0))&&(curr->next!=NULL))`, the metadata block we checked cannot be used for the allocation. So we run the following statements and go on checking one metadata block at a time.

```
 while((((curr->size)<noOfBytes)||((curr->free)==0))&&(curr->next!=NULL)){
      prev=curr;
      curr=curr->next;
      printf("One block checked\n");
 }
```

If this condition is met, `if((curr->size)==noOfBytes)`, the metadata block we checked refers to a chunk of memory that exactly fits the required size. So set the free flag to 0, indicating that it is allocated. Then return the starting address of the block of allocated memory.

```
if((curr->size)==noOfBytes){
    curr->free=0;
    result=(void*)(++curr);
    printf("Exact fitting block allocated\n");
    return result;
 }
```

If this condition is met, `else if((curr->size)>(noOfBytes+sizeof(struct block)))`, the metadata block we checked refers to a chunk of memory that is of size greater than what is required. So call the `split()` function to allocate only the block which is required and then return the starting address of the block of allocated memory.

```
else if((curr->size)>(noOfBytes+sizeof(struct block))){
    split(curr,noOfBytes);
    result=(void*)(++curr);
    printf("Fitting block allocated with a split\n");
    return result;
 }
```

Else, any other outcome means that there is not sufficient memory to allocate, so you should return a `NULL` pointer.

```
 else{
    result=NULL;
    printf("Sorry. No sufficient memory to allocate\n");
    return result;
    }
}
```

Merge

It is possible to have consecutive blocks that are released by deallocating after they have previously been allocated. This creates external fragmentation, causing the `MyMalloc()` function to return a NULL pointer despite the fact that we have enough memory to allocate. We use the `merge()` function to combine the consecutive free blocks by removing the metadata blocks in between.

```
/*This is to merge the consecutive free blocks by removing the
metadata block in the middle. This will save space.*/
void merge(){
    struct block *curr,*prev;
    curr=freeList;
    while((curr && curr->next)!=NULL){
        if((curr->free) && (curr->next->free)){
            curr->size+=(curr->next->size)+sizeof(struct block);
            curr->next=curr->next->next;
        }
        prev=curr;
        curr=curr->next;
    }
}
```

Free

After we've allocated memory for something, it's good practice to deallocate that memory when we finish using it so it can be used by something else. We'll use our `myFree()` function for this.

It will take the pointer to a block of memory previously allocated as a parameter.

```
/*Function MyFree(free)*/
void MyFree(void* ptr){
```

Here, we'll verify that the address passed as an argument to the function is inside the address range of the memory array we used, `if(((void*)memory<=ptr)&&(ptr<=(void*)(memory+20000)))`. If it is, we simply change the free flag in the metadata block to 1 to indicate that it is free, and then search through and merge any free blocks that we find.

```
    if(((void*)memory<=ptr)&&(ptr<=(void*)(memory+20000))){
        struct block* curr=ptr;
        --curr;
        curr->free=1;
        merge();
    }
    else printf("Please provide a valid pointer allocated by
MyMalloc\n");
}
```

If a valid pointer is not provided, the above message will be printed.

Main.c: Running the Implementation

Now let's see `MyMalloc()` and `MyFree()` in action! The file to the left, `Main.c` implements our homemade functions. Compile this program using the command below in the terminal.

```
gcc Main.c -o Main
```

Then run the program:

```
./Main
```

We should now know how to successfully implement `malloc()` and `free()`.