CSCI3150 Assignment 2 - Dynamic Memory Allocation

Abstract

The library function calls `malloc()` and `free()` have started puzzling you from the first day you write programs with dynamic-allocated memory. The two calls work in-pair - providing you both **allocated memory** and **segmentation faults**. In this assignment, you are going to implement the two function calls.

Contents

1 Assignment at a glance

The library functions `malloc()` and `free()` give you the convenience in controlling and using the **heap memory**. However, the `malloc()` and the `free()` are doing tons of work in order to give you such convenience.

The job of the `malloc()` call is **memory allocation** while the job of the `free()` call is **memory deallocation**. But, the story is not that simple because there are constraints in using the heap memory.

What are the things that you are supposed to learn from this assignment?

- **Hard Skills**.
  - Appreciating the great work done by the authors of the `malloc()` and the `free()` library function calls.
  - Understanding the internal workings of the `malloc()` and the `free()` library function calls.
  - Utilizing, being puzzled, and finally mastering the skill in using pointers correctly, efficiently, and (hopefully) **elegantly**.
  - Understanding **the power of casting** over pointers.
  - Removing the fear in implementing linked lists.
- **Soft Skills**.
  - The real difficulty of this assignment is to come up with useful, tricky testcases. I guarantee you that this assignment needs at most 500 lines of codes in order to finish all tasks. Advanced students may only need to write fewer than 300 lines. But, you have to learn how to test your implementations! The specification already contains hints in designing simple testcases.

    We encourage **peer learning** and we encourage you to test your codes among yourself. E.g., Bob's `malloc()` implementation pairs with Alice's `free()` implementation. This can expose and remove potential bugs in a very fast manner.

Nevertheless, we **never encourage you to copy the stuffs you need from others' codes**. You should make use of the object codes, i.e., "`.o`" files, and header files, i.e., "`.h`" files, in order to set up the peer debugging process.

– The assignment is naturally divided into independent but similar tasks. How can you write structured functions that can make your implementation compact? As a matter of fact, well-structured code is one of the keys to avoid stupid bugs.

1.1 Vocabularies

Some words and phrases that we will use in this specification may look awkward to you. The following table contains a set of vocabularies that are frequently used in this specification.

User	It means the application which invokes `malloc()` or `free()`. The relationship is shown in Figure 1.
Heap memory	It means a part of the process' memory that stores dynamically-allocated memory.
Allocated memory block	It means a finite memory zone on the heap memory that is requested by the user using the `malloc()` call.
Free memory block	It means a finite memory zone on the heap memory that the user gives up using, by the `free()` call. Note that this memory zone is still owned by the process.
Free list	It means the list of free memory blocks that are organized using a linked list implementation.
Hole	It means a free memory block in the heap memory.

1.2 Restrictions and assumptions

Figure 1 has introduced you the task that you have to finish:

Implementation Requirement #1.

You have to write a piece of C code that provides the implementations of the `malloc()` and the `free()` library calls.

In other words, you will be submitting a code **without** the `main()` function. The code that you will submit should contain the implementation of the `malloc()` call as well as the `free()` call. Nevertheless (and we will discuss later), you need to implement several versions

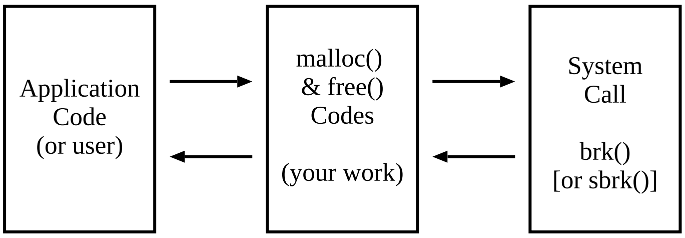

Figure 1: The relationship among application code, `malloc()` code, `free()` code, and the system call.

of the `malloc()` call as well as the `free()` call. Therefore, the calls will be renamed. For the ease of the discussion, we still use the names `malloc()` and `free()`, but, please be aware that, we will give up using the original implementations of the `malloc()` and the `free()` calls.

1.2.1 Assumptions

The `main()` function will then be stored in the application code, and such a function may not be implemented by you. Let us assume that the programmers of the application code would never commit the following errors:

- The application may provide zero or negative values to the `malloc()` call.
- The application may deallocate a pointer twice using the `free()` function implemented by you.
- The application may not provide a correct pointer value, e.g., in the middle of an allocated memory zone or even `NULL` address, to the `free()` function.

However, except the above cases, you cannot assume how the programmers use your implementation. For example, a programmer may invoke your `malloc()` call by 1,000,000 times and then invoke the `free()` call for 1,000,000 times to deallocate all the allocated memory.

1.2.2 Restrictions

Language. You must implement the assignment in either C or C++. Using languages other than the above two would have zero marks.

Environment. You must write and test your programs under a 32-bit Linux operating system. To be specific, our demonstration environment is the 32-bit Ubuntu 10.04 running as a virtual machine.

1.3 Heap memory layout

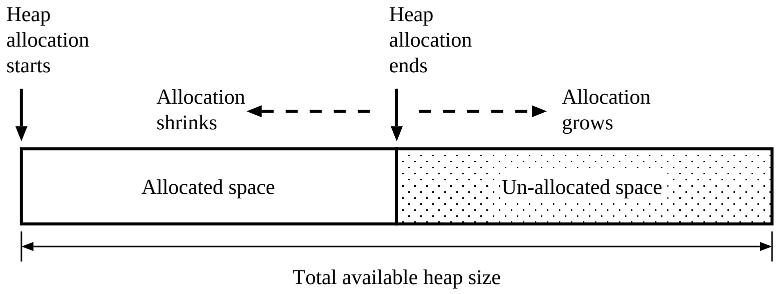

Figure 2: Heap allocation illustration.

You can treat the heap memory as a piece of contiguous memory, and it is shown in Figure 2. Its limit is bounded by the total physical memory plus the amount of the virtual memory. In simple words, calling `malloc()` means to claim memory from the available heap memory space, and calling `free()` is doing the reverse thing.

Nevertheless, there is an inconvenient limit on the way you can use the heap memory: you can only grow or shrink the contiguous space, and, after such changes, the space is still required to be contiguous. In other words,

- you cannot deallocate a piece of memory, which is in **the middle of the allocated space** nor **at the start of the allocated space**;
- you can only deallocate a piece of memory, which is **at the end of the allocated space**;

1.4 The `sbrk()` library call

To control the size of the allocated heap space, the `brk()` system call is used. However, it is not a programmer-friendly system call. Instead, we recommend you to use the `sbrk() library call`.

```
#include <unistd.h>
void * sbrk(int increment);
```

- **Purpose**. Change the size of the allocated heap space. This library call eventually invokes the `brk()` system call.
- **Parameter and return value**.
  - If “`increment > 0`”, then
    - the allocated heap space will be increased by “`increment`” bytes, and
    - the return value is the starting address of the newly allocated area.
  - If “`increment < 0`”, then
    - the allocated heap space will be decreased by “`increment`” bytes, and
    - the return value is the address of the end of the allocated heap space.
  - If “`increment == 0`”, then
    - no change will be made on the allocated heap space, and
    - the return value is the end of the allocated heap space.

The `sbrk()` call is just the basic building block. The more important thing is: “**how to manage the allocated heap space?**”

1.5 The memory allocation and deallocation strategies

When an user program calls `malloc()`, a natural but important question should be asked: *Should we call* `sbrk()` *to allocate more memory*? The follow-up questions include:

- If yes, how much memory is needed?
- If no, which address should be returned to the user?

In addition, when an user program calls `free()`, *should we call* `sbrk()` *to deallocate memory*?

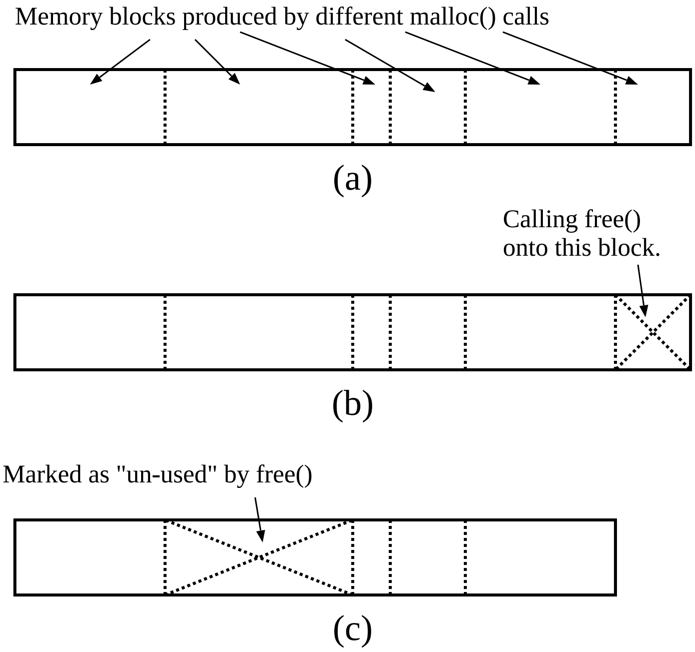
Memory blocks produced by different malloc() calls
(a)
Calling free()
onto this block.
(b)
Marked as "un-used" by free()
(c)

Figure 3: Illustration of different scenarios encountered by `free()`. (a) The allocated heap space before calling `free()` is shown. (b) The heap space may be shortened when `free()` is instructed to "free" the last block. (c) No actual deallocation should take place when `free()` targets a block which is in the middle (or the start) of the allocated heap space.

1.5.1 Deallocation strategies

First of all, we tackle the above questions by looking at deallocation. Let us consider the scenario that you are tackling the first invocation of the `free()` call.

To call or not to call the `sbrk()` system call? Because of the special property of the heap memory allocation, we have to be very careful in answering this question.

- If the parameter of the `free()` call points to the **last block of the allocated memory**, then there is no harm to reduce the allocated heap space.
- Else, i.e., the target piece of memory is in the middle of the allocated heap space, we must not reduce the allocated heap space.

Thus, the above requirement produces a strange memory layout as shown in Figure 3. Because of the above reasons, we need a special management data structure that can memorize and organize the memory blocks that are abandoned by the application.

Nevertheless, the abandoned (or unused) memory blocks are still owned by the processes: if they are handled mistakenly, the unused memory blocks will stay with the process and may not be able to be removed nor to be reused until the process terminates. We have a technical name for this kind of unattended memory, the **memory leakage**.

> **Implementation Requirement #2.**
> There should be no memory leakage caused by the `malloc()` and the `free()` calls. Memory leakage can only happen when the application code is implemented carelessly.

In the next section, we will talk about how the free blocks are managed and organized.

1.5.2 Allocation strategies

An allocation strategy determines the way that the `malloc()` call makes use of the holes within the allocated heap space. It may end up with different strategies, for example:

- ignoring all the holes, i.e., invoking `sbrk()` under any circumstances.
- using the **first intermediate hole** that fits the space requirement, i.e., without invoking `sbrk()`. If there are no such holes, then the `sbrk()` call is invoked.
- using the **largest intermediate hole** that fits the space requirement, i.e., without invoking `sbrk()`. If there are no such holes, then the `sbrk()` call is invoked.
- using the **smallest intermediate hole** that fits the space requirement, i.e., without invoking `sbrk()`. If there are no such holes, then the `sbrk()` call is invoked.

We will discuss the above strategies in Section 3 on page 20.

2 Functions Implementations

To solve the problems described in the previous section, we propose a way to embed information into the allocated heap space. This proposal will affect the implementation of the `malloc()` and the `free()` function calls.

2.1 Memory management structure

We add a **memory management structure** to the allocated heap space. Figure 4 shows the codes of the management structure. Our target is to maintain **a list of free memory block** using the management structure as shown in Figure 5.

```
typedef struct header {
    struct header *ptr;    /* pointer to the next free block */
    unsigned int size;     /* size of this block */
} Header;

Header base;        /* A list of free block, initially empty. */
```

Figure 4: Code snippet that defines the heap management structure.

2.1.1 Header data type

The `Header` data type is the implementation of a linked list. It is defined as a structure of two variables: “`ptr`” and “`size`”.

- “`struct header *ptr`” is a pointer pointing to another “`Header`” structure.
- “`unsigned int size`” is the size of the “`Header`” plus the size of the memory required by the `malloc()` call. But, why not just stating “`size`” to be the size required by the `malloc()` call? It is just a matter of **programming convenience**.

Implementation Requirement #3.

The “`size`” variable in the “`Header` structure must store the value: $n + h$.

where n is the requested size of the `malloc()` call and h is the size of the “`Header`” structure.

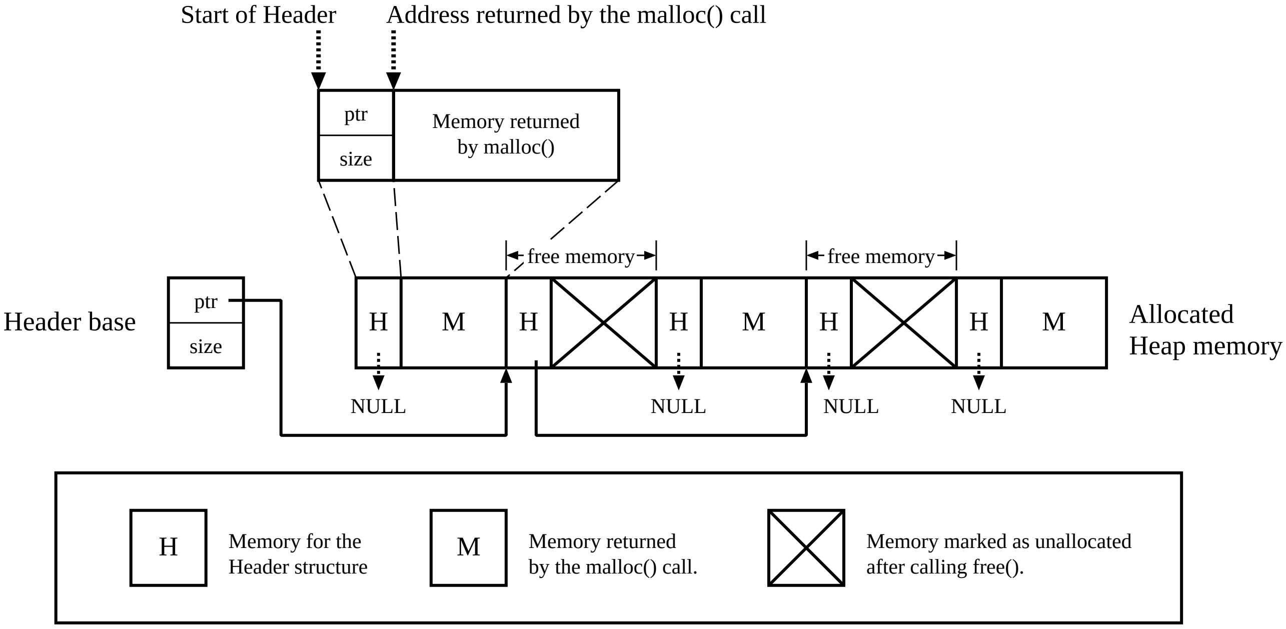
Start of Header
Address returned by the malloc() call
ptr
size
Memory returned by malloc()
free memory
free memory
Header base
ptr
size
H
M
H
H
M
H
H
M
Allocated Heap memory
NULL
NULL
NULL
NULL
H
Memory for the Header structure
M
Memory returned by the malloc() call.
Memory marked as unallocated after calling free().

Figure 5: The illustration of the allocated heap space with the header structure overlaid.

2.1.2 The free list and the "`base`" variable

A linked list is constructed and it starts with a global variable named "`base`". More importantly, according to Figure 5, this linked list represents the list of unused memory block. However, "*why do we need such a global variable?*

1. There may be no unused memory blocks.

   Then, the `base` variable indicates that an empty list by setting the `ptr` field to `NULL`. Plus, this should be the initial value when a process starts.

2. It locates the first unused memory block.

   The `free()` call may mark memory blocks as unused at arbitrary locations. A variable is needed to indicate the start of the list and it is the purpose of having the `base` variable.

   By the way, the "*first unused memory block*" means the unused memory block with the smallest memory address.

Note that the `base` variable is a **global variable**. From now on, we name such a linked list of unused memory block the **free list**.

2.2 Free list operations

2.2.1 The first few `malloc()` calls

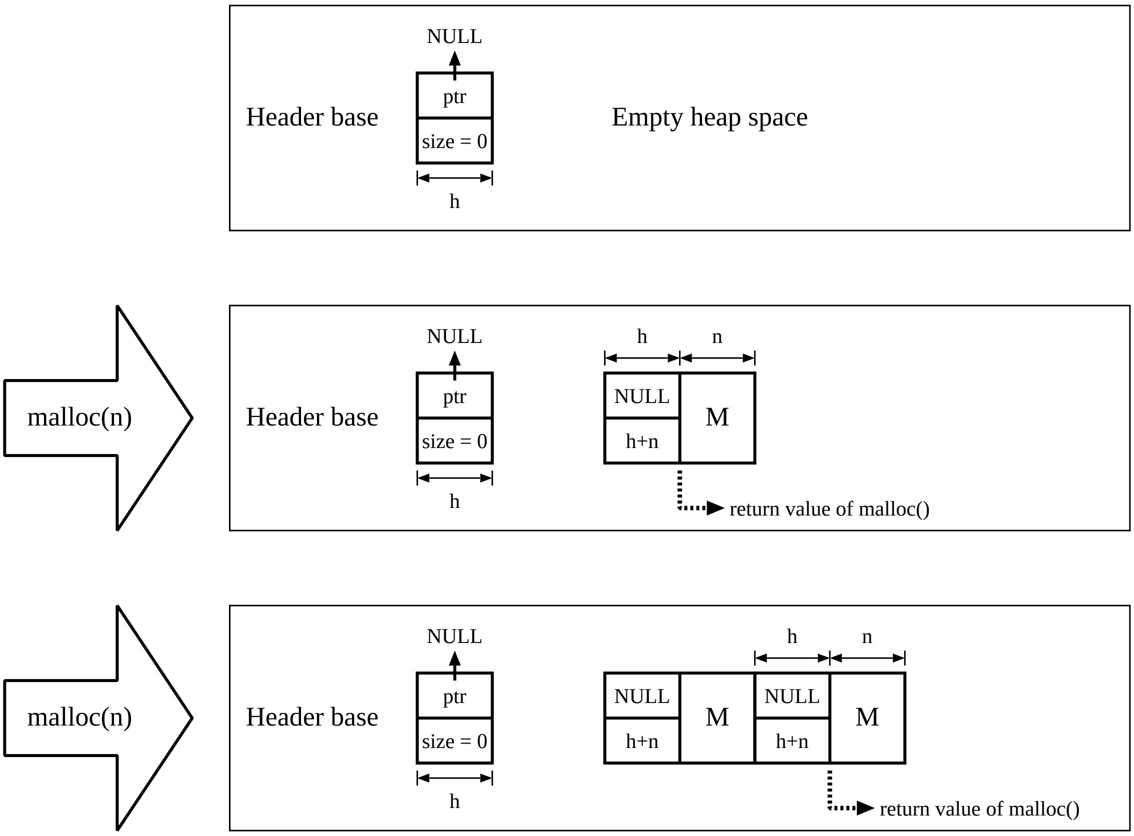

Figure 6: The illustration of the first `malloc()` call.

At the start of the program (not process) execution, the heap space is empty. The first `malloc()` call initializes the heap space using `sbrk()`.

Successive `malloc()` calls extend the allocated heap space. Nevertheless, the free list is still empty and the `ptr` field of the `base` variable is `NULL`

2.2.2 The first `free()` call

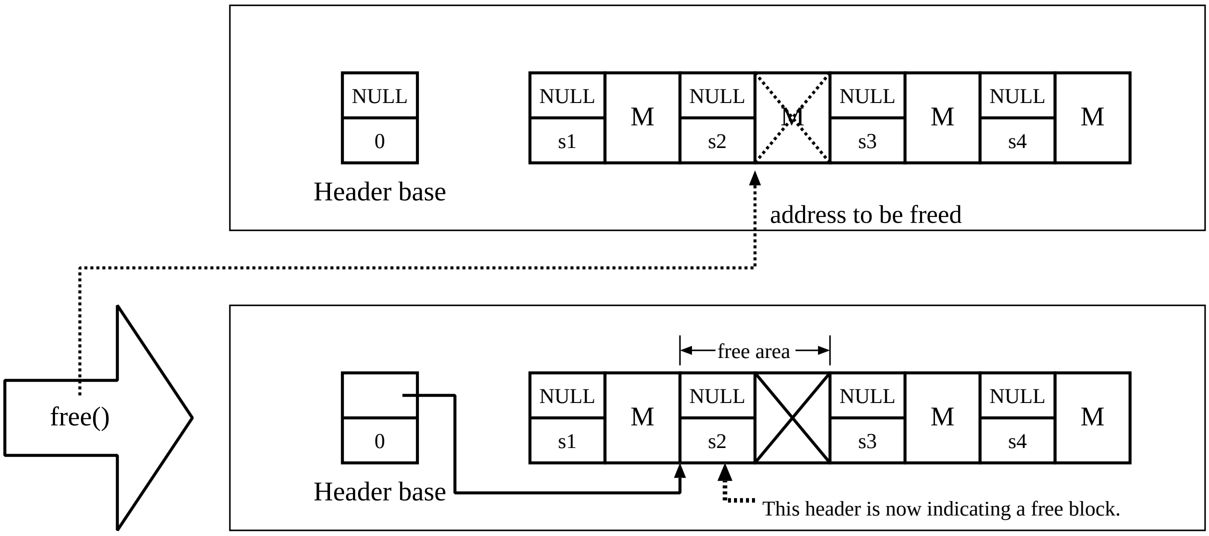

Figure 7: The illustration of the first few `free()` calls.

The first `free()` call creates the free list, and the free list is created by using the `Header` structure which is **right before** the address supplied to the `free()` call.

As you can see from Figure 7, the free list will be constructed. The `ptr` field of the `base` structure points to the start of the unused area.

Starting from this point, there will be many variations on the memory layout as well as on the free list management. The variations depend on the combinations and the execution order of the `malloc()` and the `free()` calls. Remember, we never test your implementation with erroneous combinations.

2.2.3 Successive `free()` calls

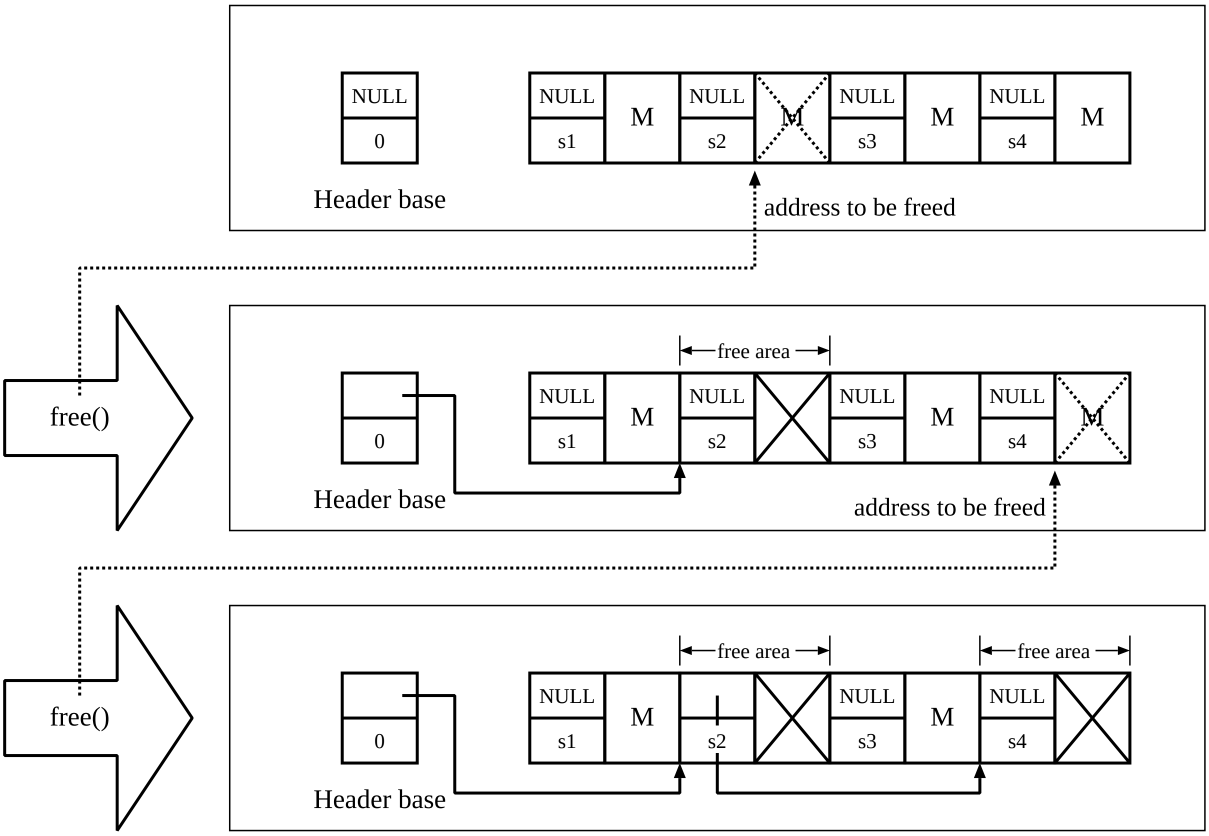

Figure 8: The illustration of two successive `free()` calls.

Successive `free()` calls extend the free list.

In Figure 8, you can observe that the second free block is at the **end of the allocated heap space**. Should we deallocate a memory block **depends on the deallocation strategy** and this will be discussed in Section 3 on page 20.

2.2.4 Successive and adjacent `free()` calls

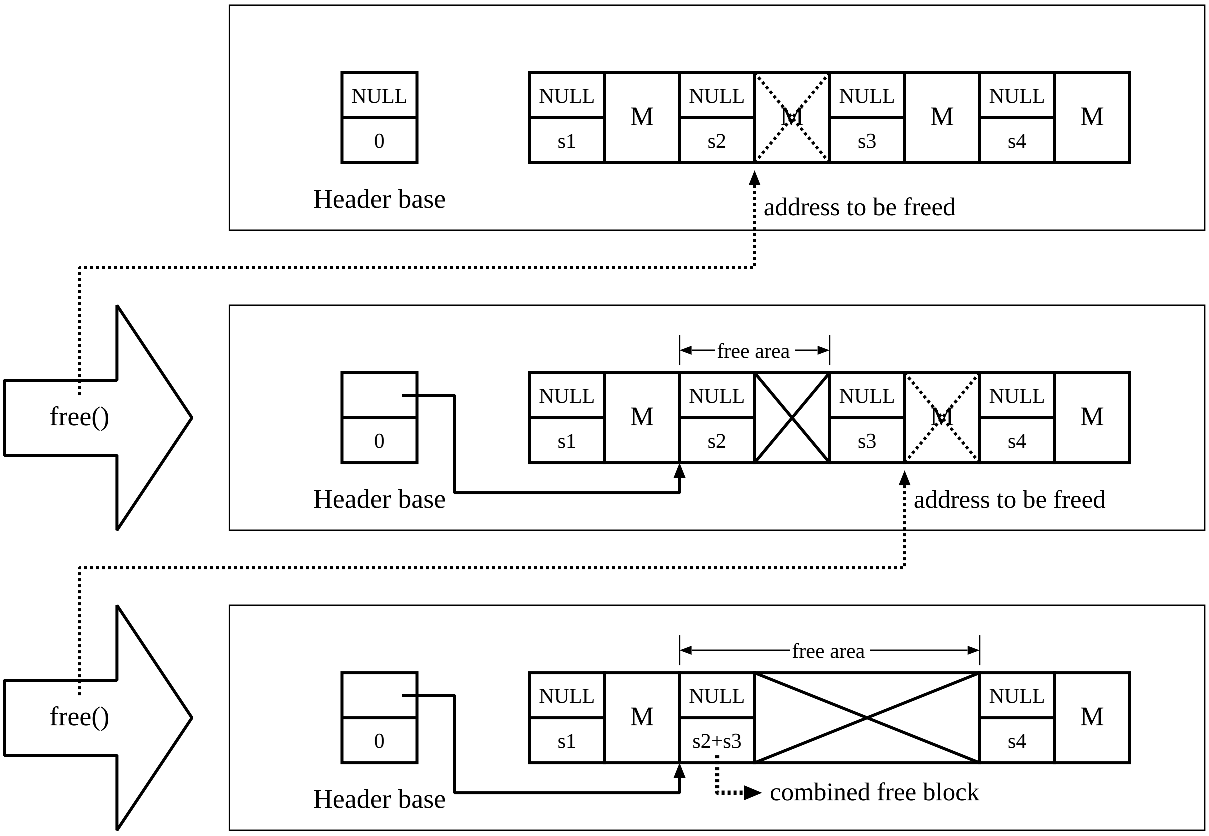

Figure 9: **Case 1 of 2**. The illustration of adjacent `free()` calls: two adjacent free blocks will be combined into 1.

When we say that "*two* `free()` *calls are adjacent*", it means that the two allocated blocks supplied to the two `free()` calls are adjacent, with one `Header` structure between the two blocks.

No matter the `free()` calls are dealing with adjacent blocks or not, the free list will become longer. Nevertheless, we add one requirement here:

Implementation Requirement #4

The two adjacent free blocks should be combined into one.

Figures 9 and 10 show the two cases that how two adjacent free blocks are combined.

- In Figure 9, the newly freed block is **right after a free block**, then the implementation is to **increase the size of the `Header` structure of the first free block**.

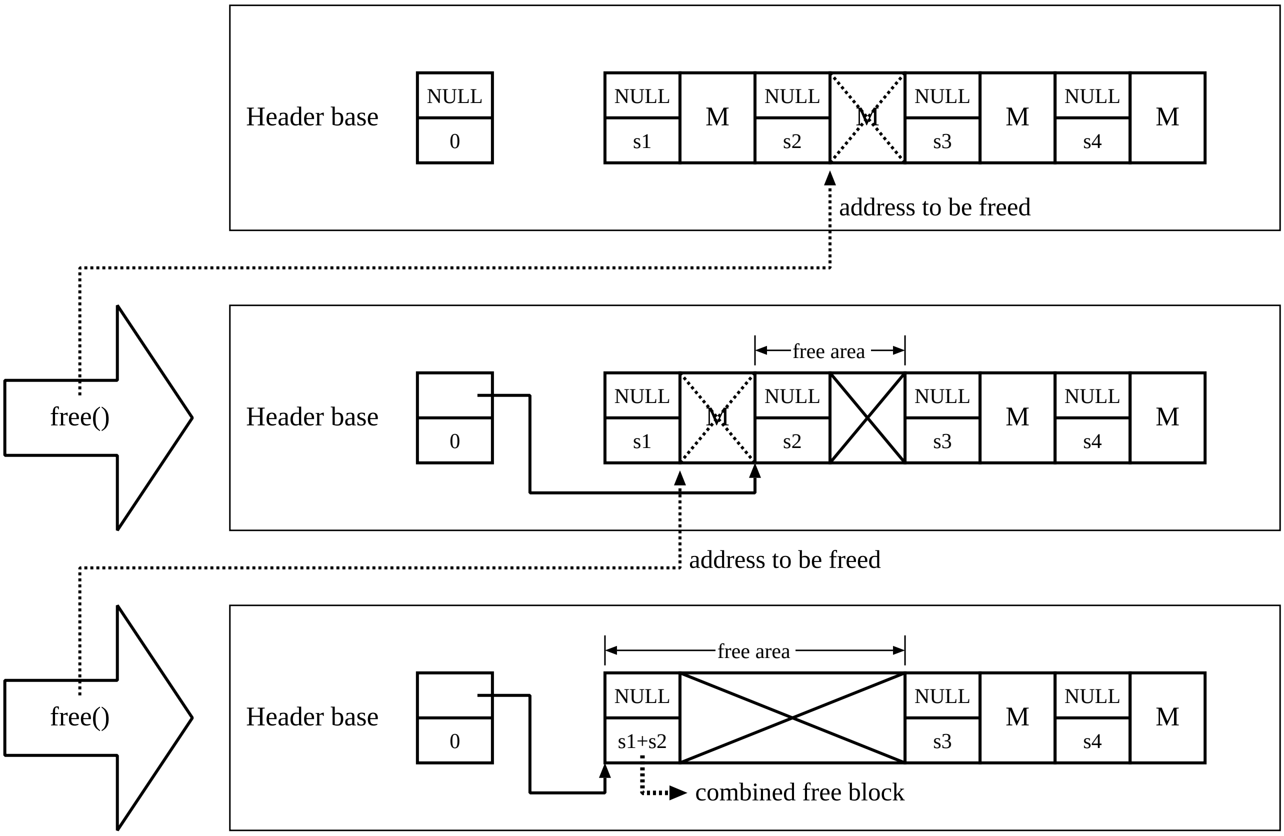

Figure 10: **Case 2 of 2**. The illustration of adjacent `free()` calls: two adjacent free blocks will be combined into 1.

- For the second case as shown in Figure 10, the newly freed block is **right before a free block**, then the implementation become a little bit complex.

  1. The "`base`" variable may need to be updated, if necessary.
  2. The `Header` of the new free block is changed.

By the way, you may ask "*why the adjacent free blocks are required to be combined?*" The reason is that: combining two free blocks can **give the application the chance for the allocation of a larger block** without calling `sbrk()`.

2.2.5 Invoking a `malloc()` call after `free()` call(s)

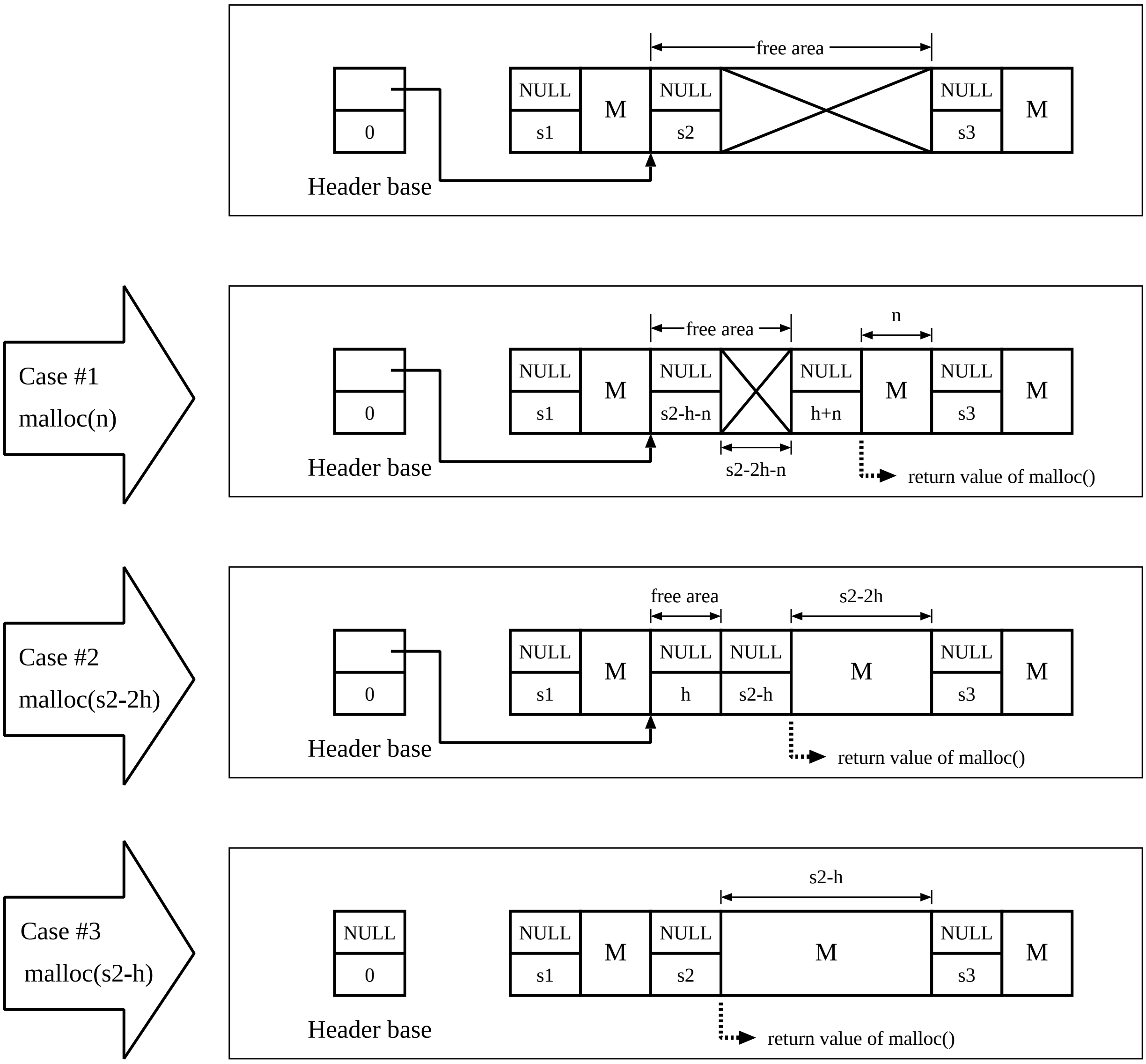

Figure 11: The illustration of invoking `malloc()` call after invoking `free()` call(s): different memory requirements result in different memory layouts.

If there exists a "*hole*", i.e., a free block, in the allocated heap space, then a `malloc()` call may be able to consume that hole. Note that this is related to the **allocation strategy** of the `malloc()` call and will be discussed in Section 3 on page 20.

For the examples shown in Figure 11, let us assume that the `malloc()` call will try to consume a hole that is large enough to meet the user's requirement. As shown in the figure, there is a hole of size `s2`. The memory size requested by the `malloc()` call is `n`.

- **Case #1**: $0 < n < s2 - 2h$, where h is the size of a `Header`.

  In this case, the `malloc()` call will not consume the entire hole.

  1. First, the `size` field of the free block is reduced from $s2$ to $s2 - h - n$.
  2. Then, a new `Header` structure is created and its size field is initialized to $n + h$.
  3. Last, the `malloc()` declares a successful operation by returning the pointer of the target piece of memory to the user.

  The remaining space inside the hole will still be able to serve other `malloc()` calls.

- **Case #2**: $n = s2 - 2h$.

  In this case, the scenario is similar to the previous one except that **there is no free space left**. As you can see from Figure 11, the memory layout becoming very interesting: the original free block header is updated to "`size = h`" and is adjacent to the new non-free block header.

- **Case #3**: $n = s2 - h$.

  In this case, the `malloc()` call will consume the hole completely. The "`base`" variable may require update. The original free block header becomes a non-free block header.

- **Case #4**: $n > s2 - h$.

  Of course, `sbrk()` is invoked because no hole in the free list was large enough to satisfy user's requirement.

By the way, **how about the case that $s2 - 2h < n < s2 - h$?** Please think about it by yourself. Nevertheless, the conclusion is that an implementation requirement of `malloc()` is then added:

Implementation Requirement #5.

No matter what the size n is, the `malloc()` should return a piece of memory of size:

$$(n - 1) + (h - ((n - 1) \ \% \ h))$$

where n is the size required by the `malloc()` call and h is the size of a `Header` structure.

If you cannot figure out what the above formula means, you can do some calculations by feeding different values of n. Then, you will find some interesting results.

2.2.6 Summary on free list operations

- A `malloc()` call
  - may traverse the entire free list;
  - may take one of the holes on the allocated heap space;
  - may call `sbrk()` to expand the heap.
- A `free()` call
  - may traverse the free list to look for adjacent free blocks.
  - may call `sbrk()` to shrink the heap;
  - must combine adjacent free blocks.
- A free list
  - must be a linked list over the allocated heap space;
  - must start with the global variable “`base`”, which is of the type `Header` structure;
  - must be there since the first “`malloc()`” call.

3 Memory allocation and deallocation strategy

In the previous section, we have described the basic management operations over the free list. In this section, we cover different strategies in managing the free list.

3.1 Allocation strategies

An allocation strategy means a way to implement the `malloc()` call and the differences among the strategies lie in the way that **the `malloc()` call takes up the holes inside the allocated heap space**.

3.1.1 No management

We are not joking: **"no management" is a way to manage**. This strategy is:

> **No-management strategy**
> For every `malloc()` call, "`sbrk()`" is invoked to allocate space for a new `Header` structure plus the piece of memory required by the user.

In other words, every `malloc()` call ignores the holes inside the allocated heap space.

3.1.2 First-fit strategy

> **First-fit strategy**
> For every `malloc()` call, the entire free list will be searched in order to find the **first** hole that fits the allocation requirement.
>
> If there is no hole satisfying the allocation requirement, the "`sbrk()`" call will be invoked. Else, the chosen hole will be allocated.

3.1.3 Best-fit strategy

> **Best-fit strategy**
> For every `malloc()` call, the entire free list will be searched to find the **smallest** hole that fits the allocation requirement.
>
> If there is no hole satisfying the allocation requirement, the "`sbrk()`" call will be invoked. Else, the chosen hole will be allocated.

3.1.4 Worst-fit strategy

Worst-fit strategy

For every `malloc()` call, the entire free list will be searched to find the **largest** hole that fits the allocation requirement.

If there is no hole satisfying the allocation requirement, the "`sbrk()`" call will be invoked. Else, the chosen hole will be allocated.

3.2 Deallocation strategies

A deallocation strategy means a way to implement the `free()` call and the differences in the strategies lie in the decision on whether the `free()` call should use `sbrk()` to **shorten** the heap space or not.

Remember, no matter which strategies, the `free()` call must implements the management of the free list and the combining process of two adjacent free blocks.

3.2.1 No management

No management

For every `free()` call, "`sbrk()`" should not be invoked.

3.2.2 Removing the last free block

No-tail management

For every `free()` call, if the target block is found to be the last block in the allocated heap space, then the `free()` call should invoke `sbrk()` to deallocate the memory block plus its corresponding `Header` structure.

Note that there is an interesting scenario under this strategy:

Implementation Requirement #6.

After the `free()` call has finished the deallocation of the target block, the `free()` call may require to continue the deallocation process if the current last block in the heap is again a free block.

4 Milestones

There is no specific milestones for you. All you need to do is to implement the following set of functions:

Function name	**Purpose**
`malloc_simple()`	Implementing allocation strategy - "*no management*"
`malloc_best_fit()`	Implementing allocation strategy - "*best-fit*"
`malloc_first_fit()`	Implementing allocation strategy - "*first-fit*"
`malloc_worst_fit()`	Implementing allocation strategy - "*worst-fit*"
`free_simple()`	Implementing deallocation strategy - "*no management*"
`free_no_tail()`	Implementing deallocation strategy - "*no-tail management*"

4.1 Deliverables

You have to submit a C or C++ source file (or a set of source files) that defines the above 6 functions.

4.2 Function Prototypes

- The `malloc_*()` functions have similar function prototypes:

```
void * malloc_*(unsigned int nbytes);
```

  - "`malloc_*`" means the set of allocation functions, i.e., `malloc_simple`, `malloc_best_fit`, `malloc_first_fit`, and `malloc_worst_fit`;
  - "`unsigned int nbytes`" is assumed to be a non-zero, positive integer.
  - The return value of the functions is the address pointing to the start of a piece of allocated memory. If `sbrk()` returns -1, the `malloc_*()` functions should return `NULL`.

- The `free_*()` functions have similar function prototypes:

```
void free_*(void *addr);
```

  - "`free_*`" means the set of deallocation functions, i.e., `free_simple` and `free_no_tail`;

  - “`void *addr`” is the starting address of an allocated memory returned by any one of the `malloc_*()` functions.

    Addresses other than the above are considered as invalid, and it is assumed that invalid addresses would not be supplied to any one of the `free_*()` functions.

  - No return value is needed, and it is assumed that `sbrk()` call would never return any errors when it is invoked correctly.

- You must use the `sbrk()` library call to grow and shrink the allocated heap space.

- The following library function calls are not allowed to be used while you are implementing the `malloc_*()` and the `free_*()` functions:

  `malloc()`, `calloc()`, `realloc()`, `alloca()`, and `free()`.

- **Reminder**: Other library functions may invoke `malloc()` and `free()` implicitly. Those library functions would conflict with your `malloc_*()` and `free_*()` implementation.

4.3 Marking

We will write programs to drive the functions you have written. Figures 13 and 14 show simple testing programs that drive your functions. This kind of programs are always called the **driver programs**.

All in all, we test the following three features:

- **Correct memory allocation**.

  The memory allocated should be usable and the size of allocated memory should match the requirement given by the user. In other words, a driver program should not catch any runtime errors such as segmentation fault. Figure 13 shows the example program that demonstrates such a test.

- **Utilizing the memory blocks**.

Our testing programs will certainly use the memory the function returned. A typical test is to invoke the `memset()` function over the range of memory returned by each `malloc_*()` call.

Yet, for buggy implementations, utilization of memory may result in errors for latter `malloc_*()` calls or `free_*()` calls. Under such situations, we do not look into the source of the problem; we will declare that you have the corresponding test case(s) failed.

- **Ordering of memory blocks**.

  Based on different strategies, we will have different resulting layouts of the allocated heap space. The next test is to check the relative locations of the allocated blocks. It is meaningless to check the exact memory address of the allocated blocks. Instead, we check the order of the allocated blocks. Figure 14 shows two example programs that demonstrate such kind of tests.

Of course, Figures 13 and 14 are **extremely easy cases**. You have to test your codes with more complicated cases.

5 Submission

For the submission of the assignment, please refer to our course homepage:

`http://www.cse.cuhk.edu.hk/~csci3150/`

Due date: 23:59, November 13, 2011 (Sun).

Note: the course instructor recommends you finishing the codes one week earilar because you may need another week for the peer-debugging process.

```
typedef struct header {
        struct header *ptr;
        unsigned int size;
} Header;

void * malloc_simple    (unsigned int nbytes);
void * malloc_first_fit (unsigned int nbytes);
void * malloc_best_fit  (unsigned int nbytes);
void * malloc_worst_fit (unsigned int nbytes);
void   free_simple      (void *address);
void   free_no_tail     (void *address);
```

Figure 12: The header file "`assignment_2.h`".

```
/** correctness_test.c **/
#include <stdio.h>
#include "assignment_2.h"

#define MALLOC  malloc_simple
#define FREE    free_simple

int main(void) {
    char *p;
    p = MALLOC(20);
    strcpy(p, "hello world");
    printf("%s\n", p);
    FREE(p);
    return 0;
}
```

Figure 13: An example program that invokes the `malloc()` and the `free()` calls implemented by you.

```
/** order_test_1.c **/
#include <stdio.h>
#include "assignment_2.h"

#define MALLOC  malloc_simple
#define FREE    free_simple

int main(void) {
    char *p1, *p2, *p3, *p4;
    p1 = MALLOC(20);
    p2 = MALLOC(20);
    p3 = MALLOC(20);
    FREE(p2);
    p4 = MALLOC(20);
    if(p1 < p2 &&
       p2 < p3 &&
       p3 < p4 )
        printf("Test passed.\n");
    else
        printf("Test failed.\n");
    FREE(p1);
    FREE(p3);
    FREE(p4);
    return 0;
}
```

```
/** order_test_2.c **/
#include <stdio.h>
#include "assignment_2.h"

#define MALLOC  malloc_worst_fit
#define FREE    free_simple

int main(void) {
    char *p1, *p2, *p3, *p4;
    p1 = MALLOC(20);
    p2 = MALLOC(20);
    p3 = MALLOC(20);
    FREE(p2);
    p4 = MALLOC(20);
    if(p1 < p2 &&
       p2 < p3 &&
       p2 == p4 )
        printf("Test passed.\n");
    else
        printf("Test failed.\n");
    FREE(p1);
    FREE(p3);
    FREE(p4);
    return 0;
}
```

Figure 14: Example programs that compare the locations of the allocated blocks.