Department of Computer
Science and Engineering

IIIT NAGPUR

2021-25

Heap Management Project

Designing Principles of Programming Languages Project

Under the guidance of:-
Dr. Milind Penurkar

By – Johann Michael (BT21CSE180)

Introduction

In this project, we have implemented a memory manager, which can be used to dynamically allocate and free memory at runtime. It conforms exactly to what we have learned during class.

3 representations of blocks were implemented. They are – blocks with only a header, blocks with a header and a footer, and finally blocks with an explicit free list.

For each representation, we have implemented 3 policies which are the first fit, best fit and next fit policies.

To avoid excessive repetition of the code, we first wrote an abstract and generic program that should work for any of the above 3 representations. This program contained a few abstract functions that were implemented separately for each representation.

When writing the common program, the design principle used was to create several layers of abstraction. Each layer uses the lower-level layer as primitives to build new operations and these new operations are used by higher level layer as primitives once again. This principle is very powerful in the structuring of large and complex programs, and is possibly the most important principle in all of programming.

Additionally, we have implemented an interactive wrapper around our program to facilitate interactive testing and examination of the heap. It functions like a mini-shell, and is very useful in debugging and testing the program, as well as to gain intuition about how the theory translates to actual bits and bytes on memory.

We have also tried to keep the code as readable as possible. Many functions as self-explanatory needing little commenting, and once one is familiar with the program, can easily be read, almost as though they are in English. We should always remember that programs are written primarily for humans to read and only incidentally for computers to execute. If we cannot read our own programs, it is doubtful to what extent we understand them. Our principle to making readable programs is to be explicit and clear in style, and write small but highly meaningful functions with clear and precise conventions on how they are to be used.

We will first show how the program can be used and then describe the code used to implement it.

How to use the program - 1

In this section, we will give a quick overview to our heap program. In the below main file, all that we do is create a heap of 512 bytes and specify the policy as BEST FIT. Immediately afterwards, we call the test_heap() function to start an interactive REPL for the purposes of examining the contents of the heap. Notice that we must include a “heap.h” header file.

```c
#include "heap.h"


int main()
{
    struct Heap* H = create_heap(512, BEST_FIT);

    /*
    char *arr = heap_alloc(H, 10);

    for(int i = 0; i < 10; i++)
        arr[i] = i;
    */

    test_heap(H);
}
```

The purpose of a makefile is to direct the compilation of a project. We use a makefile to compile our main.c program. In this project, we have implemented not just one block representation, but rather several. As the first line shows, in this instance, we have chosen to compile with the header.o file, meaning that our blocks will have only a header in their internal representation.

```makefile
HEAP_TYPE = header.o

a.out: main.c $(HEAP_TYPE)
    gcc -g main.c $(HEAP_TYPE)


header.o: header.c common.h heap.h
    gcc -g -c header.c

footer.o: footer.c common.h heap.h
    gcc -g -c footer.c

free_list.o: free_list.c common.h heap.h
    gcc -g -c free_list.c

clean:
    rm *.o a.out

.PHONY: clean
```

After compiling by running the command ‘make’ on the terminal, we will run the executable. As per the contents of the main.c file, a testing REPL opens straightway, on which we can enter commands. A sample interaction is shown below, so that you can get a feel for how it works.

```
johann:final$ ./a.out

Running testing REPL. Available commands are layout, dump, alloc, free, reset

>> layout
Displaying contents of the Heap
        Block 000 at address 00000000 of size 512 is free

>> alloc 40
Allocated 40 bytes successfully

>> layout
Displaying contents of the Heap
        Block 000 at address 00000000 of size 044 is allocated
        Block 001 at address 0000002c of size 468 is free

>> dump d 0 100   4
          0        4        8        c
00000000: 0000002d 00000000 00000000 00000000
00000010: 00000000 00000000 00000000 00000000
00000020: 00000000 00000000 00000000 000001d4
00000030: 00000000 00000000 00000000 00000000
00000040: 00000000 00000000 00000000 00000000
00000050: 00000000 00000000 00000000 00000000
00000060: 00000000 00000000 00000000 00000000

>> dump d 0 100  8
          0        4        8        c        10       14       18       1c
00000000: 0000002d 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000020: 00000000 00000000 00000000 000001d4 00000000 00000000 00000000 00000000
00000040: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000060: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

>> alloc 40
Allocated 40 bytes successfully

>> layout
Displaying contents of the Heap
        Block 000 at address 00000000 of size 044 is allocated
        Block 001 at address 0000002c of size 044 is allocated
        Block 002 at address 00000058 of size 424 is free

>> dump all 4
```

```
>> dump all 4
          0        4        8        c
00000000: 0000002d 00000000 00000000 00000000
00000010: 00000000 00000000 00000000 00000000
00000020: 00000000 00000000 00000000 0000002d
00000030: 00000000 00000000 00000000 00000000
00000040: 00000000 00000000 00000000 00000000
00000050: 00000000 00000000 000001a8 00000000
00000060: 00000000 00000000 00000000 00000000
00000070: 00000000 00000000 00000000 00000000
00000080: 00000000 00000000 00000000 00000000
00000090: 00000000 00000000 00000000 00000000
000000a0: 00000000 00000000 00000000 00000000
000000b0: 00000000 00000000 00000000 00000000
000000c0: 00000000 00000000 00000000 00000000
000000d0: 00000000 00000000 00000000 00000000
000000e0: 00000000 00000000 00000000 00000000
000000f0: 00000000 00000000 00000000 00000000
00000100: 00000000 00000000 00000000 00000000
00000110: 00000000 00000000 00000000 00000000
00000120: 00000000 00000000 00000000 00000000
00000130: 00000000 00000000 00000000 00000000
00000140: 00000000 00000000 00000000 00000000
00000150: 00000000 00000000 00000000 00000000
00000160: 00000000 00000000 00000000 00000000
00000170: 00000000 00000000 00000000 00000000
00000180: 00000000 00000000 00000000 00000000
00000190: 00000000 00000000 00000000 00000000
000001a0: 00000000 00000000 00000000 00000000
000001b0: 00000000 00000000 00000000 00000000
000001c0: 00000000 00000000 00000000 00000000
000001d0: 00000000 00000000 00000000 00000000
000001e0: 00000000 00000000 00000000 00000000
000001f0: 00000000 00000000 00000000 00000000
```

```
>> layout
Displaying contents of the Heap
        Block 000 at address 00000000 of size 044 is allocated
        Block 001 at address 0000002c of size 044 is allocated
        Block 002 at address 00000058 of size 424 is free

>> free 1
Freed block no. 1

>> free 0
Freed block no. 0

>> layout
Displaying contents of the Heap
        Block 000 at address 00000000 of size 512 is free

>> q
johann:final$ 
```

Now we will describe the available commands one by one in detail

1) Layout – This command will print the current block layout. It prints the address, size and allocation status of each block in the order of their occurrence. Sizes are printed in **decimal** while addresses are printed in **hexadecimal**.

2) Alloc – The alloc command can be used to allocate some bytes in the REPL itself. This allocation is treated as no different from an malloc() type call that would occur while a program is running. Its format is alloc <x> , where x is the allocation size in bytes.

3) Free – The free command is the counterpart to the alloc command. It can be used to free a block. Once again, it is treated as no different from a free() call that could occur while a program is running. However, instead of passing a pointer, pointing to the block to be freed, we must instead specify the block number as shown in the listing provided by the layout command, due to the limitations of the REPL.

4) Dump – Dump can be used to dump all the contents of the onto the heap. It functions like a typical memory dump. Addresses are printed as usual, in hexadecimal. Bytes are grouped into words, while printing and words are grouped into columns. The size of a column can be specified while entering the dump command. Dump can accept a few different formats.

    dump x <start address in hex> <end address in hex> <no. of columns>

    dump d <start address in decimal> <end address in decimal> <no. of columns>

    dump all <no. of columns> (dumps the entire heap)

5) q – The q command simply quits the REPL loop

6) Reset – There is one more command which is the reset command. The reset command will clear the heap and the allow the user to specify a custom initial block layout. Its format is given below. An example usage is also given afterwards.

reset (b|w)	(b – block sizes are in bytes, w – sizes are in words)
<size of block 0> (a | f)	(a – allocated, f – free)
<size of block 1> (a | f)	(a – allocated, f – free)
…	
0	(indicates end of input)

We can also batch the input commands inside an input file (similar to a shell script) . This saves us the task of repeatedly running the same set of commands.

```
reset b
16 f
16 a
48 f
40 f
32 a
24 f
16 f
0

layout

alloc 40
layout

alloc 8
layout

alloc 8
layout

dump all 4
q
```

Then we can simply execute ./a.out < input and all the commands while be run, without us having to repetitively type them out each time. This is of immense convenience when testing the program.

How to use the program – 2

The allocation and the freeing can be done during the program execution, exactly as we use malloc() and free() in our program to obtain dynamic memory. To demonstrate this, let us uncomment the commented-out lines in the main program below. Those lines allocate 10 bytes for a character array and then write the values from 0 to 9 in that array. We will have to recompile the program using make.

```c
#include "heap.h"


int main()
{
    struct Heap* H = create_heap(512, BEST_FIT);

    /*
    char *arr = heap_alloc(H, 10);

    for(int i = 0; i < 10; i++)
        arr[i] = i;
    */

    test_heap(H);
}
```

Afterwards, when we run the program, we will again enter the testing REPL. Here we can see in the memory dump that the numbers from 0 to 9 were actually written in the heap memory. Each word is 4 bytes and we are using the little endian-format. One byte takes 2 hex characters. The entire array of 10 bytes is stored the second, third and fourth words, as you see.

```
johann:final$ ./a.out

Running testing REPL. Available commands are layout, dump, alloc, free, reset

>> dump all 4
          0        4        8        c
00000000: 00000011 03020100 07060504 00000908
00000010: 000001f0 00000000 00000000 00000000
00000020: 00000000 00000000 00000000 00000000
00000030: 00000000 00000000 00000000 00000000
00000040: 00000000 00000000 00000000 00000000
00000050: 00000000 00000000 00000000 00000000
00000060: 00000000 00000000 00000000 00000000
00000070: 00000000 00000000 00000000 00000000
```

How to use the program – 3

Let us change the makefile and instead use a free list representation of the blocks. In the makefile, we must edit the first line to reflect this.

```
HEAP_TYPE = free_list.o

a.out: main.c $(HEAP_TYPE)
    gcc -g main.c $(HEAP_TYPE)


header.o: header.c common.h heap.h
    gcc -g -c header.c

footer.o: footer.c common.h heap.h
    gcc -g -c footer.c

free_list.o: free_list.c common.h heap.h
    gcc -g -c free_list.c

clean:
    rm *.o a.out

.PHONY: clean
```

After starting the program, we use the reset command. As mentioned earlier, the reset command is used to specify a custom initial block layout. Its format is

reset b|w	(b – sizes are in bytes, w – sizes are in words)
<size of block 0> a | f	(a – allocated, f – free)
<size of block 1> a | f	(a – allocated, f – free)
…	
0	(indicates end of input)

```
johann:final$ ./a.out

Running testing REPL. Available commands are layout, dump, alloc, free, reset

>> reset b
Enter the block layout. Give 0 as block size when done.
16 a
16 f
16 a
16 f
16 a
0
Done
```

Finally, when we display the contents of the heap, we can see that an explicit free-list was created. By following the free blocks in the layout, we can see how they reference each other to maintain an explicit free-list. (Important – The allocated blocks are not part of the free list). The address of all ONES works like the null pointer. It is not possible to use all zeros for the null pointer because it can be confused with a block beginning at address 0.

```
>> layout
Displaying contents of the Heap
        Block 000 at address 00000000 of size 016 is allocated
        Block 001 at address 00000010 of size 016 is free
        Block 002 at address 00000020 of size 016 is allocated
        Block 003 at address 00000030 of size 016 is free
        Block 004 at address 00000040 of size 016 is allocated
        Block 005 at address 00000050 of size 432 is free

>> dump all 4
          0        4        8        c
00000000: 00000011 ffffffff ffffffff 00000005
00000010: 00000010 ffffffff 00000030 00000004
00000020: 00000011 ffffffff ffffffff 00000005
00000030: 00000010 00000010 00000050 00000004
00000040: 00000011 ffffffff ffffffff 00000005
00000050: 000001b0 00000030 ffffffff 00000000
00000060: 00000000 00000000 00000000 00000000
00000070: 00000000 00000000 00000000 00000000
00000080: 00000000 00000000 00000000 00000000
00000090: 00000000 00000000 00000000 00000000
000000a0: 00000000 00000000 00000000 00000000
000000b0: 00000000 00000000 00000000 00000000
000000c0: 00000000 00000000 00000000 00000000
000000d0: 00000000 00000000 00000000 00000000
000000e0: 00000000 00000000 00000000 00000000
000000f0: 00000000 00000000 00000000 00000000
00000100: 00000000 00000000 00000000 00000000
00000110: 00000000 00000000 00000000 00000000
00000120: 00000000 00000000 00000000 00000000
00000130: 00000000 00000000 00000000 00000000
00000140: 00000000 00000000 00000000 00000000
00000150: 00000000 00000000 00000000 00000000
00000160: 00000000 00000000 00000000 00000000
00000170: 00000000 00000000 00000000 00000000
00000180: 00000000 00000000 00000000 00000000
00000190: 00000000 00000000 00000000 00000000
000001a0: 00000000 00000000 00000000 00000000
000001b0: 00000000 00000000 00000000 00000000
000001c0: 00000000 00000000 00000000 00000000
000001d0: 00000000 00000000 00000000 00000000
000001e0: 00000000 00000000 00000000 00000000
000001f0: 00000000 00000000 00000000 0000006c

>>
```

Structure of the program

In this section, we will explain the role of the various files in our program. There are five key files -

1) common.h – This file contains functions and abstractions that are common to all the different heap representations. In order to make this possible, some key primitive functions are kept abstract while the higher-level functions are implemented in terms of these abstract functions. In some sense, the common.h file acts like a program template rather than a program. By implementing the abstract functions suitably, we can customize towards a desired heap implementation, such as one using an explicit free-list or only headers, etc.

2) header.c – This file contains an implementation of the functions that were kept abstract in the common.h file. This implementation is for a representation of the blocks that uses only a header. We can compile common.h and header.h together to get an object file, say header.o, which can then itself be linked with our main program later.

3) footer.c – This file contains the does the same thing as the header.c file but for a block representation that has a header as well as a footer.

4) free_list.c – This file contains the does the same thing as the header.c file but for a block representation that uses an explicit free list.

5) heap.h – This file is our interface to the user. The user should be protected from the internal details of our program and only an interface should be presented to him/her. This interface is present in our heap.h file. The heap.h file also contains some type definitions and importantly, a choice of the word size (4 bytes or 8 bytes)

Additionally, there are a few other files

1) main.c – This main.c file has the driver program. This is not a necessary file for the project. It is only present as a dummy, in place of a real program that uses dynamic allocation.

2) Makefile – A makefile can be used to automate compilation. Compilation quickly becomes a tedious and tiresome process, when more than two or three files are involved. Makefiles are the solution. Additionally, in our makefile, we decide what implementation we want to link our main file with.

The program – heap.h

The heap.h file contains the set of functions and declarations that must be exposed to the user. These are purposefully kept rather minimal. A full listing of the heap.h file is given below.

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


#define WORD_SIZE 4
#define BLOCK_ALIGN 1    //is specified in words

typedef char BYTE;

#if WORD_SIZE == 8
typedef long long WORD;
#define WFX "llx"
#define WFS "lld"

#elif WORD_SIZE == 4
typedef int WORD;
#define WFX "x"
#define WFS "d"

#endif


// Defining the heap

typedef enum {FIRST_FIT, BEST_FIT, NEXT_FIT} POLICIES;

struct Heap {
    WORD size;
    BYTE *bytes;

    POLICIES policy;
    WORD* next_free;
    WORD* head_free;
};


struct Heap* create_heap(int size, POLICIES policy);


char* heap_alloc(struct Heap* H, int bytes);
void heap_free(struct Heap* H, char* payload);

void test_heap(struct Heap* H);
```

The interface consists of 4 functions

1) create_heap()
2) heap_alloc()
3) heap_free()
4) test_heap()

Importantly, this file also specifies the word size in bytes (WORD_SIZE) and the block alignment in words (BLOCK_ALIGN). Both of these can be changed. WORD_SIZE can be set to 4 or 8. BLOCK_ALIGN can be set to any power of 2.

In order to support this generality, the program has to written without assuming any specific word size. This was done by using a typedef alias - WORD. WORD is an abstract type that stands for a machine word. For a 32-bit machine, WORD is aliased to ‘int’ and in a 64-bit machine, WORD can be aliased to a ‘long long’. The correct type is automatically chosen at compilation time depending on WORD_SIZE. This was achieved through the use of conditional compilation.

In the same spirit, we also kept a typedef BYTE, which was aliased to ‘char’. Thought we do not experiment with multiple byte sizes, it keeps the program abstract in style.

The program – common.h (High level functions)

The common.h file is really a program template. It is a generic heap program that should work for any heap implementation. This was done so as to capture commonality and experiment with writing an abstract and general program.

The common.h file has a fairly complex structure, involving several layers of abstractions that isolate details from each other. In the following pages, we will list all the functions in the order of abstraction. That is, we will present high-level functions first and low-level functions afterwards.

```
// THe alloc() and free() functions, which are exposed to the user

char* heap_alloc(struct Heap* H, int bytes) {
    WORD size = block_size_of_payload(bytes_to_words(bytes));

    WORD* block = NULL;
    if(H->policy == FIRST_FIT)
        block = first_fit(H, size);
    else if(H->policy == BEST_FIT)
        block = best_fit(H, size);
    else if(H->policy == NEXT_FIT)
        block = next_fit(H, size);


    if(block == NULL)
        return NULL;

    block = alloc_from_block(H, block, size);
    clear_payload(block);

    return (char *) block_to_payload(block);
}

void heap_free(struct Heap* H, char* payload) {
    WORD* block = payload_to_block((BYTE* ) payload);
    free_block(H, block);
}

```

The code should be fairly self-explanatory.

In the alloc() function, we accept an allocation size in bytes. This is first converted into words. The resulting size is the ‘payload’ size in words. To get the minimum block size, we must account for the sizes of the meta-data such as the header, footer, etc. Through this, we get the minimum size that a block must have, to accommodate the user’s allocation requirement.

Next, we search the heap to ‘find’ a block from which we are going to allocate the needed memory. This searching depends upon the policy that we are using. It can be first fit, best fit, or next fit.

Afterwards, we actually allocate the needed memory from the chosen block, clear its payload and return a pointer to the payload to the user.

In the free() function, we do very little. The payload pointer that the user passes is first converted to a block pointer, after which the actual freeing is delegated to a different function.

Importantly, both the free_block() and the alloc_from_block() functions are abstract in the common.h file, i.e. they have no implementation. They are only declared. A specific implementation must be provided for each block representation.

```
// The basic allocator and deallocator used to build the actual alloc() and free()

WORD* alloc_from_block(struct Heap* H, WORD* block, WORD alloc_size);

void free_block(struct Heap *H, WORD* block);
```

Next, we have the definition of some of the helper functions used in heap_alloc() and heap_free()

```
// Functions to handle the relationship between a block and its payload

BYTE* block_to_payload(WORD* block) {
    return (BYTE*) (block + 1);
}

WORD* payload_to_block(BYTE* payload) {
    return (WORD *) payload - 1;
}

WORD payload_size(WORD* block) {
    return get_size(block) - ALLOC_FIXED_SIZE;
}

void clear_payload(WORD* block) {
    int size = get_size(block);
    memset(block_to_payload(block), 0, payload_size(block) * WORD_SIZE);
}




// Utility function to convert bytes to words

WORD bytes_to_words(int bytes) {
    WORD words = bytes / WORD_SIZE;
    if(bytes % WORD_SIZE != 0)
        words++;

    return words;
}
```

ALLOC_FIXED_SIZE stores the total size of the meta-data in an allocated block. FREE_FIXED_SIZE stores the total size of the meta-data in a free block. These values permit us to determine the size of a block just from the size of its payload. While determining these minimum sizes, we also account for the block alignment.

```
// Determining the minimum size of a block, given its payload size

extern int ALLOC_FIXED_SIZE;
extern int FREE_FIXED_SIZE;


WORD block_size_of_payload(WORD size) {
    size += ALLOC_FIXED_SIZE; //Adding the fixed part

    if(size % BLOCK_ALIGN != 0)  //Accounting for alignment
        size += (BLOCK_ALIGN - size % BLOCK_ALIGN);

    return size;
}


WORD min_free_block_size() {
    WORD size = FREE_FIXED_SIZE + 1;

    if(size % BLOCK_ALIGN != 0)
        size += (BLOCK_ALIGN - size % BLOCK_ALIGN);

    return size;
}
```

Next, we show the implementation of the first_fit(), best_fit() and next_fit() functions. These are purposefully kept rather simple, small and intuitive. Their purpose is only to select a block for allocation from the heap. The actual allocation is handled elsewhere. These functions work for **ALL** the different block representations.

```
// Implementation of various policies for selecting a block for allocation

WORD* first_fit(struct Heap* H, int min_size) {
    for(WORD* B = traverse_start(H); !traverse_over(H, B); B = traverse_next(H, B)) {
        if(!is_free(B))
            continue;

        if(get_size(B) >= min_size)
            return B;
    }

    return NULL;
}


WORD* best_fit(struct Heap* H, int min_size) {
    WORD* best = NULL;

    for(WORD* B = traverse_start(H); !traverse_over(H, B); B = traverse_next(H, B)) {
        if(!is_free(B))
            continue;

        if(get_size(B) >= min_size) {
            if(best == NULL || get_size(B) < get_size(best))
                best = B;
        }
    }

    return best;
}


WORD* next_fit(struct Heap* H, int min_size) {
    WORD* B = H->next_free;
    if(B == NULL)
        return NULL;

    WORD* start = B;

    int blocks_seen = 0;
    while(! (blocks_seen > 0 && traverse_wrap(H, B) == start) ) {
        if(is_free(B)) {
            if(get_size(B) >= min_size)
                return B;
        }

        B = traverse_wrap(H, B);
        blocks_seen++;
    }

    return NULL;
}
```

As you can see, they depend upon a few key functions – traverse_start(), traverse_next(), traverse_over() and traverse_wrap(). These are traversal functions, similar to C++ iterators, which we can use to traverse our heap. Additionally, traverse_wrap() does a wrap traversal, i.e. it will start over from the beginning, if we have crossed the end.

These functions are kept abstract once again. This is because, in an implicit list, we will traverse all the blocks, whereas in an explicit list, we traverse only the free blocks.

```
// Abstract functions for traversing through some arbitrary list of blocks (for the policy functions)

WORD* traverse_start(struct Heap* H);

WORD* traverse_next(struct Heap* H, WORD* B);

WORD* traverse_wrap(struct Heap* H, WORD* B);

int traverse_over(struct Heap* H, WORD* B);
```

The program – common.h (Intermediate level functions)

The intermediate level functions operate at a lower level than the high-level functions. More precisely, they define a set of operations which are used by the high-level functions in their implementation. The intermediate level functions are themselves implemented in terms of lower-level functions.

Below we have the primitive merge and split operations. These are primitive in the sense that they should be used to implement the actual merging and splitting in alloc_block() and free_block(). Both merge_block() and split_block() are defined very naturally in terms of create_block(). To merge two blocks, we simply add up their sizes and create a fresh block of the total size. To split a block, we determine the sizes of the left and right blocks and create two fresh blocks of those sizes.

Create_block() is a function that will create a fresh block on memory of a given size. It is kept abstract, since the meta-data that must be initialized varies between each representation.

```
// An implementation of merging and splitting

void create_block(WORD* block, WORD size);

void merge_blocks(WORD* lblock, WORD* rblock) {
    WORD size = get_size(lblock) + get_size(rblock);
    create_block(lblock, size);
}

void split_block(WORD* block, WORD lsize) {
    WORD size = get_size(block);

    if(lsize > size || lsize <= 0 || lsize % BLOCK_ALIGN != 0) {
        printf("ERROR: Invalid value of lsize passed to split_block()\n");
        exit(1);
    }

    create_block(block, lsize);
    create_block(block + lsize, size - lsize);
}

```

In addition, we also implement explicit functions for traversing the heap. These functions are not in any way related to the first_fit() or next_fit() functions. They are present simply to abstract away the details of moving from one block to the other inside the heap. Once again they are primitives, to be used by higher level functions.

```c
// Functions for traversing through a heap block by block

WORD* next_block(WORD* block) {
    return block + get_size(block);
}


WORD* first_block(struct Heap* H) {
    return (WORD*) H->bytes;
}


int at_heap_end(struct Heap *H, WORD* block) {
    if(block - first_block(H) < H->size)
        return 0;
    else
        return 1;
}

int at_heap_start(struct Heap* H, WORD* block) {
    if(block == first_block(H))
        return 1;
    else
        return 0;
}
```

The program – common.h (Low level functions)

Finally, we have low-level functions that operate directly with the bits.

```c
//Selector functions

WORD get_header(WORD* block) {
    return *block;
}

int get_a_bit(WORD* block) {
    return get_header(block) & 0x1;
}

int is_free(WORD* block) {
    return get_a_bit(block) == 0;
}

WORD size_from_header(WORD header) {
    return (header & ~(BLOCK_ALIGN * WORD_SIZE - 1)) / WORD_SIZE;
    //return header & ~(BLOCK_ALIGN - 1);
}

WORD get_size(WORD* block) {
    return size_from_header(get_header(block));
    //return get_header(block) & ~(BLOCK_ALIGN - 1);
}





// Setter functions

void set_header(WORD* block, WORD size, int a) {
    *block = size * WORD_SIZE | a;
    //*block = size | a;
}

void set_a_bit(WORD* block, int a) ;

```

Once may notice the absence of functions to select and modify the footer, or the next and prev pointers in an explicit free list. The reason is that common.h is a generic file. Only the header is present across all implementations, which is why only setters and getters for the header were implemented. Conveniently, this also allows us to access the size and allocation status.

The program – common.h (Defining and creating the heap)

We have some functions to handle to manage the heap data structure. First, we repeat the its definition verbatim from the heap.h file.

```

// Defining the heap

typedef enum {FIRST_FIT, BEST_FIT, NEXT_FIT} POLICIES;

struct Heap {
    WORD size;
    BYTE *bytes;

    POLICIES policy;
    WORD* next_free;
    WORD* head_free;
};

```

We have functions for creating a heap and initializing a heap.

```
#include "heap.h"

//Functions for the Heap

void init_heap(struct Heap* H);

struct Heap* create_heap(int size, POLICIES policy) {
    if(size <= 0 || size % (WORD_SIZE * BLOCK_ALIGN) != 0) {
        printf("ERROR: Invalid size passed to create_heap()\n");
        exit(1);
    } else {
        size /= WORD_SIZE;  //Converting from bytes to words

        struct Heap *H = malloc(sizeof(struct Heap));
        H->size = size;
        H->bytes = malloc(sizeof(WORD) * size);
        H->next_free = NULL;
        H->policy = policy;
        H->head_free = NULL;

        init_heap(H);
        return H;
    }
}
```

```
// An implementation of init_heap() using create_block()

void init_heap(struct Heap* H) {
    memset(H->bytes, 0, sizeof(BYTE) * H->size * WORD_SIZE);

    WORD* block = (WORD*) H->bytes;
    create_block(block, H->size);
    H->head_free = block;
    H->next_free = block;
}
```

We also have function to reset the next_free state variable in the heap data structure to the first free block.

```
// Resetting the heap to the first free block

void reset_heap(struct Heap* H) {
    WORD* B;
    for(B = first_block(H); !at_heap_end(H, B); B = next_block(B))
        if(is_free(B))
            break;

    H->next_free = at_heap_end(H, B) ? NULL : B;
}
```

The program – common.h (The testing REPL)

These functions are only for the implementation of the testing REPL.

We have a function to execute the layout command. I.e., to print the heap layout.

```
// The remaining functions are used for testing purposes only


// Prints the block layout of the heap

void heap_layout(struct Heap* H) {
    printf("Displaying contents of the Heap\n");

    int i = 0;
    for(WORD* B = first_block(H); !at_heap_end(H, B); B = next_block(B)) {
        printf("\tBlock %0*d", 3, i);
        printf(" at address %0*"WFX, WORD_SIZE * 2, (WORD) ( (BYTE* ) B - (BYTE* ) first_block(H)));
        printf(" of size %0*"WFS, 3, get_size(B) * WORD_SIZE);
        printf(" is %s\n", is_free(B) ? "free" : "allocated");

        i++;
    }
}
```

A function to implement the dump command. It performs a memory dump and prints the contents onto the screen.

```
int resolution(int x, int step) {
    return step * (x / step);
}




// Dumps memory from the heap onto the screen

void heap_dump(struct Heap* H, BYTE* start, int num, int words_per_row) {
    int width = WORD_SIZE * 2;
    int row_size = words_per_row * WORD_SIZE;

    //Printing the header
    printf("%*s ", width + 1, "");
    for(int i = 0; i < words_per_row; i++)
        printf("%-*x ", width, i * WORD_SIZE);
    printf("\n");


    int byte_start = start - H->bytes;
    int byte_end = byte_start + num;

```

```c
    int row_start = resolution(byte_start, row_size);
    int row_end   = resolution(byte_end - 1, row_size) + row_size;


    for(int row = row_start; row < row_end; row += row_size) {
        //Printing the address of the row
        printf("%0*x: ", width, row);

        //Printing the contents of the row
        for(int i = 0; i < words_per_row; i++) {
            int word = row + (i * WORD_SIZE);

            if(word/WORD_SIZE < H->size)
                printf("%0*"WFX" ", width, * (WORD *) (H->bytes + word));
            else {
                for(int i = 0; i < width; i++)
                    putchar('_');
                putchar(' ');
            }
        }
        printf("\n");
    }
}
```

Finally, a method to implement the REPL itself. It presents a shell-like prompt “>>” to the user and reads the user’s input to execute the appropriate command. It can be viewed as the implementation of a simple interactive shell.

```c
// An abstract function needed by test_heap() while performing a reset

void split_free_block(struct Heap *H, WORD* B, WORD split_size);




// An REPL testing loop, to facilitate easy testing and viewing

void test_heap(struct Heap* H) {
    char s[50];

    printf("\nRunning testing REPL. Available commands are layout, dump, alloc, free, reset\n");
    while(1) {
        printf("\n>> ");

        scanf("%49s", s);

        if(!strcmp(s, "q"))
            break;

        if(!strcmp(s, "layout")) {
            heap_layout(H);
            continue;
        }

        if(!strcmp(s, "dump")) {
            int start, end, words_per_row;
```

```
            scanf("%s", s);
            if(!strcmp(s, "d"))
                scanf("%d%d%d", &start, &end, &words_per_row);
            else if(!strcmp(s, "x"))
                scanf("%x%x%d", &start, &end, &words_per_row);
            else if(!strcmp(s, "all")) {
                scanf("%d", &words_per_row);

                start = 0;
                end = H->size * WORD_SIZE - 1;
            }

            heap_dump(H, H->bytes + start, end - start, words_per_row);
            continue;
        }

        if(!strcmp(s, "alloc")) {
            int bytes;
            scanf("%d", &bytes);
            char* p = heap_alloc(H, bytes);

            if(p == NULL)
                printf("Failed to allocate\n");
            else
                printf("Allocated %d bytes successfully\n", bytes);

            continue;
        }

        if(!strcmp(s, "free")) {
            int num;
            scanf("%d", &num);

            WORD* B = first_block(H);
            int block_num = num;
            while(num-- > 0) {
                if(at_heap_end(H, B)) {
                    B = NULL;
                    break;
                }

                B = next_block(B);
            }

            if (B == NULL)
                printf("Invalid block number\n");
            else if (is_free(B))
                printf("Block is already free\n");
            else {
                free_block(H, B);
                printf("Freed block no. %d\n", block_num);
            }

            continue;
        }

        if(!strcmp(s, "reset")) {
            scanf("%s", s);

```

```
            int unit;
            if(!strcmp(s, "w"))
                unit = WORD_SIZE;
            else
                unit = 1; //byte


            init_heap(H);
            WORD* block = first_block(H);

            printf("Enter the block layout. Give 0 as block size when done.\n");
            while(1) {
                int size, a;

                scanf("%d", &size);
                if(size == 0) {
                    printf("Done\n");
                    break;
                }

                if(at_heap_end(H, block)) {
                    printf("Heap is fully occupied\n");
                    break;
                }

                size = (size * unit) / WORD_SIZE;

                scanf("%s", s);
                if(!strcmp(s, "a"))
                    alloc_from_block(H, block, size);
                else
                    split_free_block(H, block, size);

                block = next_block(block);
            }



  //Maybe we need a new NULL value for when no next fit has happened

            reset_heap(H);
            continue;
        }
        printf("?\n");
    }
}
```

The program – common.h (Conclusion)

The common.h file, as emphasized several times, is only a template. We must actually implement the abstract functions to get a working program. By implementing them in different ways, we can get programs using different block representations. In the next sections, we will show how to do this for three implementations-

1) Blocks with only a header
2) Blocks with a header and a footer
3) Explicit free list (header, footer, as well as next and prev pointers)

The program – header.c

Here we implement all the abstract functions for a header representation.

Since both free and allocated blocks contain one word of metadata which is the header, we set both of the below values to 1.

```
#include "common.h"

int ALLOC_FIXED_SIZE = 1;
int FREE_FIXED_SIZE = 1;
```

A setter for the allocation status bit.

```
void set_a_bit(WORD* block, int a) {
    set_header(block, get_size(block), a);
}
```

A function to create a block of a given size. As you can see, all that we do is set the header, nothing more.

```
void create_block(WORD* block, WORD size) {
    set_header(block, size, 0);
}
```

An implementation of the all important alloc_from_block() and free_block() functions. To allocate from a block, we use first decide if we need to split by checking whether the leftover size is less than the minimum, if it is, we split, otherwise we don't. Finally we reset the next_free variable, which is used in the next_fit() function.

```
WORD* alloc_from_block(struct Heap* H, WORD* block, WORD alloc_size) {
    WORD size = get_size(block);
    WORD min_size = min_free_block_size();

    if(size - alloc_size < min_size) {
        set_a_bit(block, 1);
    } else {
        split_block(block, alloc_size);
        set_a_bit(block, 1);
    }

    H->next_free = traverse_wrap(H, block);


    return block;
}
```

To free a block, all we do is set the alloc bit to zero, and then we check if the next block is free. If it is then we merge both blocks.

```c
void free_block(struct Heap *H, WORD* block) {
    set_a_bit(block, 0);

    WORD* adj = next_block(block);
    if(!at_heap_end(H, adj) && is_free(adj))
        merge_blocks(block, adj);

}
```

We also implement the traversal functions. They have a direct implementation in terms of first_block(), next_block() and at_heap_end().

```c
WORD* traverse_start(struct Heap* H) {
    return first_block(H);
}

WORD* traverse_next(struct Heap* H, WORD* B) {
    return next_block(B);
}

int traverse_over(struct Heap* H, WORD* B) {
    return at_heap_end(H, B);
}

WORD* traverse_wrap(struct Heap* H, WORD* B) {
    B = next_block(B);
    if(at_heap_end(H, B))
        B = first_block(H);

    return B;
}
```

Finally, a simple function split a free block into two smaller free blocks.

```c
void split_free_block(struct Heap* H, WORD* B, WORD split_size) {
    split_block(B, split_size);
}
```

The program – footer.c

This implementation is almost exactly the same as header.c

Since we have a header and a footer, the fixed-size value must be 2, not 1.

```
#include "common.h"

int ALLOC_FIXED_SIZE = 2;
int FREE_FIXED_SIZE  = 2;
```

For the footer, we introduce setters and getters.

```
WORD get_footer(WORD* block) {
    return *(block + get_size(block) - 1);
}



void set_footer(WORD* block, WORD size, int a) {
    *(block + size - 1) = size * WORD_SIZE | a;
    //*(block + size - 1) = size | a;
}

void set_a_bit(WORD* block, int a) {
    set_header(block, get_size(block), a);
    set_footer(block, get_size(block), a);
}

```

The function to create a new fresh block must now set the header as well as the footer.

```
void create_block(WORD* block, WORD size) {
    set_header(block, size, 0);
    set_footer(block, size, 0);
}

```

Using the footer, we can now traverse backwards, and accordingly, we implement a function for that purpose.

```
WORD* prev_block(WORD* block) {
    WORD* footer = block - 1;
    WORD size = size_from_header(*footer);
    return block - size;
}
```

The alloc_from_block() function is exactly the same as for the header representation.

While freeing, we check if we can merge not only with the following block, but also with the preceding block.

```
WORD* alloc_from_block(struct Heap* H, WORD* block, WORD alloc_size) {
    WORD size = get_size(block);
    WORD min_size = min_free_block_size();

    if(size - alloc_size < min_size) {
        set_a_bit(block, 1);
    } else {
        split_block(block, alloc_size);
        set_a_bit(block, 1);
    }

    H->next_free = traverse_wrap(H, block);

    return block;
}



void free_block(struct Heap *H, WORD* block) {
    set_a_bit(block, 0);

    WORD* next = next_block(block);
    if(!at_heap_end(H, next) && is_free(next))
        merge_blocks(block, next);

    if(!at_heap_start(H, block)) {
        WORD* prev = prev_block(block);
        if(is_free(prev))
            merge_blocks(prev, block);
    }

}
```

The same implementation for the traversal functions as for the header representation.

```
WORD* traverse_start(struct Heap* H) {
    return first_block(H);
}

WORD* traverse_next(struct Heap* H, WORD* B) {
    return next_block(B);
}

int traverse_over(struct Heap* H, WORD* B) {
    return at_heap_end(H, B);
}

WORD* traverse_wrap(struct Heap* H, WORD* B) {
    B = next_block(B);
    if(at_heap_end(H, B))
        B = first_block(H);

    return B;
}
```

Finally a small function to split a free block into two free blocks.

```

void split_free_block(struct Heap* H, WORD* B, WORD split_size) {
    split_block(B, split_size);
}

```

The program – free_list.c

The free list representation is considerably more complicated than either of the previous two.

We first have the implementations of the low-level functions. Importantly among them, we have setters and getters for the next and prev pointers, which are needed by the explicit free list. Also, the fixed size portion of a free block is 4 words, while for an allocated block, it is only 2 words.

```
#include "common.h"

int ALLOC_FIXED_SIZE = 2; //What to really do about this ? Probably the best solution
int FREE_FIXED_SIZE = 4;

WORD HEAP_NULL = ~0;


WORD get_footer(WORD* block) {
    return *(block + get_size(block) - 1);
}

WORD get_prev(WORD* block) {
    return *(block + 1);
}

WORD get_next(WORD* block) {
    return *(block + 2);
}




void set_footer(WORD* block, WORD size, int a) {
    *(block + size - 1) = size | a;
}

void set_a_bit(WORD* block, int a) {
    set_header(block, get_size(block), a);
    set_footer(block, get_size(block), a);
}

void set_prev(WORD* block, WORD prev) {
    *(block + 1) = prev;
}

void set_next(WORD* block, WORD next) {
    *(block + 2) = next;
}

```

This is an implementation of the create_block() function. To create a block, we must set the header and footer, as before. Additionally, we initialize the prev and next pointers to HEAP_NULL, which is defined to be all the address of all ONES. We cannot use the value of the null pointer which is zero, because 0 can be confused with an actual block starting at address 0.

```c
//Assuming that alignment will be 4 words or equivalent
void create_block(WORD* block, WORD size) {
    set_header(block, size, 0);
    set_footer(block, size, 0);
    set_prev(block, HEAP_NULL);
    set_next(block, HEAP_NULL);
}

WORD* prev_block(WORD* block) {
    WORD* footer = block - 1;
    WORD size = size_from_header(*footer);
    return block - size;
}
```

Importantly, in the next and prev pointers we are not going to store the absolute RAM address, but instead the address relative to the start of the heap. So, for example, the first block will have an address of 0, when in fact, 0 is not a valid RAM address as it is NULL. Since we routinely use pointers to the block throughout our program, we need interfacing functions to handle this conversion.

```c
WORD* block_pointer(struct Heap* H, WORD addr) {
    if(addr == HEAP_NULL)
        return NULL;
    else
        return (WORD* ) (H->bytes + addr);
}

WORD block_address(struct Heap* H, WORD* B) {
    if(B == NULL)
        return HEAP_NULL;
    else
        return (BYTE* ) B - H->bytes;
}
```

Link_blocks() is used to connect two nodes of a doubly linked list together by setting the pointers at both sides correctly in one go.

```c
void link_blocks(struct Heap *H, WORD* A, WORD* B) {
    if(A != NULL)
        set_next(A, block_address(H, B));

    if(B != NULL)
        set_prev(B, block_address(H, A));
}
```

Next, we have an implementation of the traversal functions. While previously, the traversal functions were implemented using first_block(), next_block(), and so on, here the traversal function are implemented using the next and prev pointers stored in each free block (since it is an explicit free list)

```
WORD* traverse_start(struct Heap* H) {
    return H->head_free;
}

WORD* traverse_next(struct Heap* H, WORD* B) {
    return block_pointer(H, get_next(B));
}

WORD* traverse_wrap(struct Heap* H, WORD* B) {
    if(B == NULL)
        return NULL;

    WORD* next = block_pointer(H, get_next(B));
    if(next == NULL)
        return H->head_free;
    else
        return next;
}

int traverse_over(struct Heap* H, WORD* B) {
    return B == NULL;
}
```

Now we have implementations of the key functions alloc_block() and free_block(). Both use helper functions - reduce_block() and extend_block() respectively. These helper functions will reduce or increase the size of a block while maintaining the doubly linked list and state information. There are a few variables that need to be correctly maintained - prev pointer, next pointer, and head_free (head of the free list).

```
void reduce_block(struct Heap* H, WORD* B, int alloc_size) {
    WORD prev = get_prev(B);
    WORD next = get_next(B);

    if(alloc_size == get_size(B)) {
        link_blocks(H, block_pointer(H, prev), block_pointer(H, next));

        if(B == H->head_free)
            H->head_free = block_pointer(H, get_next(B));

    } else if (alloc_size < get_size(B)) {
        split_block(B, alloc_size);

        link_blocks(H, block_pointer(H, prev), B + alloc_size);
        link_blocks(H, B + alloc_size, block_pointer(H, next));

        if(B == H->head_free)
            H->head_free = B + alloc_size;
    }
}



WORD* alloc_from_block(struct Heap* H, WORD* block, WORD alloc_size) {
    WORD size = get_size(block);
    WORD min_size = min_free_block_size();

    if(size - alloc_size < min_size) {
        reduce_block(H, block, size);
        H->next_free = traverse_wrap(H, block);   //We traverse and wrap
    } else {
        reduce_block(H, block, alloc_size);
        H->next_free = block + alloc_size;
    }

    set_a_bit(block, 1);

    return block;
}
```

```c
void extend_block(struct Heap *H, WORD* A, WORD* B, WORD* which_one) { //which one is being extended
    WORD prev, next;

    if(which_one == A) {
        prev = get_prev(A);
        next = get_next(A);
    } else if (which_one == B) {
        prev = get_prev(B);
        next = get_next(B);
    }

    merge_blocks(A, B);

    link_blocks(H, block_pointer(H, prev), A);
    link_blocks(H, A, block_pointer(H, next));

    if(which_one == B && B == H->head_free)
        H->head_free = A;

}



void free_block(struct Heap *H, WORD* block) {
    set_a_bit(block, 0);

    WORD* next = next_block(block);
    if(!at_heap_end(H, next) && is_free(next)) {
        extend_block(H, block, next, next);

    } else if(!at_heap_start(H, block)) {
        WORD* prev = prev_block(block);
        if(is_free(prev))
            extend_block(H, prev, block, prev);
    } else {
        if(H->head_free == NULL)
            H->head_free = H->next_free = block;
        else { // We are pushing the new block onto the head. If we want, we can do something different
            link_blocks(H, block, H->head_free);
            H->head_free = block;
        }
    }

}
```

Finally, a function to split a free block into two smaller free blocks. The important thing to ensure while doing this is that the doubly linked list remains connected properly.

```c
void split_free_block(struct Heap *H, WORD* B, WORD split_size) {
    if(split_size == get_size(B))
        return;

    WORD prev = get_prev(B);
    WORD next = get_next(B);

    split_block(B, split_size);

    link_blocks(H, block_pointer(H, prev), B);
    link_blocks(H, B, B + split_size);
    link_blocks(H, B + split_size, block_pointer(H, next));
}
```