Assignment 2: allocator, an allocator

version: 1.0 last updated: 2022-06-23 09:00:00

Aims

- to give you experience writing C code to manipulate memory
- to give you experience writing a relevant low-level systems program in C
- to give you further experience using data structures in C

Getting Started

Create a new directory for this assignment called `allocator`, change to this directory, and fetch the provided code by running these commands:

```
$ mkdir -m 700 allocator
$ cd allocator
$ 1092 fetch allocator
```

This will add the following files into the directory:

- **`allocator.c`**: contains starter code. You add your code to this file and only this.
  Change only this file. You submit only allocator.c.
- **`allocator.h`**: defines prototypes for key functions. Do not change this file.
- **`test1.c`**: test `init_heap` function in `allocator.c`.
- **`test2.c`**: test of all functions in `allocator.c`.
- **`test3.c`**: test harness which makes calls to functions in `allocator.c` specified by lines on `stdin`
- **`test3_example_data.txt`**: example stdin for `test3.c`.
- **`test4.c`**: a test harness using a reallocating vector library, controlled by lines on `stdin`.
- **`Makefile`**: a specification to allows *make(1) to be used to compile the assignment.*

Background Information

Data structures such as fixed-size arrays with maximum sizes set at compile-time are ineffcient and brittle.

The C standard library provides the functions *malloc* and *free*, which allow programs to dynamically obtain and release the memory. This allows programs to, at any time, use precisely the memory needed to store their data. This more efficient and more robust.

In this assignment, you will write functions **my_malloc()** and **my_free()** which manage memory similarly to *malloc* and *free*

A top-level allocator such as *malloc* requests the memory it manages from the operating system on Unix-like systems this is via a system call, *sbrk*.

In this assignment you will request the memory **my_malloc()** and **my_free()** manage by calling *malloc*.

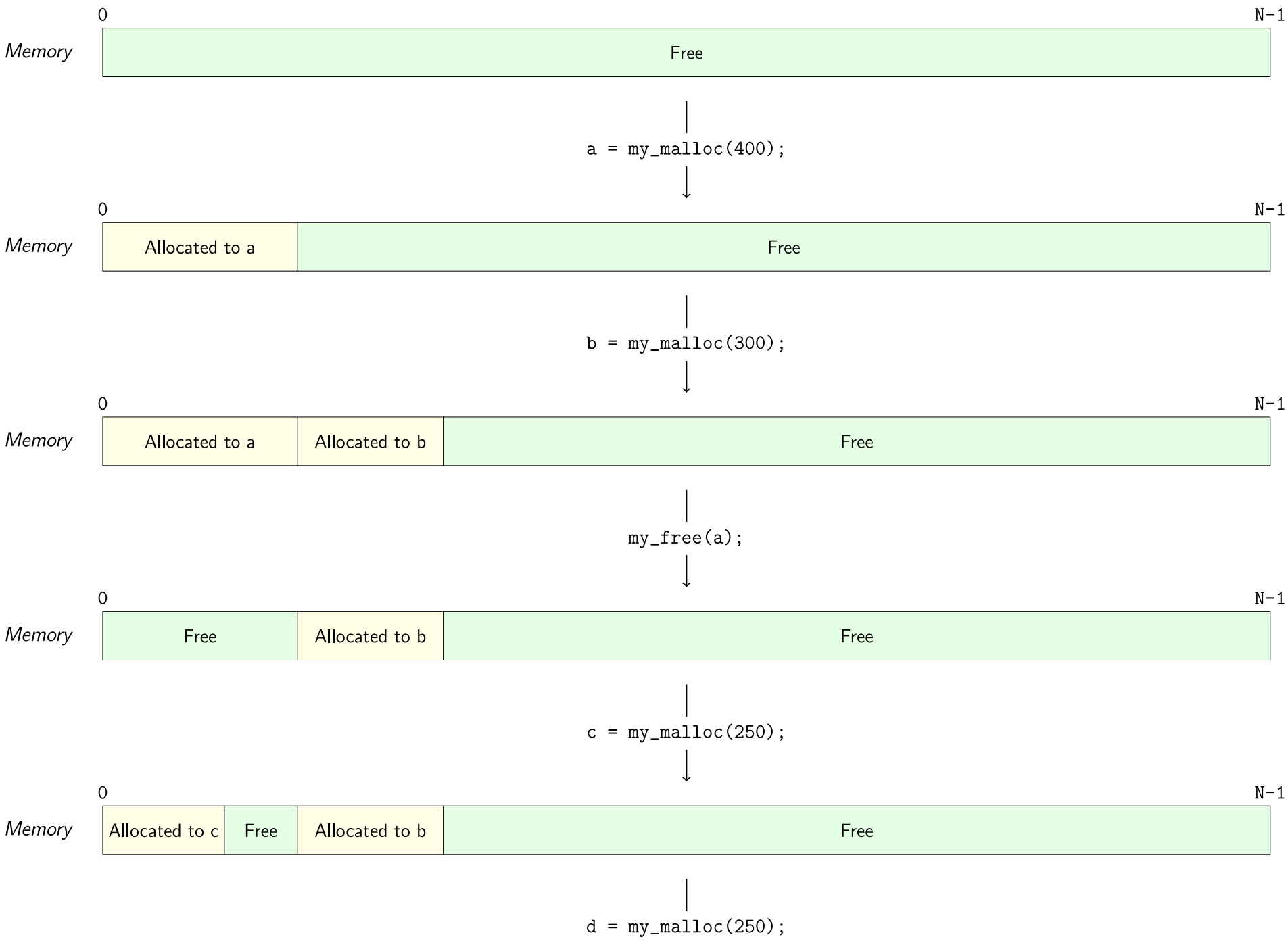
This makes **my_malloc()** and **my_free()** a *sub-allocator*,

Sub-allocators are often used in large applications that need certain guarantees (usually of performance) about the way memory allocation works, or which allocate lots of complexly-linked objects that can all be released at the same time: compilers, graphics engines, and web browsers all use sub-allocators to improve performance.

We call the large region of memory managed by a memory allocator the *heap. Your functions will manage smaller chunks of memory within their heap. There are two types of chunks: free chunks and allocated chunks*.

We'll describe the precise algorithms you must implement to determine which chunks to allocate and free (and merge) below, but first, a few diagrams showing what you need to do.

The following is an example of how some simple calls to `my_malloc()`'s and `my_free()`'s should manage the *heap*:



allocator.c: The Assignment

Your task in the assignment is to complete three functions in `allocator.c`:

- `init_heap()`,
- `my_malloc()`
- `my_free()`

There are three functions already completed in `allocator.c`:

- `free_heap()`,
- `dump_heap()`
- `heap_offset()`

- `heap_offset()`

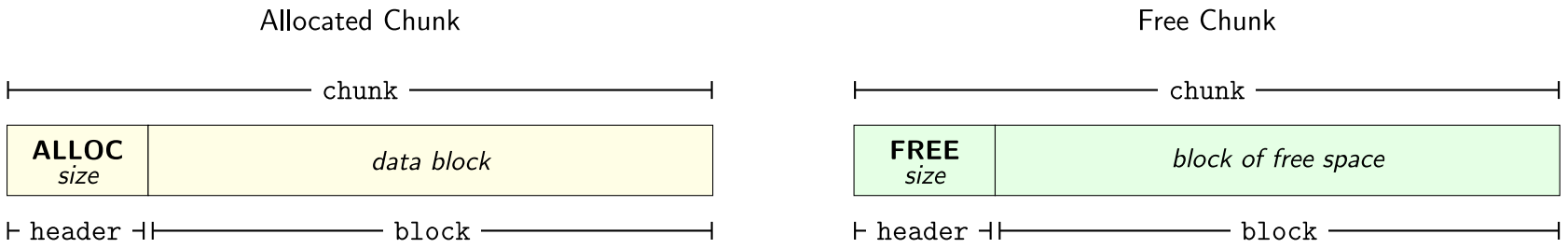
Do not change these functions.

You may wish to call `dump_heap()` and `heap_offset()` when debugging your code.

You can add as many other functions as you like to `allocator.c`: define them as `static` to make them private.

The Heap

The heap, for our purposes, is a large region of memory that is composed of free and allocated chunks. Each chunk has a header, containing a status (`FREE` or `ALLOC`) and the size of the entire chunk (including the header). The rest of the chunk after the header is a data block. The following diagram shows the two different types of chunks:



Alongside the heap region is an array of pointers to the free chunks in the heap. Its usage is described in more detail below.

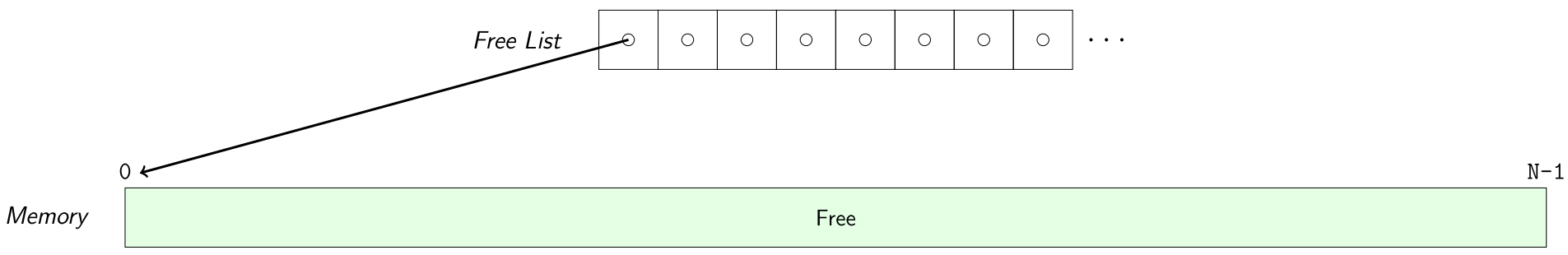
By necessity, the heap is a global variable. The `my_heap` variable is a `struct`, which groups together all of the heap's state.

`int init_heap(uint32_t size)`

This function allocates a region of memory of `size` bytes, in which all of our allocations will happen. If `size` is less than the minimum heap size (4096), then `size` is set to the minimum heap size. The value of `size` is also rounded up to the nearest multiple of 4 (e.g., 5001 would be rounded up to 5004).

The function sets `my_heap.heap_mem` to point to the first byte of the allocated region. It then initialises the region to be a single large free-space chunk. Finally, the function allocates a `free_list` array, of size `size`/`HEADER_SIZE`, containing pointers to the free chunks in `heap_mem`, and sets the first item in this array to the single free-space chunk. If it is successfully able to do all of the above, `init_heap()` returns 0; otherwise, it returns -1.

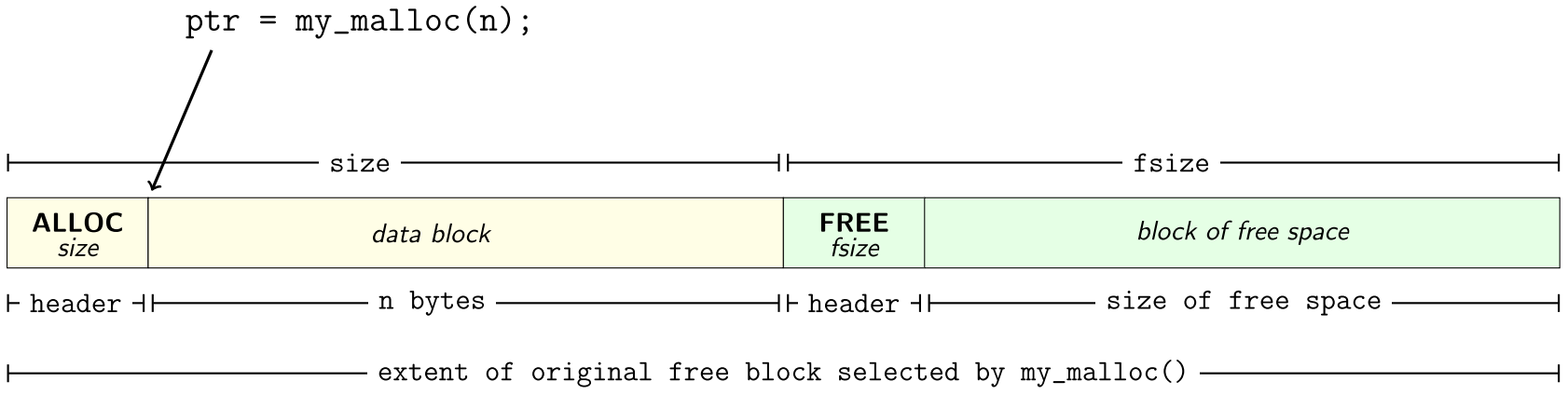
The initial state of the heap is as shown below:



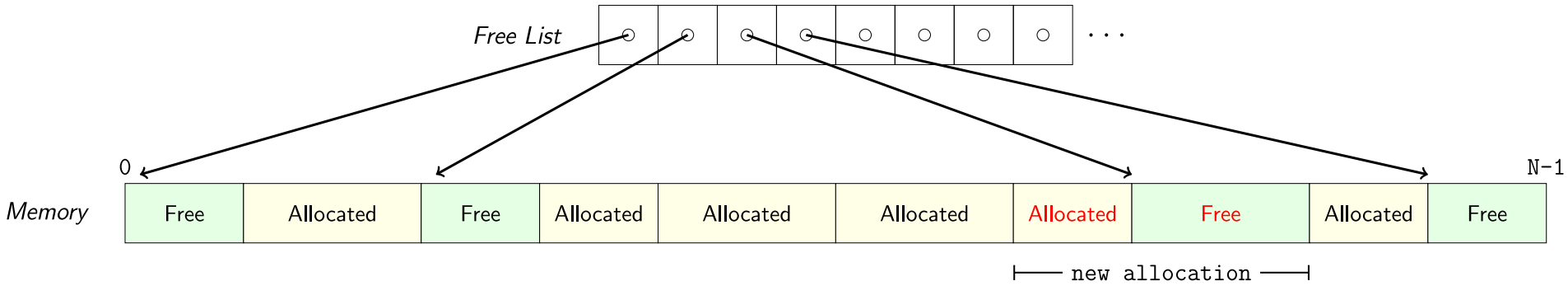
`void *my_malloc(uint32_t size)`

The `my_malloc()` function finds the smallest free chunk larger than or equal to `size + HEADER_SIZE`. If the free chunk is smaller than `size + HEADER_SIZE + MIN_CHUNK_SPLIT`, allocate the whole chunk. Otherwise, split it into two chunks, with the lower chunk allocated for the request, and the upper chunk being a new free chunk. If there are multiple chunks of the same size, the one closest to the beginning of the heap must be chosen to allocate.

The function returns a pointer to the first usable byte of data in the chunk as in the diagram below:



If a new free chunk is created, it is inserted into the appropriate location in the free list — the pointers in the free list should be maintained in **ascending** address order — as shown below:



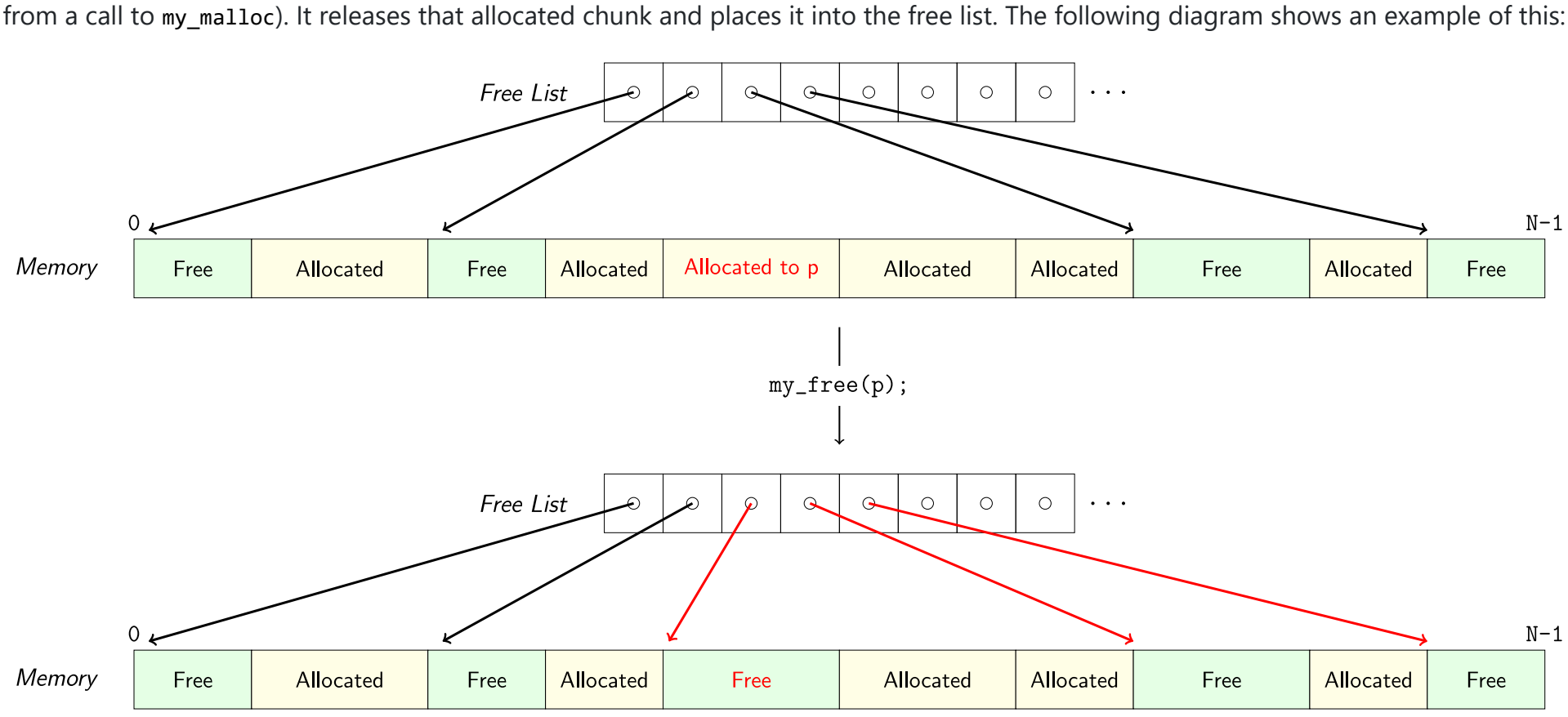
WARNING:

If `my_malloc()` is called with a value less than 1, it should return `NULL`.

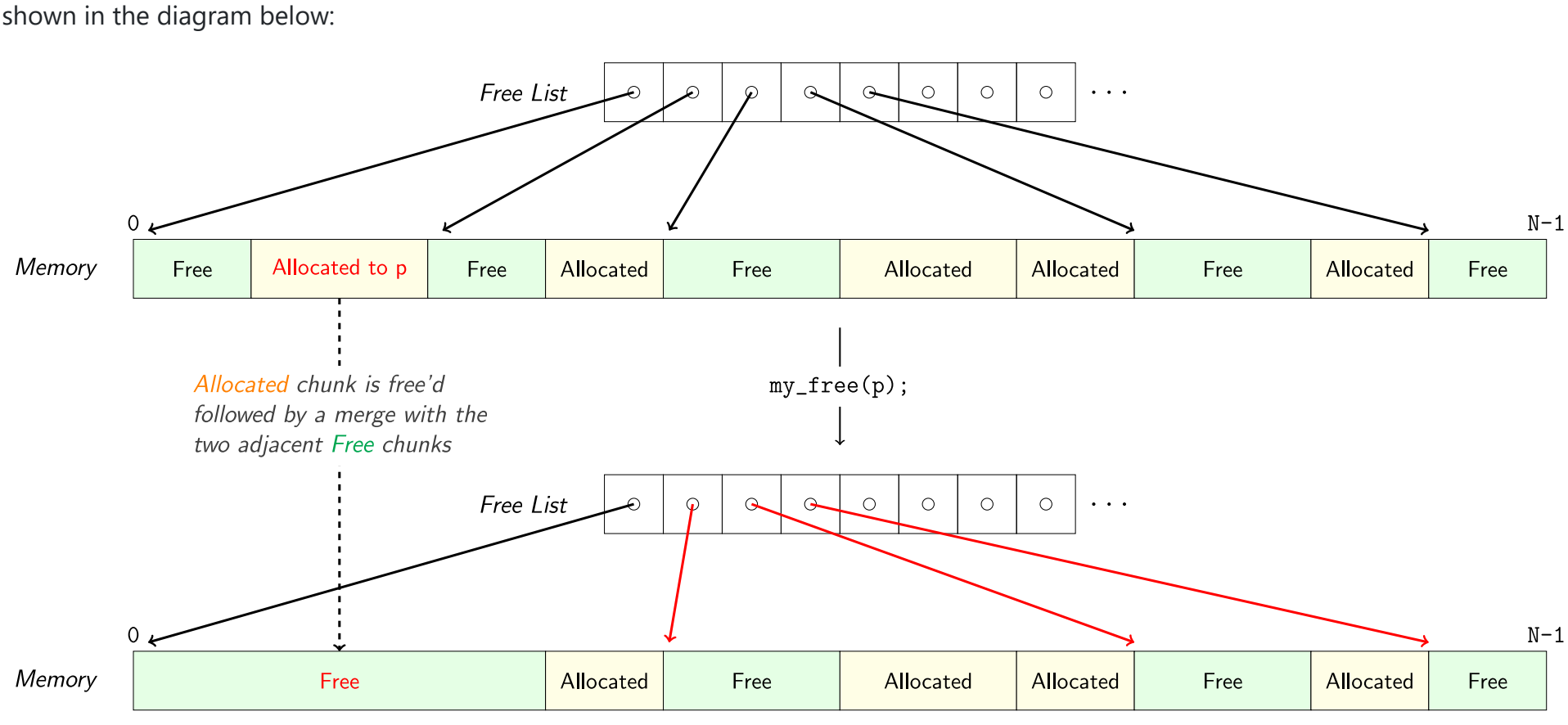
If `size` is not a multiple of 4, it should be increased to the next multiple of 4 (e.g., `my_malloc(14)` behaves as if the call had been `my_malloc(16)`).

`void my_free(void *ptr)`

The `my_free()` function is given a pointer to the first usable byte of data in a currently allocated chunk (i.e. a pointer previously returned from a call to `my_malloc`). It releases that allocated chunk and places it into the free list. The following diagram shows an example of this:



However: just updating the chunk status to `FREE` will lead to *fragmentation*, and future calls to `my_malloc` may find it harder (or even impossible) to find an appropriately-sized block. Instead, we avoid having adjacent free chunks: the free chunk(s) adjacent to the new free chunk are merged into a single larger free chunk. This could involve just two chunks being merged, or could involve three chunks as shown in the diagram below:



Note: if the argument to `my_free()` does not represent an allocated chunk in the heap, the program should print the message "`Attempt to free unallocated chunk\n`" to `stderr`, and then invoke `exit(1)`. This behaviour also applies if the argument is an address

somewhere in the middle of an allocated block; i.e., the only valid argument to `my_free()` is a pointer to the start of the data block in an allocated chunk.

Testing

Output from `test1`

The `test1.c` program simply initialises the heap to whatever size the user requests, and then dumps the contents of the heap. It should contain a single free block whose size is whatever the heap was initialised with.

If the size was less than *4096* then the size should be *4096*. As this is the minimum value allowed by `init_heap()`.

For example:

```
$ ./test1 5000
chunk 0: status = free, size = 5000 bytes, offset from heap start = 0 bytes
$ ./test1 10000
chunk 0: status = free, size = 10000 bytes, offset from heap start = 0 bytes
$ ./test1 100
chunk 0: status = free, size = 4096 bytes, offset from heap start = 0 bytes
$ ./test1 5001
chunk 0: status = free, size = 5004 bytes, offset from heap start = 0 bytes
```

The output from `heap_dump()` is a sequence of triples, where each triple represents one chunk and is of the form *+NNNNN (X, SSSSS)*, where *NNNNN* is the *offset* of the chunk from the start of the heap, *X* is either `F` for "free" or `A` for "allocated", and the *SSSSS* is the size of the chunk.

Output from `test2`

The `test2.c` implements a sorted linked list, which it fills from a sequence of 100 random numbers (since it uses the same seed each time, it will always generate the same sequence). It prints the list and dumps the heap after each insert, and prints the final heap after the list is freed. Since it generates a lot of output (potentially useful for debugging), we only show the first few inserts and the final state of the list and heap. It uses a 10000-byte heap, so after the list is freed, there should be a single 10000-byte free chunk.

Note that although you are free to, you are not *expected* to modify this code. Therefore, although `test2.c` uses linked-lists internally, it is not required (nor useful) to understand linked-lists for this assignment.

```
$ ./test2
heap size = 10000 bytes
maximum free chunks = 1250
currently free chunks = 1
chunk 0: status = free, size = 10000 bytes, offset from heap start = 0 bytes
L = 83
+00000 (A,   24) +00024 (F, 9976)
L = 83->86
+00000 (A,   24) +00024 (A,   24) +00048 (F, 9952)
L = 77->83->86
+00000 (A,   24) +00024 (A,   24) +00048 (A,   24) +00072 (F, 9928)
L = 15->77->83->86
+00000 (A,   24) +00024 (A,   24) +00048 (A,   24) +00072 (A,   24) +00096 (F, 9904)
... lots of output elided ...
L = 2->3->5->5->8->11->11->12->13->13->14->15->15->19->21->21->22->23->24->24->25->26
↳->26->26->26->27->27->29->29->29->29->30->32->34->35->35->36->36->37->39->39->40->42
↳->43->45->46->49->50->51->54->56->56->57->58->59->60->62->62->62->63->64->67->67->67
↳->67->68->68->69->70->70->72->73->73->76->76->77->78->80->81->82->82->83->84->84->84
↳->86->86->86->87->88->90->91->92->93->93->94->95->96->98->99
+00000 (A,   24) +00024 (A,   24) +00048 (A,   24) +00072 (A,   24) +00096 (A,   24)
+00120 (A,   24) +00144 (A,   24) +00168 (A,   24) +00192 (A,   24) +00216 (A,   24)
+00240 (A,   24) +00264 (A,   24) +00288 (A,   24) +00312 (A,   24) +00336 (A,   24)
+00360 (A,   24) +00384 (A,   24) +00408 (A,   24) +00432 (A,   24) +00456 (A,   24)
+00480 (A,   24) +00504 (A,   24) +00528 (A,   24) +00552 (A,   24) +00576 (A,   24)
+00600 (A,   24) +00624 (A,   24) +00648 (A,   24) +00672 (A,   24) +00696 (A,   24)
+00720 (A,   24) +00744 (A,   24) +00768 (A,   24) +00792 (A,   24) +00816 (A,   24)
+00840 (A,   24) +00864 (A,   24) +00888 (A,   24) +00912 (A,   24) +00936 (A,   24)
+00960 (A,   24) +00984 (A,   24) +01008 (A,   24) +01032 (A,   24) +01056 (A,   24)
+01080 (A,   24) +01104 (A,   24) +01128 (A,   24) +01152 (A,   24) +01176 (A,   24)
+01200 (A,   24) +01224 (A,   24) +01248 (A,   24) +01272 (A,   24) +01296 (A,   24)
+01320 (A,   24) +01344 (A,   24) +01368 (A,   24) +01392 (A,   24) +01416 (A,   24)
+01440 (A,   24) +01464 (A,   24) +01488 (A,   24) +01512 (A,   24) +01536 (A,   24)
+01560 (A,   24) +01584 (A,   24) +01608 (A,   24) +01632 (A,   24) +01656 (A,   24)
+01680 (A,   24) +01704 (A,   24) +01728 (A,   24) +01752 (A,   24) +01776 (A,   24)
+01800 (A,   24) +01824 (A,   24) +01848 (A,   24) +01872 (A,   24) +01896 (A,   24)
+01920 (A,   24) +01944 (A,   24) +01968 (A,   24) +01992 (A,   24) +02016 (A,   24)
+02040 (A,   24) +02064 (A,   24) +02088 (A,   24) +02112 (A,   24) +02136 (A,   24)
+02160 (A,   24) +02184 (A,   24) +02208 (A,   24) +02232 (A,   24) +02256 (A,   24)
+02280 (A,   24) +02304 (A,   24) +02328 (A,   24) +02352 (A,   24) +02376 (A,   24)
+02400 (F, 7600)
After free_list ...
chunk 0: status = free, size = 10000 bytes, offset from heap start = 0 bytes
```

Output from `test3`

The output from the `test3.c` depends on the input. It takes lines of the form:

- `a = allocate size`

  where *a* can be any character between `a` and `z` inclusive, and attempts to allocate *size* bytes; and

- `free a`

  where *a* can be any character between `a` and `z` inclusive, and attempts to free *a*.

`test3` also dumps the locations of the allocated variables (as heap offsets), and dumps the contents of the heap, to help you debug. Here is the correct output, using the supplied `data` file:

```
$ ./test3 10000 < test3_example_data.txt
heap size = 10000 bytes
maximum free chunks = 1250
currently free chunks = 1
chunk 0: status = free, size = 10000 bytes, offset from heap start = 0 bytes

a = allocate(100);
variable a is at heap offset     8
chunk 0: status = allocated, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = free, size = 9892 bytes, offset from heap start = 108 bytes

b = allocate(200);
variable a is at heap offset     8
variable b is at heap offset   116
chunk 0: status = allocated, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 208 bytes, offset from heap start = 108 bytes
chunk 2: status = free, size = 9684 bytes, offset from heap start = 316 bytes

c = allocate(300);
variable a is at heap offset     8
variable b is at heap offset   116
variable c is at heap offset   324
chunk 0: status = allocated, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 208 bytes, offset from heap start = 108 bytes
chunk 2: status = allocated, size = 308 bytes, offset from heap start = 316 bytes
chunk 3: status = free, size = 9376 bytes, offset from heap start = 624 bytes

free(b);
variable a is at heap offset     8
variable b is at heap offset   116
variable c is at heap offset   324
chunk 0: status = allocated, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = free, size = 208 bytes, offset from heap start = 108 bytes
chunk 2: status = allocated, size = 308 bytes, offset from heap start = 316 bytes
chunk 3: status = free, size = 9376 bytes, offset from heap start = 624 bytes

d = allocate(200);
variable a is at heap offset     8
variable b is at heap offset   116
variable c is at heap offset   324
variable d is at heap offset   116
chunk 0: status = allocated, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 208 bytes, offset from heap start = 108 bytes
chunk 2: status = allocated, size = 308 bytes, offset from heap start = 316 bytes
chunk 3: status = free, size = 9376 bytes, offset from heap start = 624 bytes

free(c);
variable a is at heap offset     8
variable b is at heap offset   116
variable c is at heap offset   324
variable d is at heap offset   116
chunk 0: status = allocated, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 208 bytes, offset from heap start = 108 bytes
chunk 2: status = free, size = 9684 bytes, offset from heap start = 316 bytes

free(a);
variable a is at heap offset     8
variable b is at heap offset   116
variable c is at heap offset   324
variable d is at heap offset   116
chunk 0: status = free, size = 108 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 208 bytes, offset from heap start = 108 bytes
chunk 2: status = free, size = 9684 bytes, offset from heap start = 316 bytes

free(b);
variable a is at heap offset 8
variable b is at heap offset 116
variable c is at heap offset 324
variable d is at heap offset 116
chunk 0: status = free, size = 10000 bytes, offset from heap start = 0 bytes
```

```
on exit:
heap size = 10000 bytes
maximum free chunks = 1250
currently free chunks = 1
chunk 0: status = free, size = 10000 bytes, offset from heap start = 0 bytes
```

Reminder: the sizes of the chunks are always eight bytes more than the requested amount of memory, because they include the 8-byte header.

You should design some inputs that test various allocation, free, and merge scenarios.

Draw them on a diagram first, then write the input file, and then run it with `test3` to see whether it produces the expected results.

Output from `test4`

The output from the `test4.c` depends on the input. It takes lines of the form:

- `push value [n]`

  where *`value`* is an integer, and *`[n]`* is an optional positive integer, which attempts to append the *`value`*, *`n`* times to the vector; and
- `pop [n]`

  where *`n`* is an optional positive integer (defaulting to 1), which attempts to remove and print the final *`n`* elements from the vector.

`test4` also dumps the current values in the vector, and dumps the contents of the heap, to help you debug. Here is the correct output, using the supplied `data` file:

```
$ ./test4 1000 < test4_example_data.txt
pushed 1, 1 times
vector: [1]
chunk 0: status = allocated, size = 72 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 40 bytes, offset from heap start = 72 bytes
chunk 2: status = free, size = 3984 bytes, offset from heap start = 112 bytes
pushed 2, 1 times
vector: [1, 2]
chunk 0: status = allocated, size = 72 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 40 bytes, offset from heap start = 72 bytes
chunk 2: status = free, size = 3984 bytes, offset from heap start = 112 bytes
pushed 3, 5 times
vector: [1, 2, 3, 3, 3, 3, 3]
chunk 0: status = allocated, size = 72 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 40 bytes, offset from heap start = 72 bytes
chunk 2: status = free, size = 3984 bytes, offset from heap start = 112 bytes
popped 3
popped 3
popped 3
popped 3
popped 3
vector: [1, 2]
chunk 0: status = allocated, size = 72 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 40 bytes, offset from heap start = 72 bytes
chunk 2: status = free, size = 3984 bytes, offset from heap start = 112 bytes
popped 2
popped 1
vector: []
chunk 0: status = allocated, size = 72 bytes, offset from heap start = 0 bytes
chunk 1: status = allocated, size = 40 bytes, offset from heap start = 72 bytes
chunk 2: status = free, size = 3984 bytes, offset from heap start = 112 bytes
freeing vector (heap should be entirely free)
heap size = 4096 bytes
maximum free chunks = 512
currently free chunks = 1
chunk 0: status = free, size = 4096 bytes, offset from heap start = 0 bytes
```

> **NOTE:**
>
> You may only call C library allocation functions
> (such as *malloc* or *calloc*)
> in `init_heap()`.
>
> You may not use any global variables.
> With the exception of `static struct heap my_heap;`,
> which has been provided for you

which has been provided for you.

Assumptions and Clarifications

Like all good programmers, you should make as few assumptions as possible.

Your submitted code must be hand-written C code, which you yourself have written:
You may not submit code in other languages.
You may not submit compiled output.
You may not submit code created by code synthesis tools.
You may not copy a solution from an online source (e.g., Github).

If you need clarification on what you can and cannot use or do for this assignment, ask in the class forum.

You are required to submit intermediate versions of your assignment.
See below for details.

Assessment

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1092 autotest allocator allocator.c [other .c or .h files]
```

Submission

When you are finished working on the assignment, you must submit your work by running `give`:

```
$ give dp1092 ass2_allocator allocator.c [other .c or .h files]
```

You must run `give` before **Week 12 Friday 23:59:59** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times.
Only your last submission will be marked.

You *cannot* obtain marks by e-mailing your code to tutors or lecturers.

You can check your latest submission on CSE servers with:

```
$ 1092 classrun -check ass2_allocator
```

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can collect your assignment by typing on the command line:

```
$ 1092 classrun -collect ass2_allocator
```

The resulting mark will also be available by typing:

```
$ 1092 classrun -sturec
```

Due Date

This assignment is tentatively due **Week 12 Friday 23:59:59**.

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 2%. For example, if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 15 hours late, it would be awarded 70%, the maximum mark it can achieve at that time.

Assessment Scheme

This assignment will contribute ***11 marks*** to your final DPST1092 mark.

80% of the marks for assignment 2 will come from the performance of your code on a large series of tests.

20% of the marks for assignment 2 will come from hand marking.
These marks will be awarded on the basis of clarity, commenting, elegance and style.
In other words, you will be assessed on how easy it is for a human to read and understand your program.

An indicative assessment scheme follows.
The lecturer may vary the assessment scheme after inspecting the assignment submissions,
but it is likely to be broadly similar to the following:

HD (85+)	beautiful, clearly-documented code; implements all behaviour perfectly
DN (75+)	very readable code; implements most behaviour perfectly

CR (65+)	readable code; some correctly-implemented behaviours
PS (50-60)	good progress, but not passing autotests
0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for DPST1092	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

Assignment Conditions

- **Joint work** is **not permitted** on this assignment

  This is an individual assignment. The work you submit must be entirely your own work: submission of work even partly written by any other person is not permitted.

  Do not request help from anyone other than the teaching staff of DPST1092 — for example, in the course forum, or in help sessions.

  Do not post your assignment code to the course forum. The teaching staff can view code you have recently submitted with give, or recently autotested.

  Assignment submissions are routinely examined both automatically and manually for work written by others.

  Rationale: this assignment is designed to develop the individual skills needed to produce an entire working program. Using code written by, or taken from, other people will stop you learning these skills. Other CSE courses focus on skills needed for working in a team.

- The use of **code-synthesis tools**, such as **GitHub Copilot**, is **not permitted** on this assignment.

  Rationale: this assignment is designed to develop your understanding of basic concepts. Using synthesis tools will stop you learning these fundamental concepts, which will significantly impact your ability to complete future courses.

- **Sharing, publishing, or distributing** your assignment work is **not permitted**.

  Do not provide or show your assignment work to any other person, other than the teaching staff of DPST1092. For example, do not message your work to friends.

  Do not publish your assignment code via the Internet. For example, do not place your assignment in a public GitHub repository.

  Rationale: by publishing or sharing your work, you are facilitating other students using your work. If other students find your assignment work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

- **Sharing, publishing, or distributing** your assignment work after the completion of DPST1092 is **not permitted**.

  For example, do not place your assignment in a public GitHub repository after this offering of DPST1092 is over.

  Rationale: DPST1092 may reuse assignment themes covering similar concepts and content. If students in future terms find your assignment work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

Violation of any of the above conditions may result in an academic integrity investigation, with possible penalties up to and including a mark of 0 in DPST1092, and exclusion from future studies at UNSW. For more information, read the UNSW Student Code, or contact your lecturer on teams.

Change Log

Version 1.0
(2022-06-23 09:00:00)

- Initial release.