Department of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 1, 2014

Assignment 2

Learning Outcomes

In this project you will demonstrate your understanding of dynamic memory allocation, data structures and file processing. You will also extend your skills in terms of program design, testing and debugging, and the skills of Assignment 1 on arrays, pointers, and functions.

The Story…

In *Foundations of Algorithms* you began your C journey with static memory allocation – you had to take care to ask for exactly what you would need ahead of time. But in return the C compiler hid much of the details of managing memory from you. You've since learned about dynamic memory allocation which grants you a lot more flexibility in asking for memory as you need it, but as a consequence you have to do more for yourself. Throughout this process, you've used `malloc()` to request a block of memory, `realloc()` to expand a previously-granted memory block, and `free()` to return control of a block of memory to the Operating System. In every programming language, including Python where memory handling is hidden, this process is running behind the scenes. For this second project, you'll build a rudimentary but fully-featured memory manager; something like the system that handles calls to `malloc()`, `realloc()` and `free()` in C.

To save you from going super low level, we'll make some simplifications. The key one will be that your "system's" memory will be stored in a static array. Around that will be infrastructure for tracking what has been allocated. These components are all collected in the global `mmanager_t`

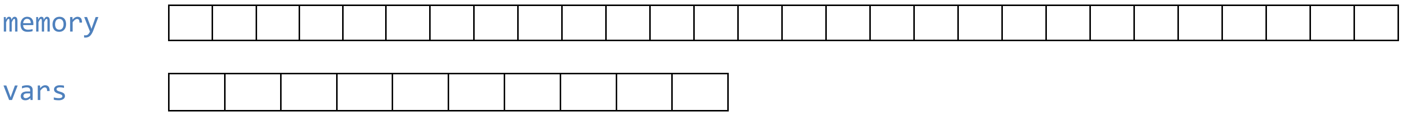
```
#define TOTALMEM   1048576
#define MAXVARS    1024

typedef struct {
      char memory[TOTALMEM];       /* TOTALMEM bytes of memory */
      void *null = memory;         /* first address is unusable */
      void *vars[MAXVARS];         /* MAXVARS variables, each at an address */
      size_t var_sizes[MAXVARS];   /* number of bytes per variable */
} mmanager_t;

mmanager_t manager;
```

The system has a blank canvas of `memory` from which to allocate memory dynamically. We've made this an array of `char` since each `char` is conveniently a byte. So storing (say) a 4-byte `int`, requires four elements of `memory`. We will reserve a `null` address which is never to be used - analogous to `NULL` but will be the first address in the pool. `vars` holds allocated addresses (the variables of the system), with the size of each allocated block in `var_sizes`.

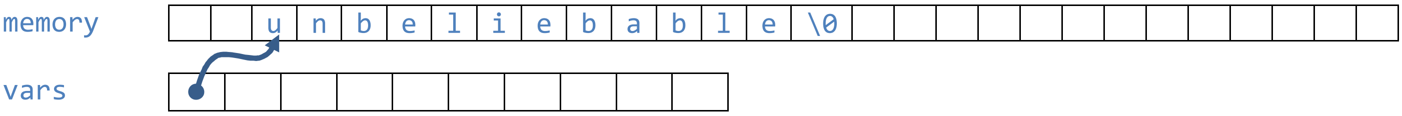
OMG WHAT ARE THOSE CRAZY TYPES!?!? `void*` means an address that is pointing to *some* data, but we don't know the type of that data. We can't dereference that address without casting to the correct type of the data. We'll see how to do that shortly. The variables tracked by `vars` could be any type, so these should be `void*`. In C it is conventional to use `size_t` for the size of things, for example this is the return type of `sizeof()`. You can think of `size_t` as being very similar to `unsigned int` (but it may have a different number of bits).

At this point you may feel confused –don't worry, the next example should help you picture what's going on*!!*

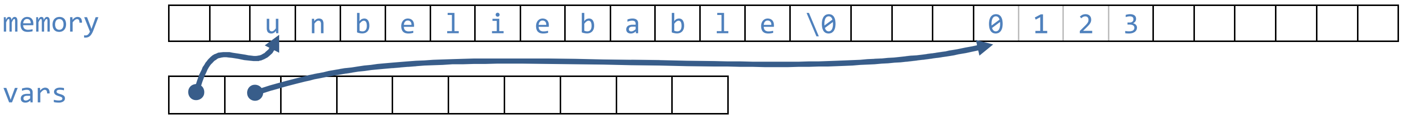
EXAMPLE. Here's a (partial) picture of the global `manager` at initialisation. There's nothing there, it's a blank canvas



Upon adding a first variable (with index in `vars` 0) of type string holding `"unbeliebable"`



To use this variable, we would cast as `(char*)vars[0]` to get a pointer to `char`. Why does `manager` need to track this variable with `vars`? So that when more memory is requested, we don't accidentally dole the same memory out twice! After adding a second variable at `1` holding a 4-byte integer `123`



The skeleton: proj2-skel.c

Your system should use (and extend if you need to) the `mmanager_t` to make available memory, manage memory as variables are allocated and freed, etc. This may sound ambitious, but fear not: while this project involves more challenging abstract thinking than Project 1, we've designed it to require less coding, and also have broken down the tasks much more in the Stages below! ***You can do it!!***

Obtain a copy of the skeleton file using the link on the LMS, and spend time studying it, including compiling &executing. Read through this project spec before starting, so that you can plan elements of your program design. Skeleton highlights

- The `main()` function is effectively written for you. It reads in lines from `stdin` that may be one of three commands: store characters (one or more as an array), store integers (one or more as an array), or free a previously stored variable. So essentially, the program has a dual personality:
  - On one hand you are writing a memory manager that can allocate and deallocate memory for any purpose;
  - On the other hand the part that is written allows you to *use* the memory manager to create and destroy variables and observe the result. See the sample text file that can be piped into the program for example syntax; supplied are the expected output of the skeleton (which is "wrong") and the expected output of your final solution. Each command comes in at some line number (first command coming in at line 1). The line number is sort of the created variable's name. Free command then refer to the variable (via its line number) to destroy. At the end of main the program prints the final state of all these variables.
- `mm_malloc()` has a very simple version implemented in the skeleton that you will be reworking. At the moment, it only supports ever managing one variable at a time! When more space is requested, it just throws away the previous variable.
- `process_free()` needs to be rewritten to use the `mm_free()` you will write below. Right now it ignores which variable to free up, because there's only currently ever one (because the manager currently only manages one!)

Stage 1 – Freeing space (marks up to 3/15)

To get started, implement the function to deallocate, or free, previously allocated memory in the system

```
void mm_free(void *ptr);
```

This function takes in a pointer (an address); the type of data being pointed to could be anything (e.g. an `int` or a `char`) and so the type is a pointer to `void`. Your function should

- Check to see if the address `ptr` has been previously allocated as the start of an allocated block. If not then the function should just return.

- If the address is being tracked, you should stop tracking it by appropriately updating `manager.vars[i]` and `manager.var_sizes[i]` for the appropriate `i`.

Remember to update code of `process_free()` now that you have a way to free any previously allocated memory.

Stage 2 – Anyone home? Checking for vacancy (marks up to 6/15)

As a building block towards implementing a version of the standard `malloc()`, implement function

```
int is_vacant(void *first, void *last);
```

which takes a contiguous range starting at address `first` and end at `last` (inclusive), and returns a logical `int` indicating whether the range is completely available or not. Remember, `0` means false and any non-zero `int` means true.

Hint: your implementation needs to check the range against all the variables allocated currently in `manager.vars` (where a variable is "allocated" if its address in `manager.vars` is not `manager.null` and the size allocated in `manager.var_sizes` is positive).

Hint: you'll want arithmetic on pointers for this stage: taking a start address and adding some number of bytes to it to get an end address. `manager.memory` is implemented as an array of characters, and adding numbers to `char*` means incrementing by the same number of bytes. However you can't add to pointers of type `void*` since the compiler doesn't know how much to add for whatever type you really have! To increment a `void*`, you need to first cast to `char*` then increment.

Stage 3 – Choosing an address and variable slot (marks up to 10/15)

Now that you can check whether a candidate contiguous block is available, it's time to: generate candidates, check them for availability using Stage 2, and make a choice. Implement a function

```
void *select_address(size_t size);
```

which takes as input an amount of space required (remember: `size_t` is just a kind of unsigned integer), and returns

- A start address, coming at least one after `manager.null`, pointing to a contiguous block of available memory of `size` bytes ending within `manager.memory`. Provided such a contiguous block is available.
- Otherwise returns `manager.null` to indicate out of memory.

Example: if `manager.memory` started at address X and went for 100 bytes (up to and including X+99) then a valid range for a request for 50 bytes could start any between X+1 and X+50 inclusive and ending (respectively) between X+50 and X+99 inclusive. The assigned range must not use the address X, nor go beyond X+99.

Next implement a function

```
int select_var(void);
```

which returns the earliest index into `manager.vars` to a variable slot that is not currently assigned (recall assigned means an address in `manager.vars` not `manager.null`; and a length in `manager.var_sizes` greater than zero).

Stage 4 – Putting it all together: Memory allocate (marks up to 12/15)

Hooray you're now ready to implement

```
void *mm_malloc(size_t size);
```

which mirrors `malloc()`. Given a requested number of bytes `size`, your function should

- Return a valid address within `manager.memory` that points to an available block of `size` bytes. In so-doing `mm_malloc()` is making this memory available, and so should track that this memory is now allocated by updating `manager.vars` and `manager.var_sizes` appropriately. Why? So that `mm_malloc()` doesn't accidently allow the same memory to be used twice, until it is freed up (with your Stage 1 code).

- If no such block is available, `mm_malloc()` should return `manager.null`

Stage 5 – Core dump (marks up to 15/15) [you can complete this stage out of sequence]

What should happen when a program crashes? A core dump, that's what! Essentially a copy of memory is written to disk, so that a post mortem can be done. Write a function

```
void core_dump(char *filename_mem, char *filename_vars);
```

which writes to disk

- The contents of `manager.memory` to the binary file with name `filename_mem`; and
- To text file named `filename_vars`, a line per allocated variable composed of: an integer offset of the variable's address from the start of `manager.memory` (that is, `0` for the address `manager.memory`, `1` for the next byte, and so on); a tab; then an integer corresponding to the size in `manager.var_sizes`.

Both files should be overwritten if present already. Modify `main()` so that core dump is called with arguments `"core_mem"` and `"core_vars"` respectively, at the end of execution of your program.

Stage 6 – Efficiency++ (***OPTIONAL BONUS***, marks up to 15/15)

As in Project 1, this final stage is optional (you can earn up to 2 marks back, not exceeding 15/15). Choosing an array for `manager.vars` (as done in the skeleton) was not the best decision. While your code works correctly, efficiency could be improved with a different data structure. Think about how `manager.vars` is used

- In determining whether a block is available or not, you search the allocated blocks in `manager.vars`
- When you find a place for requested memory, you assign the address by inserting into `manager.vars`
- When you free, you delete an item `manager.vars` (leaving it blank)

What structure might support more efficient searching (with insertion & deletion)? Try out your choice by modifying the structure used by `manager.vars` (and potentially packaging up `manager.var_sizes` in your structure too; since of course when you're inserting/deleting in `manager.vars` you're doing the same in `manager.var_sizes`). Document in your code via a brief comment, how your changes affect time/space complexity.

The boring stuff…

This project is worth 15% of your final mark. You don't have to do Stage 6 in order to get 15/15.

You need to submit your program named `proj2.c` for assessment via LMS. You can (and should) submit both early and often – to get used to the way it works, and ensure you have a submission in before the deadline. Only the last submission that you make before the deadline will be marked.

You may discuss your design/plans during your workshop, and with others in the class, but what gets typed into your program must be individual work, not from anyone else. So, do not give hard copy or soft copy of your work to anyone else; do not "lend" your memory stick to others; and do not ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say "are you out of your mind?" if they ask for a copy of, or to see, your program, pointing out that your refusal, and their acceptance of it, is the only thing that will preserve your friendship. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.

Deadline: Programs not submitted by 11:00pm on Sunday 1st of June will lose penalty marks of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email Ben.

And remember, algorithms are fun!