COMS10016 | Week 06-07 | Summative Coursework 01:

LIST CHALLENGE

This assignment runs over the next two weeks and counts 20% towards your credit for the unit. Its main purpose is to practice using pointers, strings, and bit operations. This time you must work individually - by doing this coursework you will make a significant step towards mastering fundamental parts of C. All code you submit must be ***your own***. Mastering the coursework yourself will enable you to progress in your learning and provide a foundation for many of the upcoming assessments and vivas you will have.

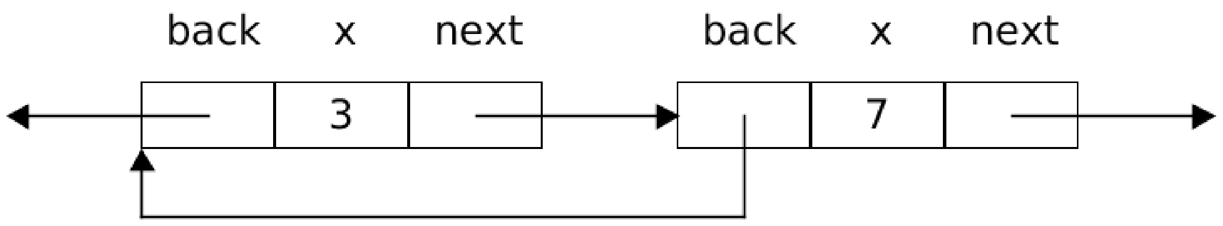
Use the Week 06/07 Channel on MS Teams to ask questions about this assignment to staff and TAs. Only ask your questions there, do not discuss the coursework in any other place. Do not share or paste code snippets or solution details in the channel or anywhere. We will answer your questions in this channel and help you along if there are questions re the assignment. There will also be dedicated hours where TAs are available on a Teams call during reading week and in the Week 07 imperative lab in MVB2.11 to provide help.

Do not copy or otherwise accept any code parts from peers or other sources and do not publish or make accessible parts of your own code anywhere. Just do the right thing for yourself. The programs we may use for checking against the web, your peers and other sources are advanced. Unethical conduct and plagiarism are unprofessional and may result in 0 marks for the coursework, the entire unit, or in repeated cases the forced end to your studies. Read the Notes on Plagiarism at the top of the unit website again.

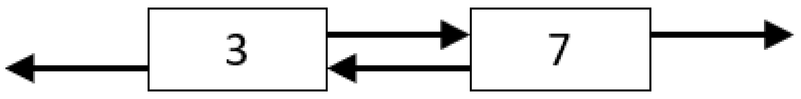
Use only standard libraries as given in the skeleton code for this task, so your code compiles and runs without linking any other libraries. Your task comes in two parts: a closed task that is worth the first 50% of your mark and an open-ended task. Backup your work regularly. Do not attempt the open-ended task before successfully and fully finishing the closed task.

Step 1: Understand Doubly-linked Lists

Before you start on this task make sure you watched all lectures up to Lecture 15. You should have compiled, run, and understood all the code provided for pointers, dynamic data, stacks, and lists. In particular, be sure you have run, compiled and understood in detail the program linkedlist.c from Lecture 15. Your task will evolve around doubly-linked lists with a sentinel node. Thus, let us understand and visualise this concept first. In essence, a doubly-linked list is made up of a sequence of nodes where neighbouring nodes point to each other. Each node is a structure that has a payload item `x` (e.g. just an `int`) and two pointers: `back` and `next`. The `back` pointer always points to the predecessor node and the `next` pointer always points to the successor node. Two neighbouring nodes in a doubly-linked list can therefore be pictured like this:

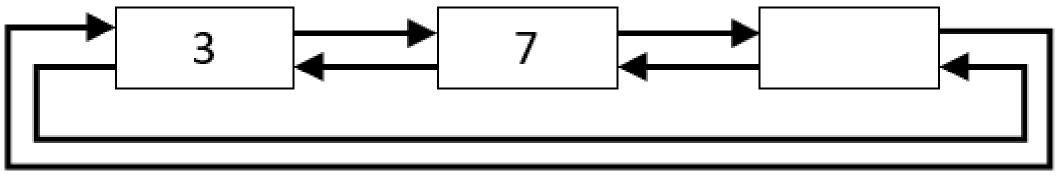


This emphasizes that a node structure contains three fields. However, for most purposes you can simplify the visualisation by depicting the above two nodes like this:

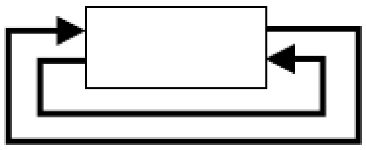


In this simplified visualisation, a pointer from the left end of a node represents its back pointer, and a pointer from the right end is its next pointer. A pointer to anywhere on a node's rectangle means a pointer to the start of the node.

The Sentinel Node. It used to be a standard solution to implement doubly-linked lists by keeping track of the first and last node of the list (like the 3 node and 7 node in the picture above), where the first node would point backward to NULL, and the last node would point forward to NULL. However, it turns out that adding an extra node, called a sentinel node, simplifies list implementations and makes the code much more readable. For a circular, doubly-linked list our sentinel node (pictured with no payload `x`) is linked in between the first and last item in the list:

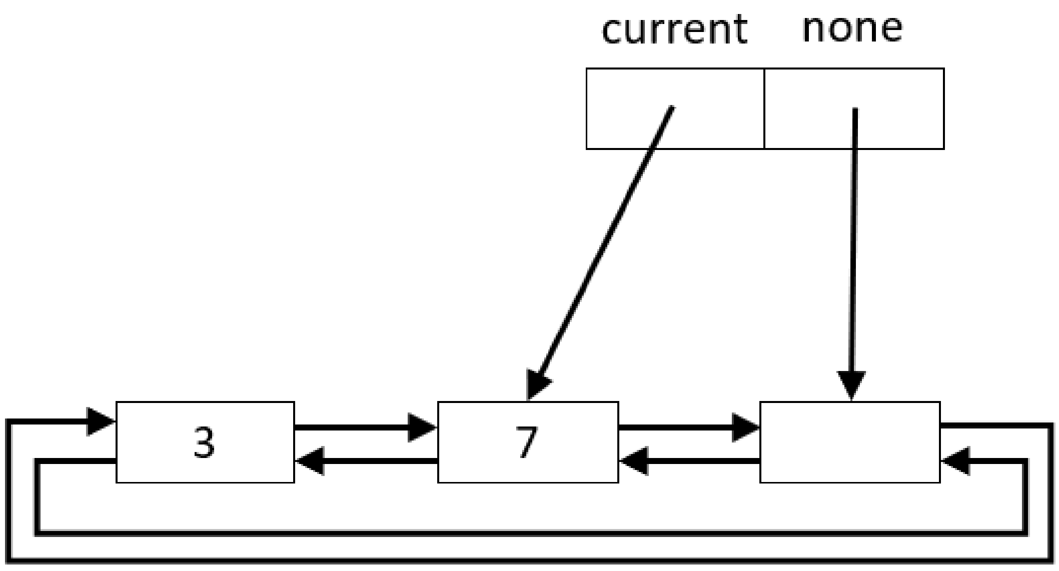


Using this idea, the nodes of a new 'empty' list with no item nodes look like this:

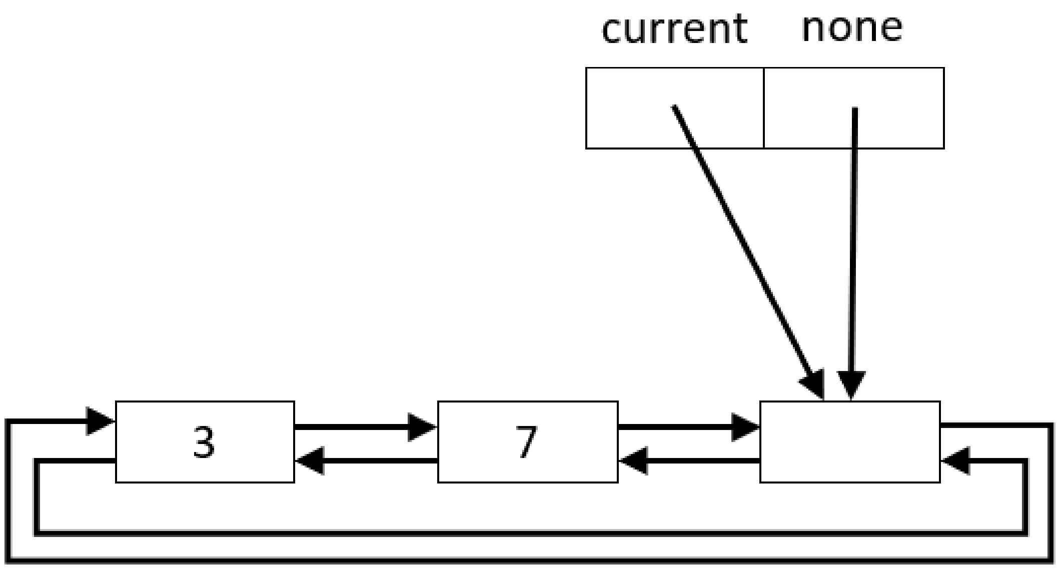


Here both the `back` and `next` pointers of the sentinel node simply point to the sentinel node itself.

The Structure `list`. To represent a list in a `list` data structure we need two node pointers: one fixed pointer to the sentinel node (called `none`) to access both list ends in constant time, and one `current` pointer that points to a current node in the list allowing for traversals:

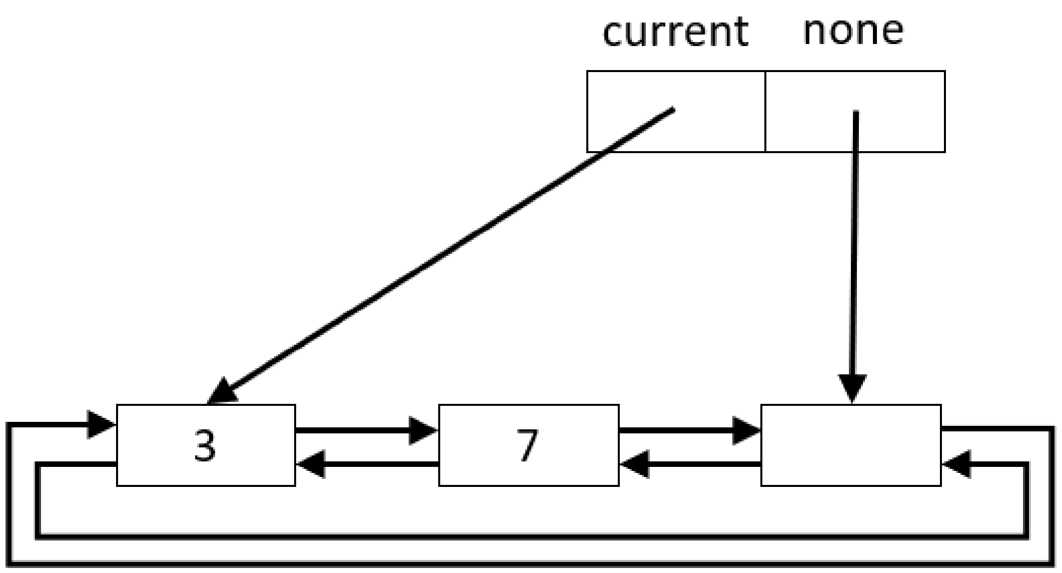


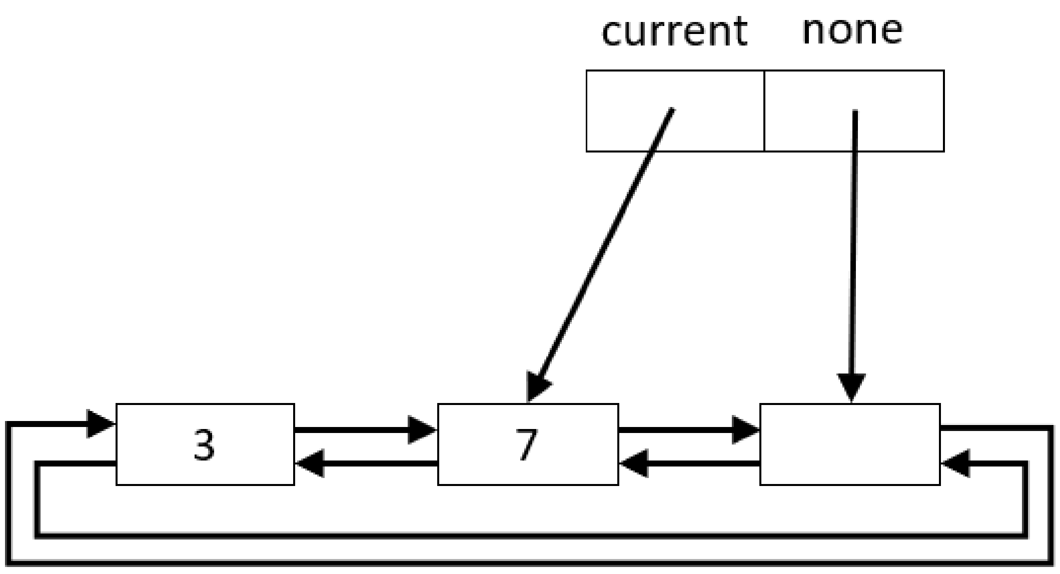
In the above image the `current` position is the node that holds 7, so item 7 is selected. If the `current` pointer in a list points to `none` then we will interpret this as 'no item is selected':



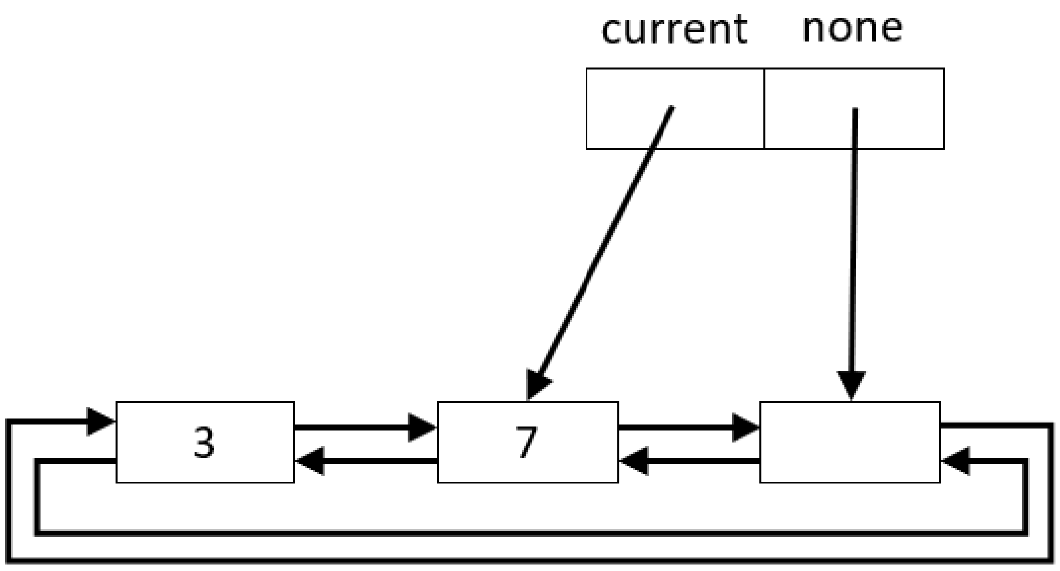
If there are item nodes in our list, `none->next` should link to the first item, and `none->back` should link to the last item. So we get simple access to both ends of the list and we do not need to store pointers to the first or last node anywhere else.

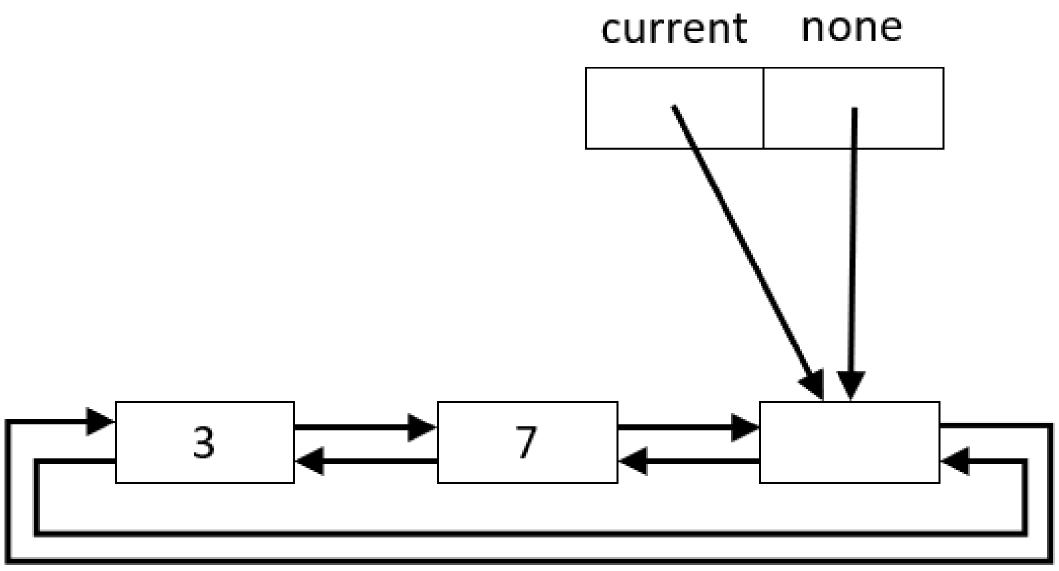
Picturing List Manipulations. To visualize what a function does to a `list`, draw a picture of the situation before the function call, and another of the situation after the call. For instance, consider the function `after`. If an item is selected it moves the `current` pointer one element forward. When applying the function to our list with the item 3 selected, it would have the simple effect of moving the `current` pointer one step forward to item 7:





Applying the `after` function to our list when the item 7 is selected moves the `current` pointer one step forward to the sentinel node, meaning 'no item' is selected after the call:





Thus, whenever in doubt, draw a picture of a particular situation before and after a function call to understand the detailed workings.

Step 2: Understand the Closed Task

Your closed task is to implement 13 missing procedures in the skeleton file list.c all of which manipulate circular doubly-linked lists with one sentinel node. The 13 missing procedures are described in detail in the header file list.h. You must use the provided files and are ***not allowed to alter any of the provided code***, only ***add*** the 13 missing functions where marked:

list.h (header file)
list.c (skeleton)
Makefile

The header file `lists.h` forms an API, which you should read ***carefully*** because the comments describe what the functions you will have to implement in `list.c` have to do. The `list.c` file has just two data structures `node` and `list` which you must use, and a lot of tests. The program as given to you will not compile initially. So, our first task is to produce a compiling skeleton by studying the signatures of the 13 missing procedures and accordingly defining some initial dummy functions.

Step 3: Create a Compiling Skeleton

For a start, download all above files into a folder of your development machine that only you have access to. After that, your first task is to turn `lists.c` into a full skeleton program that compiles. You can do this without understanding all of the technicalities of the assignment.

Two Key Data Structures. In the file `lists.c` a structure for the `node`s which make up the list is defined, and a structure for the `list` itself. The node structure `struct node` is not visible to the user of the module. Each node is used to hold an item `x` and pointers to the two neighbouring nodes (`next` and `back`) which define the list ordering. The overall list structure `struct list` represents a list and is essential so that your `newList` function can return something to the user which is well defined. This structure holds two pointers: one to the sentinel node of the list and one to the currently selected node. Read the code comments about the `list` structure carefully. ***You will have to use the two data structures exactly as described to comply with the tests.***

Define Dummy Functions. Write a minimal dummy definition of each of the 13 functions mentioned in the header file. The safest way to do that is to copy-and-paste a function's declaration from `lists.h`, then replace the semicolon by curly brackets. If the function returns anything other than `void`, add a return statement which returns the easiest temporary value of that type you can think of (e.g. `NULL` for a pointer, `false` for a boolean). For functions returning an item, you can return `0` for now, but beware that depends on `item` being `int`, so it may need to be fixed later. At this point, check that the program compiles fine via `make test` or directly via:

```
clang -Dtest_list -std=c11 -Wall -pedantic -g list.c -o list -
fsanitize=undefined -fsanitize=address
```

Pay attention to use the exact line for compilation, including that the parameter `-Dtest_list` must be used to run the tests.

Step 4: Understand the Tests

There is a separate test function for each of the 13 list functions you need to implement, except for `freeList` which can't be tested directly. The tests specify each function using before and after pictograms compressed into strings. Single digits represent items and the '|' symbol in front of a digit indicates that this is the current item. If the '|' symbol is at the end of the string then 'none' of the items is selected. The strings "|37", "3|7", "37|" represent a list of two items, with the current position at the first item, the last item, and a situation where 'none' of the items is selected. The tests utilise this pictogram string notation to drive the testing. For example, the one-line test for applying the `after` function when item 3 is selected in our example list will be encoded as:

```
    assert(__LINE__, check(After, -1, "|37", "3|7", true));
```

The one-line test for inserting 5 when the current item is 3 in our example list using the `insertAfter` function is:

```
    assert(__LINE__, check(InsertAfter, 5, "|37", "3|57"));
```

There is a different `check` function for each function type. The check function builds a list matching the before picture, calls the given function (in this case `insertAfter` with 5 as the second argument) and compares the result to the after picture.

Checks and Default. Most functions are designed to return a testable value. For example, if no item is selected, a call of `after` does nothing and returns `false`, which is easy to test. The `get` function returns an item in any case. To make sure there is an item which can be returned in any case, the `newList` function is passed a default item. The default item should be stored in the sentinel node.

Function Descriptions. What does each function do? There is a detailed comment for each function in the `list.h` header which gives a summary. For each function, there is a test function with some `assert` calls. These show precisely what the function does on the empty list `"|"` and a list with two items in at least each of the three cases `"|37"` and `"3|7"` and `"37|"`. That should be enough for you to work out what the function does in every possible case.

Details on Support Functions. The functions `build`, `destroy` and `match` form the heart of the testing and are implemented 'brute force'. The `build` function is used to build a list from the 'before' picture of a test, the function being tested is applied to the list, `match` is used to check that the result list matches the 'after' picture, and `destroy` frees up the list. Each of the functions uses an array of nodes in a very direct manner, so there is no ambiguity about what is going on. But that is not a technique you are supposed to be using in the list functions, because ***all of your 13 functions must take O(1) constant time***. The style of testing set up here is very carefully designed to allow you to work on one list function at a time.

Step 5: Write the 13 Functions One by One

Programming with pointers is difficult. When a test fails, there is generally a segfault or similar, which can be very difficult to track down. You will need to use several or maybe all of:

- the warning options `-Wall -pedantic`
- the `sanitize` options to pinpoint segfaults and memory leaks
- print statements

Develop `newList`. The first thing to do is to comment out all the tests except `testNewList` in `main`. After that, keep all the tests beyond the one you are working on commented out. That's because if a test fails, causing a segfault, it may be unreasonably difficult to know which test function caused it. Develop `newList` until it passes its test, ***and*** you don't get any messages from the various compiler options. In `newList` you will essentially have to allocate memory on the heap for a new list structure and a new sentinal node, initialise the sentinal node with the default item, let the current and none pointers in the list point to the sentinel node, and link the two pointers in the sentinel node point to the sentinal node itself.

Develop `freeList`. For all the functions, the compiler options test things that the tests themselves can't. In the case of `freeList`, there is no explicit testing that can be done. Therefore the ***only*** testing is that memory leak detection does not give any messages. Your `freeList` procedure should first free all nodes of the list including the sentinel node and finally free the list structure itself.

Develop in Small Steps. You may want to stick to the development sequence given by the test sequence for the functions. Thus, step by step uncomment the call to its testing function first, develop and test. Remember, the more exceptions and different cases your code handles, the more liable it is to have bugs in, because there are more places for bugs to hide, and it is harder for you to see at a glance that the code is correct. You aren't being given much opportunity for making your own implementation decisions in this closed part of the assignment. That simplifies checking correctness, and allows us to help you more easily. It is very tempting to write lines of code like this, with lots of arrows:

```
current->back->next->...
```

The trouble is, this is very error-prone. The code may be written with a mental picture of where the nodes were at the start of the function, but one or more of the pointers used in the expression may have been changed already by the time this line is executed. Trouble can arise particularly when shuffling lines of code around. A line of code that used to work may suddenly no longer work. And it is possible to 'lose' a node altogether, because there are no pointers left pointing to it, and therefore no way to reach it.

Use Robust Strategies. In this assignment, the insert and delete functions are the most difficult ones. They involve handling three nodes, either a new 'middle' node being inserted between (up to) two existing ones, or one existing node being deleted and its (up to) two neighbours being linked up together to close the gap in the circular list. A good strategy is to set up three local pointer variables (e.g. `p`, `q` and `r` or whatever you like) for these three nodes at the beginning of a function, so that you can keep track of them no matter what changes are made to the pointers between them. Each line of code after that can then be written simply using only one arrow, and the order in which the lines of code are executed doen't matter, making the code much more robust.

Enjoy programming and make sure your code adheres to the C Programming Style Guide! As always use the labs and the Teams chat for help and feedback throughout the two weeks. Once your program compiles and runs without errors and warnings, and passes all the tests you will have gained the first 50% of this coursework's marks. ***Everybody should work hard to get to this point.***

Step 6: Notes on the Design

A List of Items. The header is set up to store `item` values in lists. In the header `item` is defined as `int` to provide an example case. However, `item` must be used as a synonym for `int` everywhere in your code, so that there is only one place in the header file where a change would need to be made to store some other type of items. This means the module can be used with different item types in different programs. For those who are interested, note that even this it is not truly generic since the setup cannot be used multiple times for different item types in the same program. There is no really satisfying way of making fully generic modules in C. It is recommended, as a last test before submitting, that you change the item type to `double`, to check that you haven't inadvertently assumed `int` anywhere. (The numbers used in the tests should still work as doubles.)

Conceptual Design. The header doesn't say that the list is to be doubly linked, nor to be circular, or use a sentinel node (comments in list.c do though). That's because a user of the module need not know or care, and the implementation could be changed to something completely different in a later version of the module, without any effect on programs that use it. On the other hand, the header does say that all the operations are constant time. This is a strong hint that the implementation does use a doubly linked list or something similar, because it is difficult to achieve constant time operations otherwise. The claim of constant time doesn't cover the vagueness in the time taken by `malloc` and `free` calls but, conventionally, memory management costs are considered separately from the 'logic' costs of operations. The function names use the camelCase convention, where capitals make the letters go up and down like the humps on a Bactrian camel. I should point out, for those who are interested, that including a current position in a list structure itself is not thread safe. A more thread-safe approach is to create a separate iterator object each time the list is traversed. However, that approach can still easily lead to 'concurrent modification' problems where the list structure is changed by one thread while another is traversing it. It is much safer to make sure that a list is owned by a single thread. You will also have noticed that we have to compile our `list.c` program with the `-Dtest_list` flag to enable the tests. As you will learn in later lectures, if we don't use this flag then, in our case, we compile our program as a module without a main function and the tests. The program then seizes to be a stand-alone program, but instead can become part of another program that just uses its functionality.

Step 7: The Open-ended Task

Only if your program passes all tests in the above closed task and you have time left, then you can do some extra open-ended work towards a mark above 50% in your own program called `visualise.c` on the following problem: Using only standard libraries, if any, write a program that visualises the bit structure of data types in C in binary when entered in decimal form. The program must take 1) a type, and 2) particular data of this type in decimal notation as command line arguments. It should check for input errors (and print the exact string "Input error." in this case, any other output will cost you marks). If there are no input errors, it should print the bit structure of the data in groups of a nibble and no other output before or after that. The below examples show the exact program runs with the correct output:

```
./visualise char 7
0000 0111
```

```
./visualise char -128
1000 0000
```

```
./visualise char 255
Input error.
```

```
./visualise char 08
Input error.
```

```
./visualise char -x0
Input error.
```

We recommend to keep things relatively simple at first, for instance, by starting with investigating just `char`. The knowledge you already gained in computer architecture may be handy. Most importantly for this unit, your program should contain detailed unit tests for all functions. These functions should be run if no command line parameters are provided, i.e.:

```
./visualise
All tests pass.
```

If you have time left, follow up with visualising the bit structure of `int`, `long`, `unsigned char`, `unsigned int` and `double` exactly in this development sequence. Example runs with correct outputs should look exactly like this:

```
./visualise unsigned char 255
1111 1111
```

```
./visualise int 10000000
0000 0000 1001 1000 1001 0110 1000 0000
```

```
./visualise double -1.25
1011 1111 1111 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0
```

You may need to do some research into interpreting the bit structure of floating point representations if you attempt to look into `double`. If you still have time left after that, extend your program (keeping all previous functionality intact) so that it shows the decimal value of data types in C when entered in binary form in groups of nibbles. The following examples show again the exact program runs and output expected:

```
./visualise char 1000 0000
-128
```

```
./visualise int 0000 0000 1001 1000 1001 0110 1000 000
Input error.
```

```
./visualise unsigned char 0000 0111
7
```

```
./visualise double 1011 1111 1111 0100 0000 0000 0000 0000 0000 0000 0000 00
-1.25
```

Use the exact development sequence as before, starting with binary to char, binary to int etc. Note that it is always clear from the structure of your input if you are converting to decimal or to binary since binary input has leading zeros and comes in chunks of a nibble. You can extend your program further (keeping all previous functionality intact) by allowing for structured input using semicolon separated and {} enclosed types such as:

```
./visualise {char\;int\;unsigned char} 7 10000000 255
0000 0111 0000 0000 1001 1000 1001 0110 1000 0000 1111 1111
```

```
./visualise {char\;int\;unsigned char} 0000 0111 0000 0000 1001 1000 1001 01
7 10000000 255
```

Your source file `visualise.c` must compile error-free and warning-free for a valid submission in any case. Your program MUST comply exactly with the specified format for input and output. You are not allowed to use any libraries other than standard libraries and `math.h` if needed - your code must compile without linking any other libraries. You are encouraged to write a summary file `readme.txt` which describes what your program can do in no more than 100 words (strict limit, longer readme.txt files will cost you marks). There are no marks for report writing, but the summary may be necessary for us to make sense of your program. As long as your program works, even if your program is very basic (e.g. just checks for `char` that some the input char is valid), still submit it. Whenever you have reached a well working version, take a copy of your work, so you can revert back to it at any point. Rather submit something simple that works bug-free and is well tested than something that contains bugs - you will not get many marks for buggy code at all. Be careful not to over spend on time, since the task is completely open-ended. Make sure you manage your time well and stop at an appropriate point. Make sure your programming adheres to the C Programming Style Guide. Enjoy programming, as always use the labs and the Teams chat for help and feedback throughout the two weeks!

A mark out of 50% for the extra work will be awarded by swiftly reading the summary, checking whether your program matches what you claim, judging the sophistication and extent of what has been done, and checking whether the program follows the conventions and advice given in the unit. In particular, writing tests as part of the program, in the same way as the skeletons we provide, is very much recommended. Also recommended is working in very small steps, one test at a time, keeping your program in a working state. Again, make sure you do manage your time well and stop at a reasonable point. Remember, there are significantly deminishing returns.

The mark will aim to make your total for the assignment meet the university scale. So assuming you get full marks for your `list.c` program, for the open-ended task 10/50 means "this raises your total result from good to very good", 15/50 means "a near excellent result", 20/50 means "excellent, overall above and beyond what was expected", 25/50 means "superb work for this stage of the unit", 30/50 means "truly exceptional work, potential mastery of the subject area" and 40/50 means "novel and publishable in a research journal as is".

Step 8: Submit

Submit your work via Blackboard under the submission point "List" for the 2021/22 unit COMS10016. A link for submission is available on the Blackboard Unit Website or directly here. It will be linked on the unit website. You are responsible for early submission, thus submit AT LEAST an hour before the deadline (deadline is 01:00pm UK time on 12/11/21) to make sure there are no upload problems. The university systems will automatically apply quite severe penalties if the coursework is late only by one second. You must submit ALL individual files (`list.c`, and if completed `visualise.c` and `readme.txt`) as attachments in a SINGLE submission, do not zip or compress your work. If you decide to resubmit you MUST submit ALL FILES AGAIN - only the files present in your last submission will be marked. If your last submission is late you will receive the associated mark penalty in any case.

Closed Task: Submit your program `list.c` (not `List.c` or any other name, unless you want to lose marks, and not the compiled program). Make sure your program compiles without warnings, without using any additional libraries, runs without errors, and doesn't still contain debugging print statements.

Open-ended Task: If you attempted the open-ended part, then also submit your extra program `visualise.c` and a `readme.txt` file with any comments you might want to make (max. 100 words). Again, make sure your program compiles without warnings, runs without errors, and doesn't still contain debugging print statements.

Again, all code you submit must be ***your own*** - this is the only way to learn. We reserve the right to spot-check suspicious cases where you have to explain 'your code' to us live and in detail.