

CS2106 Introduction to Operating Systems
Lab 1 - Leveling Up on C
Week of 25 August 2025 (Initial Release)
Week of 1 September 2024 (Demos)
Final Submission: 7 September 2024, 2359 hours

Introduction

This course assumes that you have basic knowledge of C programming, and this lab will build on those basics to introduce you to some intermediate and advanced topics, including dividing your work up into multiple source files, creating pointers to functions, and using dynamic memory to create arbitrarily large data structures (subject to available memory).

Instructions

This lab may be completed individually or with a partner. There are two components to this lab:

- a. A written component. You should fill all your answers in the file AxxxxxxY.docx, renaming it to your actual student number.
- b. If you are submitting with a partner, decide who will submit, and rename the file AxxxxxxY.docx to the student number of the person submitting. **ONLY ONE SUBMISSION IS REQUIRED.** The marks for both students in a team will be recorded in Canvas.
- c. All submissions will be made to Canvas. Each lab group has its own assignment submission folder.
- d. Please submit this lab by **Sunday, 7 September 2025, 2539 hours**. The folder will remain open until 0015 hours on 8 February 2025 latest, **after which NO SUBMISSIONS will be accepted, no matter how hard you've worked on it!**
- e. There are demos in Parts 1, 3 and 4 of this lab. The demos will take place on the week of **1 September 2025**. Demos are done **individually** with your respective TA even if you did the project with a partner. **This means attending your own lab session for the demos.** However, you may attend a common lab session in the first week to ask questions together.
- f. This lab is worth 20 marks; 14 marks for the written portion, 6 marks for the demos.

All questions are worth 1 mark. Tutors may assign ½ mark if there are errors, or 0 marks if the answer is completely wrong. Demos are worth 2 marks. You will get 0, 1 or 2 marks depending on the success of the demos and your ability to answer questions from the TA.

0. Uploading and Unpacking your Program Files

- a. Copy lab1programs.zip over to your home directory on xlog:

```
scp lab1programs.zip <userid>@xlog.comp.nus.edu.sg:~
```

- b. ssh to xlog:

```
ssh <userid>@xlog.comp.nus.edu.sg
```

- c. Create a directory called L01 and change to it.

```
mkdir L01; cd L01
```

- d. Get a job allocation:

```
salloc
```

- e. Unzip lab1programs.zip (which is presumably one directory level up in your home directory):

```
srunch unzip ../lab1programs
```

1. Let's Modularize!

Switch to the "part1" directory.

If you had taken CS2100 Computer Organization, you would have learnt to write all your C code in a single source file. This can be very undesirable in large projects that have hundreds of thousands of lines of code. In such projects it makes sense to break up our source code into many modules that group related functions together.

Here we will look at how to break your code up into modules and link them together, by creating a library to maintain a circular linked list in an array.

- a. Using your favorite editor, open "stack.c" and examine the code.

Question 1.1 (1 mark)

At the start of stack.c you will see:

```
#include <stdio.h>
#include "stack.h"
```

The `#include "stack.h"` statement looks for the `stack.h` file in the currently directory, where does the `#include <stdio.h>` statement look for the `stdio.h` file?

Question 1.2 (1 mark)

At the top of `stack.c`, you will notice that `_stack`, `_top` and are declared as "static". What does the "static" declaration mean?

- b. Using your favorite editor again, open "`stack.h`" and you will see just a single line:

```
// Maximum number of elements in a stack
//
#define MAX_STACK_SIZE 10
```

Open "`lab1p1.c`" and you will see:

```
int main() {
    int v;
    for(int i = 0; i<= MAX_STACK_SIZE; i++) {
        printf("Adding %d\n", i);
        push(i);
    }

    for(int i = 0; i<= MAX_STACK_SIZE; i++) {
        v = pop();
        printf("Element %d is %d\n", i, v);
    }
}
```

Notice how `MAX_STACK_SIZE` is brought in from `stack.h` via the `#include` statement. Notice also that unlike Python there is no concept of a namespace in C.

- c. If you are running on the SoC cluster, you need to get a job allocation first. Log in to `xlog.comp.nus.edu.sg`, then type:

```
salloc
```

The system will respond with:

```
salloc: Granted job allocation <allocation ID>
```

Where <allocation ID> is some identifier for your job.

- d. We will now compile our program. To do so, switch to the directory containing `stack.c`, `stack.h` and `lab1p1.c` and type (on slurm)

```
srslun gcc stack.c lab1p1.c -o lab1p1
```

On a normal Linux machine, simply type:

```
gcc stack.c lab1p1.c -o lab1p1
```

To run your program on slurm:

```
srslun ./lab1p1
```

On a normal Linux machine:

```
./lab1p1
```

(Note: When you run this program, you will notice that the last element cannot be inserted as the stack is full, and when reading the last element cannot be read because the stack is empty. This is normal and it's meant to test that the stack can detect full and empty conditions)

Question 1.3 (1 mark)

You may notice that you get the following error (**Note:** Some compiler versions may return a warning instead of an error, and some compilers may return neither a warning nor an error but fail silently instead (!!)):

```
lab1p1.c:8:3: error: call to undeclared function 'push'; ISO C99 and later do not support implicit function declarations [-Wimplicit-function-declaration]
  8 |     push(1);
    |     ^~~~~
lab1p1.c:12:7: error: call to undeclared function 'pop'; ISO C99 and later do not support implicit function declarations [-Wimplicit-function-declaration]
  12 |     v = pop();
    |         ^~~~
2 errors generated.
```

What causes this error (or warning)?

- e. We will now fix the warning/error in Question 1.3 by creating “function prototypes” in “`stack.h`”. Briefly, a function prototype specifies the name, number and types of parameters and return type of a function.

Let's suppose we have a function named “`proto_example`” that looks like this:

```
char proto_example(int x, float y, double z) {
```

```

        ... Body of function ...

    }

```

Then its prototype will look like this:

```
char proto_example(int, float, double);
```

Notice two things about the prototype:

- i. We do not need to specify the names of the parameters, only the types.
- ii. The prototype definition ends with a semicolon.

Function prototypes should always be defined before the function is called at any point in your C program. Since in lab1p1.c we will be calling push and pop inside main, we need to create their prototypes before main.

There are at least three ways to do this:

- i. Define the prototypes inside stack.c. This is just silly since we don't #include stack.c into lab1p1.c and thus main will never see the prototypes.
- ii. Define the prototypes inside lab1p1.c before main. This will work, but what if you also want to use stack.c elsewhere?
- iii. Define the prototypes inside stack.h, which will be #include into lab1p1.c (and anywhere else you want to use push and pop). Now THAT makes sense!

Question 1.4 (1 mark)

Copy and paste the prototypes for push and pop you defined in stack.h here.

- f. Finally let's work with function pointers. Function pointers are, as their name suggests, pointers to functions, and they're particularly useful in implementing "callbacks", also known as "delegates" in Objective-C, or "decorators" in Python (although the callback mechanisms in these two languages is completely different from C function pointers.)

Let's look first at this declaration:

```
int *func(int x) {
    ...
}
```

This function returns a **pointer to an int**, and takes an int as a parameter. Now look at this declaration:

```
int (*fptr)(int x);
```

Now this is a **pointer to a function that returns an “int”**, and takes an int as an argument. Notice that a function pointer looks similar to a prototype, except that it has an additional bracket around the function name.

Open “lab1p1a.c” and you will see some examples of how to use function pointers:

```
#include <stdio.h>

int (*fptr)(int);

int func(int x) {
    return 2 * x;
}

int y = 10;

int *(*pfptr)();

int *func2() {
    return &y;
}

int main() {
    printf("Calling func with value 6: %d\n", func(6));
    printf("Now setting fptr to point to func.\n");
    fptr = func;
    printf("Calling fptr with value 6: %d\n", fptr(6));

    printf("\nNow calling func2 which returns the address of global variable y: %p\n", func2());
    printf("Pointing pfptr to func2.\n");
    pfptr = func2;
    printf("Now calling pfptr: %p\n", pfptr());
}
```

Here we see “fptr” declared as a pointer to a function returning “int”, and “pfptr” as a pointer to a function returning int *. We also see two functions “func” and “func2”, one returning an int, and another returning an int *.

In main, we assign func to fptr and func2 to pfptr. Note that when we assign func to fptr and func2 to pfptr, we do:

```
fptr = func;
pfptr = func2;
```

And NOT:

```
fptr = &func;
pfptr = &func2;
```

This is because the name of a function is a pointer to the function, just like the name of an array is a pointer to the first element of the array.

We will now extend our “stack.c” and “stack.h” to build a reduce function. Open stack.c again and you will see some functions related to function pointers near the bottom of the file.

Function	Purpose
clear_xor(int *acc)	Clears the acc variable by setting it to 0.
clear_or(int *acc)	Clears the acc variable by setting it to 0.
int xor(int x, int y)	Returns $x \wedge y$
int or(int x, int y)	Returns $x \vee y$

reduce()	Calls clear_xor() to set the accumulator _res to 0, then calls xor to xor over all members of the stack and returns the result in res.
----------	--

Load up testr.c, and you will see some code that fills up the stack with 1 to 9, then a call to reduce() to sum up the contents of the stack, finally printing out the results. There are also two commented-out lines for you to test your flex_reduce function later.

Now do the following:

- i. Add in the prototypes for clear_xor, clear_or, xor, or and reduce to stack.h
- ii. Compile testr.c and stack.c using:

```
srun gcc testr.c stack.c -o testr
```

OR:

```
gcc testr.c stack.c -o testr
```

- iii. Run testr:

```
srun ./testr
```

OR:

```
./testr
```

NOTE: From this point onwards we will assume you are running on slurm and will show commands accordingly. On normal Linux systems simply leave out the srun command.

- iv. You will see an output like this:

```
Pushing 1
Pushing 2
Pushing 3
Pushing 4
Pushing 5
Pushing 6
Pushing 7
Pushing 8
Pushing 9
Calling reduce result is 1
```

- v. Now we want to create a new function called "flex_reduce" that takes in two arguments: clear and op, which are pointers to functions to clear _res, and to operate on the elements of the stack.
- vi. The pseudo-code for flex_reduce is shown here:

```

int flex_reduce(clear, op) {
    clear(&_res); // Clear _res to either 0 or 1
    for every element in stack:
        Call op with element and _res

    return _res;
}

```

Implement the `flex_reduce` function in `stack.c`, and add its prototype to `stack.h`.

Now uncomment the two statements in `testr.c` to test `flex_reduce`, and compile your program. You should see the following similar results:

```

Pushing 1
Pushing 2
Pushing 3
Pushing 4
Pushing 5
Pushing 6
Pushing 7
Pushing 8
Pushing 9

Calling reduce result is 1
Calling flex reduce with XOR. Result is 1
Calling flex reduce with OR. Result is 15

```

DEMO 1 – This demo is to be done at the lab on the week of 1 September 2025 (2 marks)

Run your code to show that `flex_reduce` works. Show the code to your TA, and answer any question that he or she might have.

2. Throw It On the Stack!

Switch to the “part2” directory.

In this section we will put aside modularizing C code for a while (we will return to it in section 3), and explore the lifetime of local variables, and what it means.

- a. Use your favorite editor and open `lab1p2a.c`. You will see:
 - i. Four integer pointer variables `p1` to `p4` declared as global variables.
 - ii. A function called “`fun1`” that has two parameters `x` and `y`, and two local variables `w` and `z`. It sets `p1` to `p4` to point to `w`, `x`, `y` and `z` respectively, and prints out the addresses of `p1` to `p4`, `w`, `x`, `y` and `z`, and their values.
 - iii. A function called `fun2` that takes 3 arguments `f`, `g` and `h`, and also prints out the values of `w`, `x`, `y` and `z` using pointers `p1` to `p4`.

Compile and run `lab1p2a` using:

```

srun gcc lab1p2a.c -o lab1p2a
srun ./lab1p2a

```

Question 2.1 (1 mark)

Record your observations in the tables below, writing “G” or “L” in the global or local column depending on whether the variable is a global or local one, and the address of the variables as shown when you run lab1p2a. (You can classify function parameters as “L” or “G” based on whether you think they are globally accessible or not.)

Variable	Global / Local	Address
p1		
p2		
p3		
p4		
w		
x		
y		
z		

Question 2.2 (1 mark)

Based on your answers to Question 2.1, Where do you think (stack, data, text or heap) each of these variables are located?

Variable	Location (S, D, T or H)
p1	
p2	
p3	
p4	
w	
x	
y	
z	

How did you infer these answers, from your answers to Question 2.1?

Question 2.3 (1 mark)

Notice that the contents of `x`, `y` and `z` may be changed (“corrupted”) when we exit `fun1` or when we call `fun2`. This is expected as `x`, `y` and `z` are local variables, and their life-span is only within `fun1`. Thus their values are not guaranteed to stay the same throughout the lifetime of the entire program.

However notice that `w` remains unchanged when we exit `fun1`, and even after we call `fun2`. This is because “`w`” is declared as “static”. Where are static variables created in memory and why does this allow them to preserve values between calls to a function?

Question 2.4 (1 mark)

In Question 1.2 we saw global variables being declared as “static”, and here we see a local variable “`w`” being declared as “static”. What does it mean to declare a local variable to be “static”, versus a global variable?

Use your favorite editor now to open `lab1p2b.c`. We see a function called “accumulate” that attempts to accumulate values passed to it. Meanwhile the for-loop in `main` passes in 1 to 10 to `accumulate`, which SHOULD produce the following result:

```
acc is now 1
acc is now 3
acc is now 6
acc is now 10
acc is now 15
acc is now 21
acc is now 28
acc is now 36
acc is now 45
acc is now 55
```

Now compile and run `lab1p2b.c` using:

```
srunc gcc lab1p2b.c -o lab1p2b
srunc ./lab1p2b
```

You will see we get a wrong result; `accumulate` doesn’t accumulate values passed to it instead just prints out these values:

```
acc is now 1
acc is now 2
acc is now 3
acc is now 4
acc is now 5
acc is now 6
acc is now 7
acc is now 8
acc is now 9
acc is now 10
```

Question 2.5 (1 mark)

Fix lab1p2b.c to produce the correct result, without declaring any new variables, and without using any global variables. Summarize your change(s) here and explain why it works.

3. Thinking Dynamically!

Switch to the “part3” directory.

In the lecture we learnt that processes have a section of memory called a “heap” for creating dynamic variables. In this part we will look at what dynamic variables are, and how to use them.

a. Creating and using Dynamic Variables

Open lab1p3a.c with your favorite editor and examine the code. You will notice several things:

- i. We do `#include <stdlib.h>`. This is to bring in the `malloc` and `free` function prototypes.
- ii. The `sizeof(.)` function returns the number of bytes of the type specified as its argument. So `sizeof(int)` returns the number of bytes in an `int`.
- iii. The `malloc(.)` function takes one argument; the number of bytes to allocate. The `malloc(.)` function then returns a pointer to the memory that was allocated.
- iv. The `malloc(.)` function’s return type is “`void *`”. While it seems strange to have a pointer to void, this is used in C to indicate a “generic” data type.
- v. We want to assign the return pointer to a variable “`z`” of type `int *`. Thus we need to type-cast the “`void *`” return type of `malloc(.)` to `int *`.
- vi. Overall this gives us the following statement allocate memory to store an `int`:

```
z = (int *) malloc(sizeof(int));
```

- vii. When we are done using the memory, we call free. For example:

```
free(z) ; // Frees memory pointed to by z.
```

Question 3.1 (1 mark)

Compile and run lab1part3a.c, and observe the addresses of x, y, z, p, and the memory returned by malloc. Notice that the address of the memory allocated by malloc is from a completely different range of addresses used by x, y, z and p. Explain why this is so.

b. Using valgrind

Valgrind is a very useful utility for detecting memory errors in your program, helping to find dynamic variables that were allocated and never freed (memory leaks), accessing memory that doesn't belong to a process leading to segmentation faults, and other types of errors.

Using valgrind is simple. If you have a program called "mine", simply type:

```
srun valgrind ./mine
```

Valgrind will then run your program and examine for leaks and other errors.

Open the file "lab1p3b.c", and you will see a program that simply allocates memory using malloc then frees it.

Compile the code using:

```
srun gcc -g lab1p3b.c -o lab1p3b
```

The "-g" option here causes gcc to generate debugging symbols so that valgrind can report line numbers in your code if there are errors.

Run valgrind:

```
srun valgrind ./lab1p3b
```

Since our code is very simple, there are no errors, and you get a beautiful output like this:

```

==277759== Memcheck, a memory error detector
==277759== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==277759== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==277759== Command: ./lab1part3b
==277759==
p is 0x49ea040 and *p is 5
==277759==
==277759== HEAP SUMMARY:
==277759==   in use at exit: 0 bytes in 0 blocks
==277759==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==277759==
==277759== All heap blocks were freed -- no leaks are possible
==277759==
==277759== For lists of detected and suppressed errors, rerun with: -s
==277759== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

We can see valgrind running our program (and printing the output of the program), then present a heap summary that shows that all heap blocks were freed and there are no memory leaks.

Now let's do something adventurous. You have a program called "lab1p3c.c" that has memory errors in it. Compile your program and run it:

```

srun gcc -g lab1p3c.c -o lab1p3c
srun ./lab1p3c

```

You will see that it terminates with a segmentation fault. We will now see why by running valgrind:

```

srun valgrind ./lab1p3c

```

Now witness the disaster that has happened:

```

==278234== Memcheck, a memory error detector
==278234== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==278234== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==278234== Command: ./lab1part3c
==278234==
ADDING PERSONS
Adding Tan Ah Kow aged 65
==278234== Use of uninitialised value of size 8
==278234==    at 0x484D368: strcpy (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==278234==    by 0x1089C7: makeNewNode (lab1part3c.c:12)
==278234==    by 0x108AB3: main (lab1part3c.c:35)
==278234==
==278234== Invalid write of size 1
==278234==    at 0x484D368: strcpy (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==278234==    by 0x1089C7: makeNewNode (lab1part3c.c:12)
==278234==    by 0x108AB3: main (lab1part3c.c:35)
==278234== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==278234==
==278234==
==278234== Process terminating with default action of signal 11 (SIGSEGV)
==278234== Access not within mapped region at address 0x0
==278234==    at 0x484D368: strcpy (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==278234==    by 0x1089C7: makeNewNode (lab1part3c.c:12)
==278234==    by 0x108AB3: main (lab1part3c.c:35)
==278234==
==278234== If you believe this happened as a result of a stack
==278234== overflow in your program's main thread (unlikely but
==278234== possible), you can try to increase the size of the
==278234== main thread stack using the --main-stacksize= flag.
==278234== The main thread stack size used in this run was 8388608.
==278234==
==278234== HEAP SUMMARY:
==278234==   in use at exit: 16 bytes in 1 blocks
==278234==   total heap usage: 2 allocs, 1 frees, 1,040 bytes allocated
==278234==
==278234== LEAK SUMMARY:
==278234==   definitely lost: 0 bytes in 0 blocks
==278234==   indirectly lost: 0 bytes in 0 blocks
==278234==   possibly lost: 0 bytes in 0 blocks
==278234==   still reachable: 16 bytes in 1 blocks

```

You will see that at line 12, we have a "Use of uninitialised value of size 8". This means that we've used some sort of uninitialized variable that is 8 bytes long inside strcpy.

Since strcpy is a standard library function it is unlikely to be buggy, so the problem is likely to be in your code. Examining line 12 we see:

```
9
10 TPerson *makeNewNode(char *name, int age) {
11     TPerson *p = (TPerson *) malloc(sizeof(TPerson));
12     strcpy(p->name, name);
13     p->age = age;
14
15     return p;
16 }
```

Since “name” is provided correctly (you can check main if you’re not convinced), the problem is likely to be p->name. Indeed you will see further down the valgrind report that you are trying to do an invalid write to p->name in line 12:

```
==279674== Invalid write of size 1
==279674==    at 0x484D368: strcpy (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_mencheck-arm64-linux.so)
==279674==    by 0x1089C7: makeNewNode (lab1part3c.c:12)
==279674==    by 0x108AB3: main (lab1part3c.c:35)
==279674== Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

If you look at TPerson, the “name” field is declared as char *, and when you malloc TPerson, name is set to NULL, giving us this problem. You need to allocate memory to copy the name.

Between lines 11 and 12, add the following:

```
p->name = (char *) malloc(strlen(name) + 1);
```

Your code should now look like this:

```
10 TPerson *makeNewNode(char *name, int age) {
11     TPerson *p = (TPerson *) malloc(sizeof(TPerson));
12     p->name = (char *) malloc(strlen(name) + 1);
13     strcpy(p->name, name);
14     p->age = age;
15 }
```

Notice that we allocate one extra byte for the ‘\0’. Recompile your program and run valgrind again, and you will see that the situation has improved tremendously:

```

==281489== Memcheck, a memory error detector
==281489== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==281489== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==281489== Command: ./lab1part3c
==281489==
ADDING PERSONS
Adding Tan Ah Kow aged 65
Adding Sio Bak Pau aged 23
Adding Aiken Dueet aged 21

DELETING PERSONS
Deleting Tan Ah Kow aged 65
Deleting Sio Bak Pau aged 23
Deleting Aiken Dueet aged 21
==281489==
==281489== HEAP SUMMARY:
==281489==   in use at exit: 35 bytes in 3 blocks
==281489==   total heap usage: 7 allocs, 4 frees, 1,107 bytes allocated
==281489==
==281489== LEAK SUMMARY:
==281489==   definitely lost: 35 bytes in 3 blocks
==281489==   indirectly lost: 0 bytes in 0 blocks
==281489==   possibly lost: 0 bytes in 0 blocks
==281489==   still reachable: 0 bytes in 0 blocks
==281489==   suppressed: 0 bytes in 0 blocks
==281489== Rerun with --leak-check=full to see details of leaked memory
==281489==
==281489== For lists of detected and suppressed errors, rerun with: -s
==281489== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

However now you have a memory leak; the Leak Summary says “Definitely lost: 35 bytes in 3 blocks”.

The University of South Carolina has a rather nice valgrind cheatsheet at <https://bytes.usc.edu/cs104/wiki/valgrind/> that explains what these sentences mean.

You can use this (or other documentation) to fix lab1part3c.c until valgrind shows no more memory leaks or errors, then answer the following question.

Question 3.2 (1 mark)

Summarize the changes to made to lab1part3c.c

Question 3.3 (1 mark)

Instead of using strcpy, can you still produce the correct string copy result with memcpy? If it is possible, paste your code here. What are the pros and cons?

c. Making a Double Linked List Library in C

We will now create a double-linked list library in C, that we will use in “part4” to implement a file directory. Throughout this lab, the phrase “linked list” both in this lab manual and in the source codes should be taken to mean “double linked list”.

A file directory is a persistent data structure used by the filesystem to store information about files on your computer. At a minimum we need to know the name of the file, the size of the file, and its starting location on the disk. Disks are organized into “blocks”, and we shall see what this means much later on in this class.

A double-linked list is like a standard linked list, except that in addition to the “next” pointer pointing to the next node, there is also a “prev” pointer pointing to the previous node.

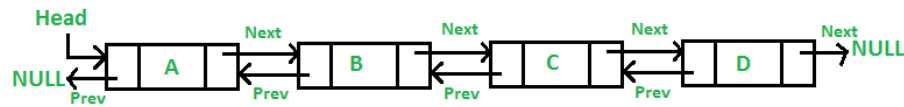


Figure 1. A double-linked list.

The “next” pointer of the last node and the “prev” pointer of the first node are both set to NULL. Double-linked lists make it very convenient to perform deletions, though at the expense of a slightly more complex insertion.

To help you complete the assignment, let’s look at “pointers to pointers”:

Pointers to Pointers

We are all familiar with pointers. For example:

```
int *ptr;
int x = 5;

ptr = &x; // ptr now points to 5.
```

We know that ptr is a pointer to an integer. But what about a pointer to a pointer to an integer? Since int *ptr is a pointer to a variable of type “int”, then a pointer to a variable of type “int *” would, unsurprisingly, be:

```
int **pptr;
pptr = &ptr;
```

Recall that in the above examples, to access the integer variable “x” using ptr, we would simply do:

```
y = *ptr; // y = x
```

We can similarly access ptr from pptr by using *pptr:

```
int *ptr2 = *pptr; // ptr2 = ptr
```

Why would we require pointers to pointers? The main reason is that it allows functions to change pointer variables that are passed to it. Recall that C passes parameters by value; thus, the only way to allow a function to change a pointer is to pass a pointer to the pointer.

One example might be when you are writing a function to allocate memory to store data. For example:

```
void alloc_mem(int **ptr) {
    *ptr = malloc(sizeof(int));
}

...
int *p;

// Allocate memory and assign to p
alloc_mem(&p);
```

We may similarly want to free memory pointed to by a pointer, and set that pointer to NULL:

```
void free_mem(int **ptr) {
    if(*ptr != NULL) {
        free(*ptr);
        *ptr = NULL;
    }
}
```

You are to implement the following functions. Some suggestions of how you can do each function is provided below. See the source file `llist.c` for descriptions of each function and of the parameters expected by each function, and what they're expected to do.

Function	Suggested Pseudocode
init	Set head to NULL
create_node	Create the node, copy over the filename, filesize and starting block, and return the node.
Insert_llist	If head is NULL, then head = node. Otherwise: Let trav=head. Traverse until trav->next is NULL Set trav->next = node and node->prev = trav
delete_llist	Adjust node->prev->next and node->next->prev accordingly. Free node.
find_llist	Traverse down the list until fname is found or the traverser becomes NULL. Return the traverser
traverse	Traverse down the list and call fn, until the traverser is NULL.

You need to add the appropriate prototypes to `llist.h`. There is a test program called "testlist.c". If you have implemented `llist.c` properly (with the correct prototypes in `llist.h`) and compile with `testlist.c`, you will see an output like this:

```

Initializing.
Traversing empty list

Entering filenames
Inserting test.txt
Inserting hello.txt
Inserting a.exe
Inserting c.exe
Inserting d.tmp
Inserting e.bin

Filenames after insertion:
Filename: test.txt Filesize: 32 Starting Block: 0
Filename: hello.txt Filesize: 172 Starting Block: 93
Filename: a.exe Filesize: 2384 Starting Block: 381
Filename: c.exe Filesize: 8475 Starting Block: 123
Filename: d.tmp Filesize: 8374 Starting Block: 274
Filename: e.bin Filesize: 283 Starting Block: 8472

Deleting list.
Deleting test.txt
Deleting hello.txt
Deleting a.exe
Deleting c.exe
Deleting d.tmp
Deleting e.bin

Printing after deletion:

```

Fix all memory errors that you find, including errors, if any, in code that was provided to you.

DEMO 2 - This demo is to be done on the week of 1 September 2025 (2 marks)

Run your program using valgrind to show to your TA that it has no memory leakages or errors. Open the llist.c program and show it to your TA, answering any questions that he might have.

4. Building a File Directory

Change to the “part4” directory, and copy over the COMPLETED llist.c and llist.h from your “part3” directory. Ensure that your implementation in llist.c works and gives the correct results first!

We shall now implement a file directory structure using the linked list we created earlier.

Since searching the linked lists is an $O(n)$ operation, we would like to reduce the lengths of the linked lists we search.

To do this, we will make use of a hash table. A hash table is a data structure in which we decide where to place an item (in this case the information about a file) in an array, by applying a “hashing function”. Here we would apply the hashing function to the filename, which would produce an index of 0 to $n-1$, for an n -row hash table.

The simplest hash function “hashfun(filename, n)” would be to sum up all the ASCII values in “filename”, then modulo n. This is the hash function we will use as it is the simplest. There are other hash functions possible.

It is possible for different names to hash to the same row, so we will use a linked list to store all the file details of files with names that map to the same row.

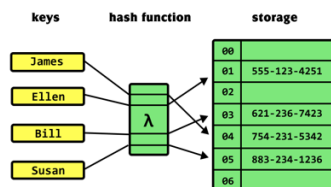


Figure 2. An Example Hash Table

The choice of hash functions has an impact on performance because some hash functions will result in mapping more files to the same table entry (“collisions”) than others, thus leading to longer linked lists that take longer to search. However, this is beyond the scope of this lab.

You need to implement the following functions in dir.c, using the functions created in llist, the helper functions writelog, get_filelist and update_hashtable, and anything else, in order to implement the directory structure for our “filesystem”.

Function	Suggested Pseudocode
init_hashtable	Iterate over every entry of hashtable and set it to NULL
find_file	Get the file list for the given filename, then search the file list for that filename using llist functions. Return NULL if not found.
add_file	If file exists return an error. Otherwise create a new node with the file data, then insert it into the appropriate file list, using llist functions.
delete_file	If file does not exist, return an error. Otherwise locate the appropriate file list, and delete the node using llist functions.
rename_file	Find the file, return error if not found. Otherwise, rename the file (see question 4.1 below).
listdir	Iterate over every entry in hashtable, then call the appropriate function in llist to print out all the entries. Printing does not have to be in any order.

You must add in the appropriate prototypes into dir.h.

Question 4.1 (1 mark)

For renaming the file, you cannot simply find the appropriate file list using `get_filelist`, then locate the appropriate node, and just do `strcpy(node->filename, new_filename)`. Why not? What is the correct way to perform the file renaming? Give a brief pseudocode here.

Question 4.2 (1 mark)

In `testdir.c`, `listdir()` function takes in two argument `hash_table` and `TABLE_LEN`. Why do we need to pass `TABLE_LEN` as a parameter here?

There is a file called `testdir.c`. If your `dir.c` functions are implemented correctly (with the correct prototypes in `dir.h`), compiling it with `testdir.c` and `l1st.c` and running it should produce an output like this:

```
Adding files.
Listing files.
Filename           File Size      Start Block
e.bin              283           8472
a.exe              2384          381
test.txt           32            0
hello.txt          172           93
c.exe              8475          123
d.tmp              8374          274

Searching for existing file a.exe
OK!
Filename: a.exe File size: 2384 Starting Block: 381

Searching for non-existing file
OK. File not found.

Adding the missing file
Ok, file found.

Renaming bork.jpg to work.jpg

Searching for bork.jpg
OK. File not found.

Searching for work.jpg
Ok, file found.

Searching for file Before deleting:
OK! Found file!

Deleting file hello.txt
OK! File no longer found!
```

In particular, you should not see any output that says “ERROR”.

DEMO 3 – This demo is to be done on the week of 1 September 2025 (2 marks)

Compile `testdir.c` creating an output executable file called `testdir`, in a way that `testdir` can be checked with `valgrind`. Run `testdir` with `valgrind` and demonstrate it to your TA, showing that there are no memory leakages or errors.