CS2106 Introduction to Operating Systems

Lab 1 - Leveling Up on C

Week of 26 January 2026 (Initial Release)

Week of 2nd Feb, 2026 (Demos – Segment A)

Week of 9th Feb, 2026 (Demos – Segment B)

Common Instruction for All Labs

Each of the four labs follows the same arrangement. Please read the following carefully and clear your doubts with instructor or lab TA in the first lab.

a. Each lab will be released on Week X, followed by two "demo weeks" in Week X+1 and Week X+2. Each lab group (~20 students) will be split into two segments A and B. The students in segment A will demo their work in "Demo Week One", while the students in segment B will demo their work in "Demo Week Two". This pattern is then reversed in the next lab for fairness.

   The lab schedule is as follows:

Lab #	Release in week	Demo Week One | Demo Segment	Demo Week Two | Demo Segment
#01	3	4 | A	5 | B
#02	6	7 | B	8 | A
#03	8*	9 | A	10 | B
#04	11	12 | B	13 | A

   Note*: Lab #03 is the only lab with a slight overlap with earlier lab (#02). Please be more diligent and plan your work accordingly.

b. Lab assignments will be graded **only via in person demonstration in the Demo Weeks**. Each lab weightage is **7 marks**, with the lab total weightage at **25 marks**. Since (7 marks x 4 labs = 28 marks), the excess 3 marks act as **buffer**. i.e. you can afford to make some mistakes (3 marks worth to be exact) in labs without compromising the total score. If you scored above 25 marks in the lab component, the excess marks do not carry over to other CA components. In summary, your lab score at the end of semester is **`min( your total score, 25)`**.

c. You will work in **pairs** to complete the lab assignments, i.e. work with another student in the same demo segment of your lab group. [For lab session with odd number of students, one student group may have 3 members instead of two]. Please attend the first lab session in week #03 to sort out the segment and pairing arrangement. For lab demonstration, lab TA will ask each of the students in the group about different parts of the lab assignment. Make sure everyone in your group has good understanding of your group's solution before the demonstration.

Introduction

This course assumes that you have basic knowledge of C programming. 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. Many of the mechanisms introduced help to illustrate ideas from the lecture and / or used in OS implementations in the real world.

This lab is worth 7 marks across 3 demos. Each demo requires you to show a correctly working C program. Note that having a working C program in each of the demo part is a **pre-requisite for Q&A with the TA**, i.e. if your program doesn't work as intended, the tutor will not award any marks for that demo part. Your TA will ask 2-3 questions per demo part. Note that for some questions, your TA may give partial grades depending on your answer.

0. Uploading and Unpacking your Program Files

If you want to use the compute cluster. Otherwise, just download the files to where you want to run them on your local linux machine and unzip the files.

a. Copy lab1code.zip over to your home directory on xlogin:

   scp lab1code.zip <userid>@xlogin.comp.nus.edu.sg:~

b. ssh to xlogin:

   ssh <userid>@xlogin.comp.nus.edu.sg

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

   mkdir L01; cd L01

d. Unzip lab1code.zip (which is presumably one directory level up in your home directory):

   srun unzip ../lab1code

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 stack.

a. Using your favorite editor, open “stack.c” and examine the code.

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 current directory. In contrast, the other #include <stdio.h> - where does the compiler look for the stdio.h file?

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_SIZE 10
```

Open “lab1p1.c” and you will see:

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

int main() {

        int times = 3;
        int val = 42;
        printf("Adding %d times the value %d\n", times, val);
        push_multiple(val, times);

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

        //This will lead to an error. Why?
        //extern int _top;
        //printf("The size of the stack currently is %d\n", size());
        //Make changes to stack.c to allow us to get the size of the stack here.

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

Notice how MAX_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 xlogin.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)

```
srun 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:

```
srun ./lab1p1
```

On a normal Linux machine:

```
./lab1p1
```

(Note: When you run this program, you will notice that the last 3 elements cannot be inserted as the stack is full, and when reading the last 3 elements 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)

You will notice that you get the following error (**Note:** Some compiler versions may return a warning instead of an error):

```
lab1p1.c: In function 'main':
lab1p1.c:9:9: error: implicit declaration of function 'push_multiple' [-Wimplicit-function-declaration]
    9 |         push_multiple(val, times);
      |         ^~~~~~~~~~~~~~
lab1p1.c:13:17: error: implicit declaration of function 'push' [-Wimplicit-function-declaration]
   13 |                 push(i);
      |                 ^~~~
lab1p1.c:22:25: error: implicit declaration of function 'pop' [-Wimplicit-function-declaration]
   22 |                 int v = pop();
      |                         ^~~
```

e. We will now fix the warning/error 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 enq and deq 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 use the stack). Now THAT makes sense!

So now, add the function prototypes to stack.h. Throughout the rest of this lab, you will need to add many more function prototypes to the header files.

f. A quite natural requirement for the stack is knowing how many elements are contained in it. If you look at lab1p1.c, you will notice that it tries to print out how many elements are in the stack by accessing the _top variable using the extern keyword. If you try to compile it, it will produce the following error.

```
/usr/bin/ld: /tmp/ccoxeSMf.o: warning: relocation against `_top' in read-only section `.text'
/usr/bin/ld: /tmp/ccoxeSMf.o: in function `main':
lab1p1.c:(.text+0x7d): undefined reference to `_top'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
collect2: error: ld returned 1 exit status
```

You can look up for what prevents other files from accessing _top in stack.c. This ties in with the idea of encapsulation in programming. The best way to allow lab1p1.c get the size of the stack would be to add a function that returns the size of the stack.

g. 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:

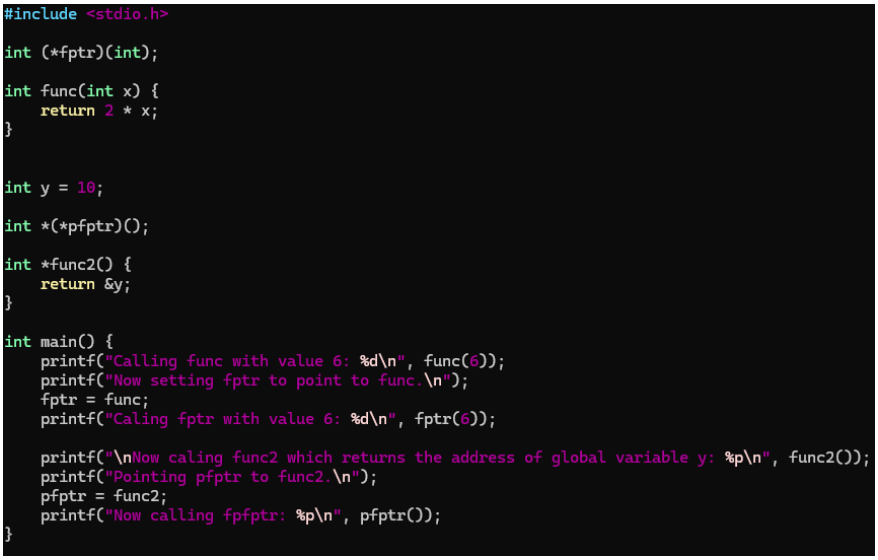
```
int *func(int x) {
      …
}
```

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:



```c
#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("Caling fptr with value 6: %d\n", fptr(6));

    printf("\nNow caling func2 which returns the address of global variable y: %p\n", func2());
    printf("Pointing pfptr to func2.\n");
    pfptr = func2;
    printf("Now calling fpfptr: %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 and 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 map function, which applies an input function over every element in the stack (typically a map function over a list of items returns the results in a new list, but for simplicity, let's ignore that for now). Open stack.c again and you will see some functions related to function pointers near the bottom of the file.

Function	**Purpose**
int add(int x, int y)	Returns x + y
int multiply(int x, int y)	Returns x * y
int modulo(int x, int y)	Returns x % y
void map_addition(int val)	Calls sum over each members of the stack to increment each member of the stack by val.

Load up testr.c, and you will see some code that fills the stack with 1.0 to 9.0, then a call to map_addition which increments each element in the stack with the same value. There are also two commented-out lines for you to test your flex_map function later.

Now do the following:

i. Add in the prototypes for add, multiply, modulo and map_addition 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:

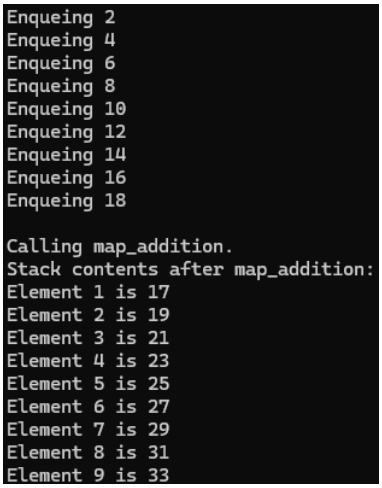
```
srun ./testr
```

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:



We want to create a new function called "flex_map" that takes in two arguments: op and value where op is a function pointer, and value is an integer. The function flex_map should apply op with value as argument to every element in the stack.
The pseudo-code for flex_map is shown here:

```
void flex_map(op, value) {
      for every element in stack:
            Call op with element and value, writing
            the return value back to the stack.

}
```

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

Now the statements in testr.c to test flex_map, and compile your program.

DEMO 1 - 2 marks

Run your code to show that flex_map works. Show the code to your TA, and answer his / her questions.

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
```

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	=1	0x6323872dd018
p2	1	0x6323872dd020
p3	1	0x6323872dd028
p4	1	0x6323872dd030
w	42(5)	0x6323872dd038
x	2	0x7ffe5643881c
y	2	0x7ffe56438818
z	2	0x7ffe56438824
f	2	
g	2	
h	2	

Using the observations above and your understanding of the 4 different regions of memory usage (Text, Data, Stack and Heap), can you tell where is each variable resided?

b.

Notice that the contents of x, y and z may be changed ("corrupted") when we exit fun1. 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".

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:

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

Based off what you have seen in this part, try fixing lab1p2b.c to produce the correct result, without declaring any new variables, and without using any global variables.

DEMO 2 - 2 marks

Run your corrected `lab1p2b` to show that accumulate now work as intended. Show the code to your TA, and answer any question(s) that he or she might have. Your lab TA may ask questions about both `lab1p2a` and `lab1p2b`.

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.
```

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.

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:

```
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:

```
==559== Memcheck, a memory error detector
==559== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==559== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==559== Command: ./a.out
==559==
p is 0x4a74040 and *p is 5
==559==
==559== HEAP SUMMARY:
==559==     in use at exit: 0 bytes in 0 blocks
==559==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==559==
==559== All heap blocks were freed -- no leaks are possible
==559==
==559== For lists of detected and suppressed errors, rerun with: -s
==559== 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:

```
==575== Memcheck, a memory error detector
==575== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==575== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==575== Command: ./a.out
==575==
ADDING PERSONS
Adding Tan Ah Kow aged 65
==575== Use of uninitialised value of size 8
==575==    at 0x484F38C: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==575==    by 0x10921F: makeNewNode (lab1p3c.c:12)
==575==    by 0x109322: main (lab1p3c.c:35)
==575==
==575== Invalid write of size 1
==575==    at 0x484F38C: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==575==    by 0x10921F: makeNewNode (lab1p3c.c:12)
==575==    by 0x109322: main (lab1p3c.c:35)
==575==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==575==
==575==
==575== Process terminating with default action of signal 11 (SIGSEGV)
==575==  Access not within mapped region at address 0x0
==575==    at 0x484F38C: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==575==    by 0x10921F: makeNewNode (lab1p3c.c:12)
==575==    by 0x109322: main (lab1p3c.c:35)
==575==  If you believe this happened as a result of a stack
==575==  overflow in your program's main thread (unlikely but
==575==  possible), you can try to increase the size of the
==575==  main thread stack using the --main-stacksize= flag.
==575==  The main thread stack size used in this run was 8388608.
==575==
==575== HEAP SUMMARY:
==575==     in use at exit: 1,040 bytes in 2 blocks
==575==   total heap usage: 2 allocs, 0 frees, 1,040 bytes allocated
==575==
==575== LEAK SUMMARY:
==575==    definitely lost: 0 bytes in 0 blocks
==575==    indirectly lost: 0 bytes in 0 blocks
==575==      possibly lost: 0 bytes in 0 blocks
==575==    still reachable: 1,040 bytes in 2 blocks
==575==         suppressed: 0 bytes in 0 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:

```
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:

```
Invalid write of size 1
   at 0x484F38C: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x10921F: makeNewNode (lab1p3c.c:12)
   by 0x109322: main (lab1p3c.c:35)
 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:

```
==610== Memcheck, a memory error detector
==610== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==610== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==610== Command: ./a.out
==610==
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
==610==
==610== HEAP SUMMARY:
==610==     in use at exit: 35 bytes in 3 blocks
==610==   total heap usage: 7 allocs, 4 frees, 1,107 bytes allocated
==610==
==610== LEAK SUMMARY:
==610==    definitely lost: 35 bytes in 3 blocks
==610==    indirectly lost: 0 bytes in 0 blocks
==610==      possibly lost: 0 bytes in 0 blocks
==610==    still reachable: 0 bytes in 0 blocks
==610==         suppressed: 0 bytes in 0 blocks
==610== Rerun with --leak-check=full to see details of leaked memory
==610==
==610== For lists of detected and suppressed errors, rerun with: -s
==610== 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.

b. 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.

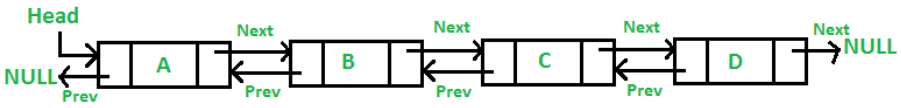


Figure 1. A double-linked list.

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 expcted 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:
```

DEMO 3 – 3 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 [For exploration only]

This exercise is provided your own exploration only and will not be graded / demonstrated. The idea (file directory) will be revisited in more depth in later part of the course.

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.

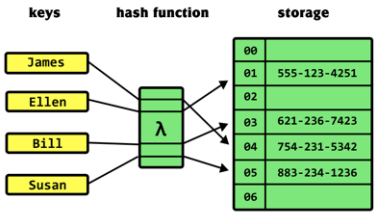


Figure 2. An Example Hash Table

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.

Another issue that naturally has an impact on performance is the size of the hashtable. If the number of files stored is much larger than the size of the hashtable, the linked lists must become longer as files necessarily map to the same slots. In practice, hashtables will dynamically resize themselves if the number of items stored in the hashtable exceeds some factor of its size. Therefore, we want to be able to allow resizing the hashtable. (Hashtables also get resized smaller if they are sparsely populated to save memory use).

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.
resize_hashtable	For each existing file, find the correct slot in the new hashtable, and place the file inside the appropriate linked list. The files can be inserted into the new hashtable in any order, as long as they are all correctly added to the new hashtable. Then, free the old hashtable and return a pointer to the array that is the new hashtable.
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 the appropriate prototypes into dir.h.

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 llist.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.

Searhing 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”.