

CS2106 Introduction to Operating Systems
Lab 1 - Leveling Up on C
Week of 24 January 2022 (Initial Lab)
Week of 31 January 2022 (Demos)

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.
- c. All submissions will be made to the Files->Lab Submissions->Lab 1 Submissions folder by **Sunday, 6 February 2022, 11.59 pm**. The folder will remain open until 12.10 am on 7 February 2022 latest, after which NO SUBMISSIONS will be accepted.
- d. There are demos in Parts 1 and 3 of this lab. The demos will take place on the week of 31 January 2022. Demos are done **individually** with your respective TA even if you did the project with a partner.
- e. **This being CNY Week, there are disruptions to labs from Monday 31 January 2022 to Wednesday 2 February 2022. Your TAs will contact you to arrange for make-up sessions where you can do the demos.**
- f. This lab is worth 15 marks; 13 marks for the written portion, 2 marks for the demos.
- g. Download the Lab1Programs.zip file, and transfer it to sunfire, then to one of the xcne compute cluster servers. You should do your labs on these compute cluster servers and all demos will be done there.
- h. Unzip Lab1Programs.zip to get the source codes for the rest of the lab. You will see three directories labeled part1, part2 and part3, corresponding to the parts shown below.

All questions are worth 1 mark. Tutors may assign ½ mark if there are errors, or 0 marks if the answer is completely wrong.

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 “queue.c” and examine the code.

(Quick note: The circular queue as implemented uses a “lazy” check for a full queue, resulting in the queue storing one element less than its declared size. We will ignore this fact.)

Question 1.1 (1 mark)

At the start of queue.c you will see:

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

What is the difference between using < and > vs "" (double quotes) in our #include statement?

Question 1.2 (1 mark)

You will notice that _queue, _front and _rear are declared as “static”. What does the “static” declaration mean?

- b. Using your favorite editor again, open “queue.h” and you will see just a single line:

```
// Maximum number of elements in a queue
//
#define MAX_Q_SIZE 10
```

Open “lab1p1.c” and you will see:

```

#include <stdio.h>
#include "queue.h"

int main() {
    double v;

    for(int i = 0; i<= MAX_Q_SIZE; i++) {
        v = ((double) i / 10.0);
        printf("Adding %3.2f\n", v);
        enq(v);
    }

    for(int i = 0; i<= MAX_Q_SIZE; i++) {
        v = deq();
        printf("Element %d is %3.2f\n", i, v);
    }
}

```

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

- c. We will now compile our program. To do so, switch to the directory containing queue.c, queue.h and lab1p1.c and type:

```
gcc queue.c lab1p1.c -o lab1p1
```

(Note: When you run this program, you will notice that the last 2 elements cannot be inserted as the queue is full, and when reading the last 2 elements cannot be read because the queue is empty. This is normal and it's meant to test that the queue can detect full and empty conditions)

Question 1.3 (1 mark)

You will notice that you get the following warning:

```

lab1p1.c: In function 'main':
lab1p1.c:10:3: warning: implicit declaration of function 'enq' [-Wimplicit-function-declaration]
    enq(v);
    ^~~
lab1p1.c:14:7: warning: implicit declaration of function 'deq' [-Wimplicit-function-declaration]
    v = deq();
    ^~~

```

What do they mean?

Notice however that despite the warnings, the executable file lab1p1 was still produce. This program will fill the queue with numbers from 0.0 to 0.8 (0.9 and 1.0 will exceed the queue capacity and not be enqueued), then read back the values it stored.

Question 1.4 (1 mark)

You will notice that in the second “for” loop we get back incorrect values from the queue. Explain why. (Hint: It has something to do with the warnings in Question 3.)

- d. We will now fix the problems in Questions 1.3 and 1.4 by creating “function prototypes” in “queue.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 queue.c. This is just silly since we don’t #include queue.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 queue.c elsewhere?
- iii. Define the prototypes inside queue.h, which will be #include into lab1p1.c (and anywhere else you want to use enq and deq). Now THAT makes sense!

Question 1.5 (1 mark)

Add the function prototypes inside queue.h. Recompile lab1p1.c and run it. Are there anymore warnings? Does the second for-loop now get the correct values?

- e. 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 begin first by seeing how to declare a function pointer. Let's begin first by declaring:

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

This, as we know, is a function that returns a pointer to an "int" and takes an "int" as argument. Contrast this example with:

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

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. Notice how, like arrays, the name of a function is also the pointer to the function itself, and thus we can just do:

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

And NOT:

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

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

Function	Purpose
clear_sum()	Clears the _res variable by setting it to 0.
clear_prod()	Clears the _res variable by setting it to 1.
sum(double x)	Adds x to _res
prod(double x)	Multiplies _res by x
reduce()	Calls clear_sum() to set _res to 0, then calls sum to sum over all members of the queue and returns the result in res.

Load up testr.c, and you will see some code that fills up the queue with 1.0 to 9.0, then a call to reduce() to sum up the contents of the queue, 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:

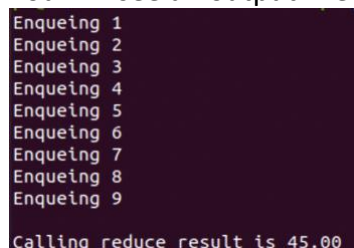
- i. Add in the prototypes for clear_sum, clear_prod, sum, prod and reduce to queue.h
- ii. Compile testr.c and queue.c using:

```
gcc testr.c queue.c -o testr
```

- iii. Run testr:

```
./testr
```

- iv. You will see an output like this:



```
Enqueing 1  
Enqueing 2  
Enqueing 3  
Enqueing 4  
Enqueing 5  
Enqueing 6  
Enqueing 7  
Enqueing 8  
Enqueing 9  
Calling reduce result is 45.00
```

- 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 queue.
- vi. The pseudo-code for flex_reduce is shown here:

```
double flex_reduce(clear, op) {  
    clear(); // Clear _res to either 0 or 1  
    for every element in queue:  
        Call op with element.  
  
    return _res;  
}
```

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

Now uncomment the two statements in testr.c to test flex_reduce, and compile your program.

DEMO 1 – This demo is to be done at the lab on the week of 31 January 2022 (1 mark)

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 life-time 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:

```
gcc lab1p2a.c -o lab1p2a  
./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. Explain why this is so.

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:

```
gcc lab1p2b.c -o lab1p2b
./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! (Directory: part3)

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:

```
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:

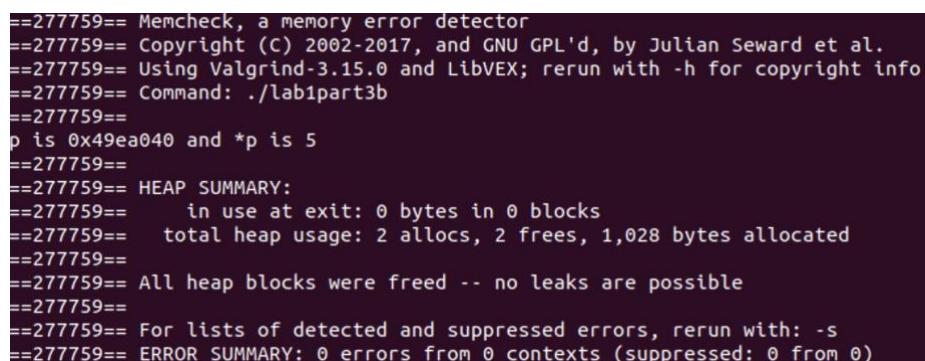
```
gcc -g lab1p3b.c -o lab1p3b
```

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:

```
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:

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

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

```
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: ./lab1p3c
==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== 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_memcheck-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

c. Building a Small Phone Book

We are now going to use dynamic memory to create a telephone book. The telephone book will store its data in a binary search tree (BST). The BST insert, search, in-order traversal and delete and other utility functions are in bintree.c, with header file bintree.h.

The phonebook.c file contains routines to add a new person, delete a person, search for a person, and delete the entire phonebook. It comes with its header file phonebook.h.

Finally you have testpb.c, which contains routines to test your phonebook in main.

The code in phonebook.c, phonebook.h and testpb.c is complete; you only have to complete the code in bintree.c and bintree.h.

Open up bintree.h, and you will the main tree structure:

```
typedef struct tn {
    char *name;
    char phoneNum[9];
    struct tn *left, *right;
} TTreeNode;
```

This is a simple structure containing:

- "name": A pointer to a string containing the person's name, which also serves as the key for inserting/searching the BST.
- "phoneNum": An 8-character string of the person's phone number.
- "left" and "right": Pointers to the left and right nodes in the BST.

Using the information above, you need to implement the following functions:

Function	Purpose
<code>TTreeNode *makeNewNode(char *name, char *phoneNum)</code>	Creates and returns a new node containing “name” and “phoneNum”, which are the name and phone number of the new person being inserted.
<code>void addNode(TTreeNode **root, TTreeNode *node)</code>	Adds a new node “node” to the tree at root “root”. Note that “root” is a pointer to the tree’s root, not the root itself.
<code>void delTree(TTreeNode *root)</code>	Delete a tree whose root is pointed to by “root”.
<code>void freenode(TTreeNode *node)</code>	Frees the memory allocated to “node”.
<code>void print_inorder(TTreeNode *node)</code>	Does an in-order printing of the (sub-)tree pointed to by “node”. (Hint: Best done using recursion).

To help you complete the assignment, let’s look at some details and hints:

i. 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;
    }
}
```

ii. Creating a New Node (makeNewNode)

This is fairly straightforward; you can just use malloc to create your new node. Note however that you might need to malloc the name in TTreeNode as well. ☺

Also, when allocating memory for a string, remember to allocate one extra byte for the '\0'.

iii. Inserting into a BST

Insertion into a BST can follow this algorithm:

Input: “newNode”, a node that was created using makeNewNode

- a. Let “root” be the root of the BST.
 - b. if root == NULL, root = newNode.
 - c. Let trav = root.
- While(true)
- (1) If trav->name < newNode->name
 1. If trav->right == NULL, trav->right = newNode. Exit.
 2. Else trav = trav->right
 - (2) Else:
 1. if trav->left == NULL, trav->left = newNode. Exit.
 2. Else trav = trav->left

iv. Deleting the Entire Tree (delTree)

It’s easiest to do so recursively by doing tree traversal. Of the three traversals (pre-order, in-order and post-order), one works particularly well.

v. Freeing a Node (freenode)

If you allocated space for “name”, remember to free it as well.

vi. In-Order Printing (print inorder)

In-order printing makes use of in-order traversal to print the “name” and “phoneNum” details in the tree, using recursion.

Using the information above, complete the code in bintree.c, and answer the following questions:

Question 3.3 (1 mark)

What command do you use to compile your phonebook to the executable file “testpb”, so that you can test it with valgrind? Write the full command here:

DEMO 2 – This demo is to be done on the week of 31 January 2022 (1 mark)

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